

CS300 Couchbase NoSQL Server Administration

Lab 5.1 Exercise Manual



Release: 6.5.1

Revised: June 22nd, 2020



Lab #5: Eventing and Analytics

Objective: This 1-hour lab will first walk you through adding nodes #7 and #8 into the cluster. Then create and use N1QL indexes.

Overview: The following high-level steps are involved in this lab:

- Add nodes #7 and #8 into cluster

Add Nodes #7 and #8 into the Cluster:

Follow instructions for adding nodes to a cluster contained in Lab3 if you need explicit instructions.

Nodenames are as follows:

Couchbase07

Couchbase08

For nodes to be used in this lab use the nodes designated as

Server_1 of cluster 2

Server 2 of cluster 2

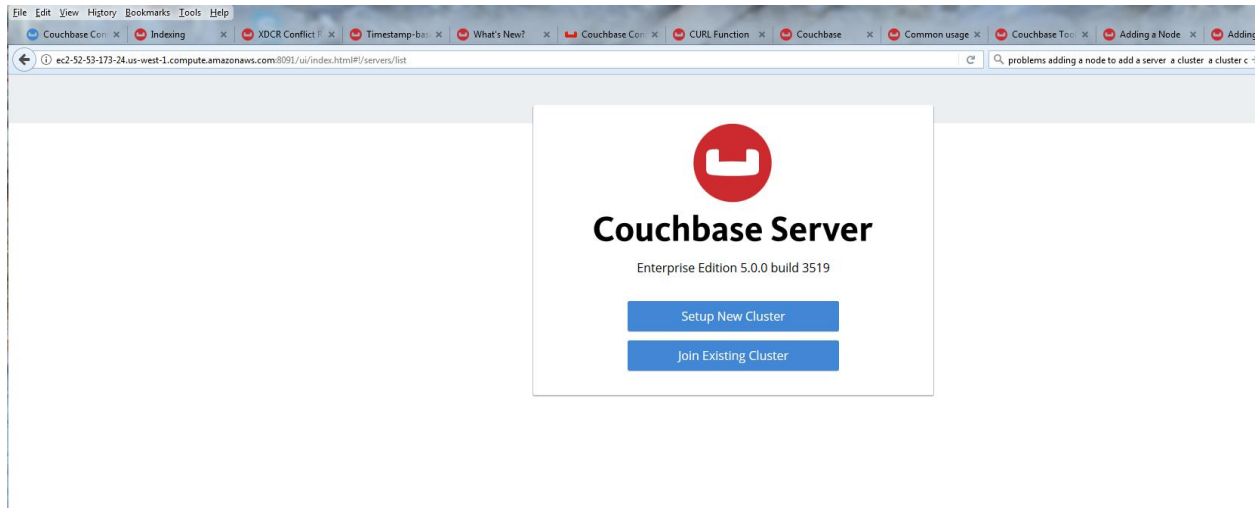
[In your spreadsheet.](#)

Point your browser to the URL for node 7:

<http://ec2-13-57-48-149.us-west-1.compute.amazonaws.com:8091/ui/index.html>



Lab-5: Eventing and Analytics page 3

**Choose****Join Existing Cluster**


Fill in the information required making sure to use the Amazon ec2 address for both cluster and joining node name.

Select analytics and eventing service(deselect other services)

Edit analytics path to /opt/couchbase/var/lib/couchbase/analytics



Lab-5: Eventing and Analytics page 4

 Couchbase > Join Cluster

Cluster Host Name/IP Address

Cluster Admin Username

Cluster Admin Password

▼ Configure Services & Settings For This Node


☐ Data

☐ Index

☐ Search

☐ Query

☒ Eventing

☒ Analytics 

This Node: Host Name/IP Address Usually localhost or similar

Data Disk Path Path cannot be changed after setup

Free: 17 GB

Indexes Disk Path Path cannot be changed after setup

Free: 17 GB

Analytics Disk Paths Paths cannot be changed after setup

Free: 17 GB + -

[< Back](#) [Join With Custom Configuration](#)

Click[Join With Custom Configuration](#)

Repeat this procedure for Node #8



Lab-5: Eventing and Analytics page 5

You should now see as ADD Pending Rebalance.

name	group	services	CPU	RAM	swap	disk used	Items	
ec2-54-152-103-168.compute-1.amazonaws.com	Group 1	data	5.55%	42.9%	---	43.7MB	2015/1917	Statistics
ec2-54-152-22-85.compute-1.amazonaws.com	Group 1	data	5.52%	43.2%	---	42MB	1977/1990	Statistics
ec2-54-158-72-58.compute-1.amazonaws.com	Group 2	data	7.03%	43%	---	41.6MB	1938/1963	Statistics
ec2-54-161-83-194.compute-1.amazonaws.com	Group 2	data	7.61%	36.8%	---	33.6MB	1969/2029	Statistics
ec2-54-163-43-233.compute-1.amazonaws.com	Group 1	index query search	3.55%	28.6%	---	---	0/0	Statistics
ec2-54-164-165-62.compute-1.amazonaws.com	Group 2	index query search	2.04%	28.9%	---	---	0/0	Statistics
ec2-54-164-254-223.compute-1.amazonaws.com	Group 1	analytics eventing	1.5%	19.7%	---	---	0/0	Statistics
New node Not taking traffic ADD pending rebalance								Cancel Add
ec2-54-166-172-102.compute-1.amazonaws.com	Group 1	analytics eventing	1%	19.4%	---	---	0/0	Statistics
New node Not taking traffic ADD pending rebalance								Cancel Add

Click the Rebalance button

Rebalance

The screenshot shows the Couchbase Server Administration Console for a 6 Node Cluster. The 'Servers' tab is selected, displaying a table of server details. A modal dialog in the top right corner indicates that a rebalance operation is in progress, showing 'rebalancing 6 nodes 33.3%' with a progress bar.

name	group	services	CPU	RAM	swap	disk used	Items
ec2-13-56-207-54.us-west-1.compute.amazonaws.com	Group 1	data	82.1%	27.3%	---	16.9MB	5/5
ec2-13-57-48-149.us-west-1.compute.amazonaws.com	Group 1	data	78.8%	18.3%	---	10.7KB	0/0
ec2-52-53-173-24.us-west-1.compute.amazonaws.com	Group 1	data	80.8%	20.6%	---	10.7KB	0/0
ec2-54-183-85-83.us-west-1.compute.amazonaws.com	Group 1	data	75.6%	30.7%	---	19.6MB	5/5
ec2-54-193-62-173.us-west-1.compute.amazonaws.com	Group 2	full text index query	65.6%	24.7%	---	---	0/0
ec2-54-193-79-108.us-west-1.compute.amazonaws.com	Group 1	full text index query	67.7%	27.2%	---	---	0/0

The rebalance operation will start running:



Go to the servers links and then to the groups link and place one of the newly joining nodes(8) into group 2

6 Node Cluster > Servers > Server Groups

Dashboard Servers Buckets Indexes Search Query XDCR Security Settings Logs

Group 1 [edit name](#)

ec2-13-56-207-54.us-west-1.compute.amazonaws.com	Data	move to ▼
ec2-13-57-48-149.us-west-1.compute.amazonaws.com	Data	✓ pending move to Group 2 cancel
ec2-52-53-173-24.us-west-1.compute.amazonaws.com	Data	✓ pending move to Group 2 cancel
ec2-54-183-85-83.us-west-1.compute.amazonaws.com	Data	move to ▼
ec2-54-193-79-108.us-west-1.compute.amazonaws.com	Full Text Index Query	move to ▼

Group 2 [edit name](#)

ec2-54-193-62-173.us-west-1.compute.amazonaws.com	Full Text Index Query	move to ▼
---	-----------------------	-----------

Reset [Apply Changes](#) [Add Group](#)

Click

[Apply Changes](#)

Eventing

Goal: A document contains attributes whose form makes them difficult to search on. Therefore, on the document's creation or modification, a new attribute should be created to accompany each original attribute; this new attribute being instantiated with a value that directly corresponds to that of its associated, original attribute; but takes a different form, thereby becoming more supportive of search. Original attributes are subsequently retrievable based on successful searches performed on new attributes.

Implementation: Create a JavaScript function that contains an **OnUpdate** handler. The handler listens for data-changes within a specified, source bucket. When any document within the bucket is created or mutated, the handler executes a user-defined routine. In this example, if the created or mutated document contains two specifically named fields containing IP addresses (these respectively corresponding to the beginning and end of an address-range), the handler-routine converts each IP address to an integer. A new document is created in a specified, target bucket: this new document is identical to the old, except that it has two additional fields, which contain integers that correspond to the IP addresses. The original document, in the source bucket, is not changed.



Preparations

This example requires the creation of three buckets: `metadata_bucket`, `target_bucket` and `source_bucket` buckets.

Proceed as follows:

1. Create target and metadata buckets. To create a bucket, refer to [Create a Bucket](#). The target bucket contains documents that will be created by the Function. Don't add any documents explicitly to this bucket.
2. Follow Step 1. and create a source bucket. In the Source bucket screen:
 - a. Click **Add Document**.
 - b. In the **Add Document** window, specify the name ***SampleDocument*** as the **NewDocument ID**
 - c. Click **Save**.
 - d. In the **Edit Document** dialog, paste the following within the edit panel.

```
{  
  
  "country": "AD",  
  
  "ip_end": "5.62.60.9",  
  
  "ip_start": "5.62.60.1"  
}
```

- e. Click **Save**. This step concludes all required preparations.



Procedure

Proceed as follows:

1. From the **Couchbase Web Console > Eventing** page, click **ADD FUNCTION**, to add a new Function.
2. In the **ADD FUNCTION** dialog, for individual Function elements provide the below information:
 - For the Source Bucket drop-down, select the *source option* that was created for this purpose.
 - For the Metadata Bucket drop-down, select the *metadata option* that was created for this purpose.
 - Enter **enrich_ip_nums** as the name of the Function you are creating in the **FunctionName** text-box.
 - Enter **Enrich a document, converts IP Strings to Integers that are stored in new attributes**, in the **Description** text-box.
 - For the **Settings** option, use the default values.
 - Click the plus button for Bindings
 - For the Bindings option, specify **bucket alias** and **tgt** as the alias of the bucket, and specify **target** as the associated bucket value, **read and write** as access

3. After providing all the required information in the **ADD FUNCTION** dialog, click **Next: Add Code**.



The **enrich_ip_nums** dialog appears. The **enrich_ip_nums** dialog initially contains a placeholder code block. You will substitute your actual **enrich_ip_nums** code in this block.



4. Copy the following Function, and paste it in the placeholder code block of the **enrich_ip_nums** dialog:

```
function OnUpdate(doc, meta) {
  log('document', doc);
  doc["ip_num_start"] = get_numip_first_3_octets(doc["ip_start"]);
  doc["ip_num_end"]   = get_numip_first_3_octets(doc["ip_end"]);
  tgt[meta.id]=doc;
}

function get_numip_first_3_octets(ip)
{
  var return_val = 0;
  if (ip)
  {
    var parts = ip.split('.');

    //IP Number = A x (256*256*256) + B x (256*256) + C x 256 + D
    return_val = (parts[0]*(256*256*256)) + (parts[1]*(256*256)) +
(parts[2]*256) + parseInt(parts[3]);
    return return_val;
  }
}
```

After pasting, the screen appears as displayed below:



```

1- function OnUpdate(doc, meta) {
2   log('document', doc);
3   doc["ip_num_start"] = get_numip_first_3_octets(doc["ip_start"]);
4   doc["ip_num_end"]   = get_numip_first_3_octets(doc["ip_end"]);
5   tgt[meta.id]=doc;
6 }
7
8 function get_numip_first_3_octets(ip)
9- {
10  var return_val = 0;
11  if (ip)
12  {
13    var parts = ip.split('.');
14
15    //IP Number = A x (256*256*256) + B x (256*256) + C x 256 + D
16    return_val = (parts[0]*(256*256*256)) + (parts[1]*(256*256)) + (parts[2]*256) + parseInt
      (parts[3]);
17    return return_val;
18  }

```

The **OnUpdate** routine specifies that when a change occurs to data within the bucket, the routine **get_numip_first_3_octets** is run on each document that contains **ip_start** and **ip_end**.

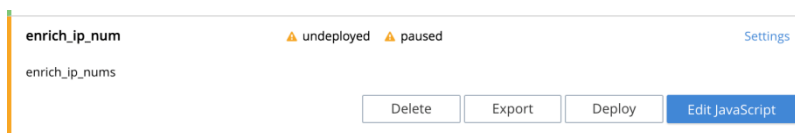
A new document is created whose data and metadata are based on those of the document on which **get_numip_first_3_octets** is run;

but with the addition of **ip_num_start** and **ip_num_end data-fields**, which contain the numeric values returned by **get_numip_first_3_octets**.

The **get_numip_first_3_octets** routine splits the IP address, converts each fragment to a numeral, and adds the numerals together, to form a single value; which it returns.

5. Click **Save**.

6. To return to the Eventing screen, click **Eventing** and click on the newly created Function name. The Function **enrich_ip_nums** is listed as a defined Function.





7. For feed boundary choose **Everything**
8. Click **Deploy Function**.

A screenshot of the 'Confirm Deploy Function' dialog box. The dialog has a blue header bar with the title 'Confirm Deploy Function' and a close button 'X'. Below the header, the text reads: 'Execute the new function for all invocations henceforth.' Underneath, it says 'Feed boundary:' followed by a dropdown menu. The dropdown menu is open, showing three options: 'Everything' (highlighted in blue), 'Everything', and 'From now'. To the right of the dropdown are two buttons: 'Cancel' and 'Deploy Function'.

9. From the **Confirm Deploy Function** dialog, click **Deploy Function**. From this point, the defined Function is executed on all existing documents and on subsequent mutations.
10. To check results of the deployed Function, click the **Documents** tab.
11. Select **target** bucket from the **Bucket** drop-down. As this shows, a version of **SampleDocument** has been added to the **target** bucket. It contains all the attributes of the original document, with the addition of **ip_num_start** and **ip_num_end**; which contain the numeric values that correspond to **ip_start** and **ip_end**, respectively.

Additional documents added to the **source** bucket, which share the **ip_start** and **ip_end** attributes, will be similarly handled by the defined Function: creating such a document, and changing any attribute in such a document both cause the Function's execution.



Analytics

This lab explores the pairing of Analytics and the beer sample bucket. You'll create a set of Analytics datasets based on the Couchbase beer-sample bucket and then explore a set of illustrative queries as a quick way to get familiar with the new Analytics user experience.

The complete set of steps to create and connect the sample datasets are included, along with a collection of runnable SQL++ queries and their expected results.

As you read through this document, you should try each step for yourself on your own Analytics instance.

You can use your favorite Analytics interface to do this:


the Analytics Workbench,

the cbq shell,

or the REST API.

Using the web UI

For the UI use the Analytics link to gain access to the web UI(you may need to follow an additional link if you are not already on an Analytics node.

 8 node cluster > Analytics

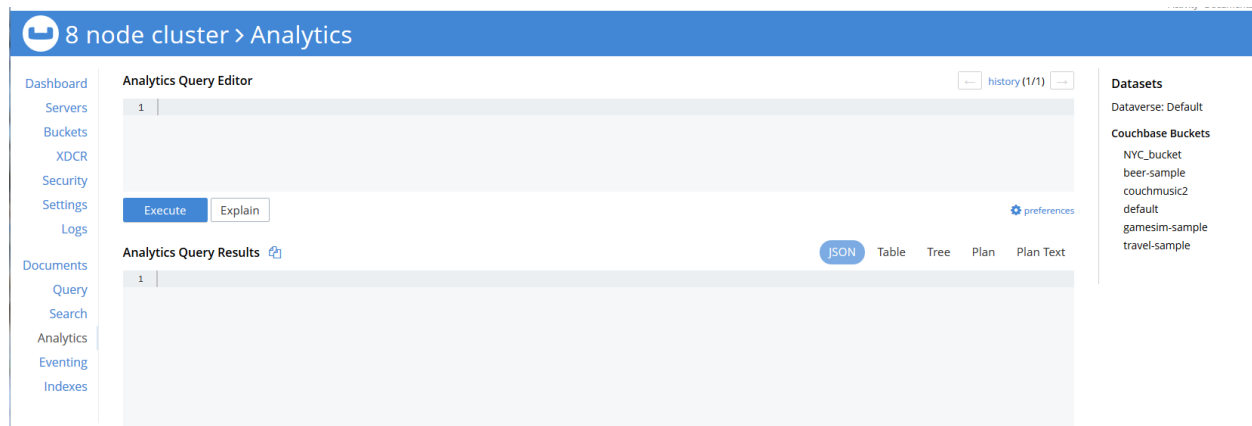
[Dashboard](#)
[Servers](#)
[Buckets](#)
[XDCR](#)
[Security](#)
[Settings](#)
[Logs](#)
[Documents](#)
[Query](#)
[Search](#)
[Analytics](#)
[Eventing](#)
[Indexes](#)

The analytics interface is only available on Couchbase nodes running the analytics service.

The analytics service was found on the following nodes:

<http://ec2-13-57-5-196.us-west-1.compute.amazonaws.com:8091/ui/index.html#/cbas/workbench>
<http://ec2-54-219-183-105.us-west-1.compute.amazonaws.com:8091/ui/index.html#/cbas/workbench>

⚠ Note that the above links are provided as a convenience. They may not work if you are accessing the Web Console via a web proxy or if Couchbase Server has been deployed in a split horizon DNS environment, with public and private hostnames and IPs. In the event the links don't work, log directly into the node.



You can verify that everything's working by issuing a simple SQL++ test query as shown below:

"Let there be beer!";

Note: You must press the Execute button in the Analytics Workbench. Unlike Query Workbench, pressing Enter or Return will not immediately execute a statement because Analytics accepts multi-line statements.

Using the Command Line Query Tool

You can use the command line tool, cbq, to run queries. cbq is the command line shell for executing queries against the Query service as well as the Analytics service in Couchbase.

Run cbq with the following options to specify the user, password, host, and port for the Analytics service:

```
$ /opt/couchbase/bin/cbq -u Administrator -p password
-e http://localhost:8095
```

NOTE: localhost is IP of one Analytics node

Enter the query in cbq:



```
cbq> SELECT "Hello, beer!" AS greeting;
{
  "requestID": "dfb78524-0a2b-4dd3-8b99-1554d8dc0bcd",
  "signature": "*",
  "results": [ {
    "greeting" : "Hello, beer!"
  } ]
,
  "status": "success",
  "metrics": {
    "elapsedTime": "21.947426ms",
    "executionTime": "18.042413ms",
    "resultCount": 1,
    "resultSize": 34,
    "processedObjects": 0
  }
}
```

Here you can see that the result shown in the Analytics Workbench is embedded in an envelope that contains additional information about the request like the requestID, the status, and some metrics.

Using the REST API

You can use the REST API to run Analytics queries. When the cluster is up, requests can be posted to Couchbase Analytics. Access <http://localhost:8095/analytics/service> that implements the same HTTP interface as the Couchbase Query service. For details, see Analytics REST API.

For example, you can run the query `SELECT "Hello, beer!" AS greeting` through curl:

```
$ curl -u Administrator:password -d 'statement=SELECT "Hello, beer!" AS greeting' http://localhost:8095/analytics/service
```

The request returns:

```
{ "requestID": "f1b8451c-f626-46be-b1f8-e9ffc891550b",
  "signature": "*", "results": [ { "greeting": "Hello, beer!" } ]
, "status": "success", "metrics": { "elapsedTime":
"28.292741ms", "executionTime": "24.82841ms", "resultCount": 1,
"resultSize": 31, "processedObjects": 0 } }
```



again including the envelope that we've seen in the example above. In addition to submitting queries to the Analytics Service you can also use the REST API for administrative purposes. E.g. you can retrieve version information for the Analytics Service with

```
$ curl -u Administrator:password  
http://localhost:8095/analytics/cluster
```

Organizing Data in Analytics

The top-level organizing concept in the Analytics data world is the dataverse. A dataverse, short for data universe, is a namespace that gives you a place to create and manage datasets and other artifacts for a given Analytics application. In that respect, a dataverse is similar to a database or a schema in a relational DBMS. To store your data in Analytics, you create a dataverse and then use it to hold the datasets for your own data. We'll get to this next. You get a Default dataverse for free, and Analytics will just use that if you don't specify another dataverse.

Datasets are containers that hold collections of JSON objects. They are similar to tables in an RDBMS or keyspaces in N1QL. In this tutorial, all of the datasets you create contain real-time synchronized copies of selected data from Couchbase Server. Analytics uses DCP to automatically maintain its own local representations for the JSON documents in the Couchbase Server data nodes.

A newly created Analytics instance starts out empty. That is, it contains no data other than the Analytics system catalogs. These system catalogs live in a special dataverse called the Metadata dataverse. If you want to know what dataverses have been defined so far, you can query the Dataverse dataset in the Metadata dataverse as shown below:

```
SELECT * FROM Metadata.`Dataverse`;
```

```
[
  {
    "Dataverse": {
      "DataverseName": "Default",
      "DataFormat":
"org.apache.asterix.runtime.formats.NonTaggedDataFormat",
      "Timestamp": "Fri Jul 07 17:22:27 PDT 2017",
      "PendingOp": 0
    }
  },
  {
    "Dataverse": {
      "DataverseName": "Metadata",
      "DataFormat":
"org.apache.asterix.runtime.formats.NonTaggedDataFormat",
      "Timestamp": "Fri Jul 07 17:22:27 PDT 2017",
      "PendingOp": 0
    }
  }
]
```



```
}
}
]
```

The output above shows that a fresh Analytics instance starts out with two dataverses, one called Metadata (the system catalog) and one called Default (available for holding data). This sample scenario deals with the information about beers and breweries, which you'll need to get from Couchbase Server. To keep things simple, we will use the Default dataverse here. That means your first task is to tell Analytics about the Couchbase Server data that you want to shadow and the datasets where you want it to live. The following Analytics DDL statements show you how to tell Analytics about the beer-sample bucket in Couchbase Server, where all of the beer and brewery information resides, and ask Analytics to shadow the data using two datasets, breweries and beers.

NOTE: use the backtic key(upper left with the tilde key) in the following commands. Additionally do not leave any space when specifying type(highlighted)

```
CREATE DATAVERSE `beer-data`;
```

```
CREATE DATASET
`beer-data`.beers ON `beer-sample` WHERE `type` = "beer";
```

```
CREATE DATASET
`beer-data`.breweries ON `beer-sample` WHERE `type` = "brewery";
```

The first statement (CREATE DATAVERSE) gives Analytics the information needed to access the data of interest from Couchbase Server. The next two statements (CREATE DATASET) create the target datasets in Analytics for the information of interest. Notice how WHERE clauses are utilized to direct beer-related data into separate, type-specific datasets for easier querying. Each of these datasets will be hash-partitioned (sharded) across all of the nodes running instances of the Analytics service. Hash partitioning sets the stage for the parallel processing that Analytics employs when processing analytical queries.

To actually initiate the shadowing relationship of these datasets to the data in Couchbase Server, one more step is needed, namely:

```
CONNECT LINK `beer-data`.Local;
```

Once you run this statement, Analytics begins making its copy of the Couchbase Server documents and continuously monitors it for changes.

Next, verify that your datasets are really there and being populated. The following SQL++ SELECT statements are one good way to accomplish that:

```
SELECT ds.BucketName, ds.DatasetName, ds.`Filter`
FROM Metadata.`Dataset` ds
WHERE ds.DataverseName = "beer-data";
```

The query looks in the Analytics system catalogs for datasets that have been created in the Default dataverse. At this point, assuming that you are just getting started, you will see only your two new datasets:



```
[ {
  "DatasetName" : "beers",
  "BucketName" : "beerBucket",
  "Filter" : "`type` = \"beer\""
}, {
  "DatasetName" : "breweries",
  "BucketName" : "beerBucket",
  "Filter" : "`type` = \"brewery\""
} ]
```

The following query asks Analytics the number of breweries:

```
SELECT VALUE COUNT(*) FROM `beer-data`.breweries;
```

It returns:

```
[
  1412
]
```

The next query retrieves a sample of breweries:

```
SELECT * FROM `beer-data`.breweries ORDER BY name LIMIT 1;
```

It returns:

```
[
  {
    "breweries": {
      "address": [
        "407 Radam, F200"
      ],
      "city": "Austin",
      "code": "78745",
      "country": "United States",
      "description": "(512) Brewing Company is a microbrewery located in the heart of Austin that brews for the community using as many local, domestic and organic ingredients as possible.",
      "geo": {
        "accuracy": "ROOFTOP",
        "lat": 30.2234,
        "lon": -97.7697
      },
      "name": "(512) Brewing Company",
      "phone": "512.707.2337",
      "state": "Texas",
      "type": "brewery",
    }
  ]
```



```

        "updated": "2010-07-22 20:00:20",
        "website": "http://512brewing.com/"
    }
}
]

```

The following query asks Analytics the number of beers:

```
SELECT VALUE COUNT(*) FROM `beer-data`.beers;
```

It returns:

```

[
  5891
]

```

The following query retrieves a sample of beers:

```
SELECT * FROM `beer-data`.beers limit 1;
```

It returns:

```

[
  {
    "beers": {
      "abv": 0,
      "brewery_id": "big_ridge_brewing",
      "category": "North American Lager",
      "description": "",
      "ibu": 0,
      "name": "#17 Cream Ale",
      "srm": 0,
      "style": "American-Style Lager",
      "type": "beer",
      "upc": 0,
      "updated": "2010-07-22 20:00:20"
    }
  }
]

```

SQL++: Querying Your Analytics Data

Congratulations! You now have your Couchbase Server beer-related data being shadowed in Analytics. You're ready to start running ad hoc queries against your breweries and beers datasets.

To do this, you'll program Analytics using the SQL++ query language, a SQL-inspired language designed for working with semistructured data. SQL++ has much in common with SQL, but there are differences due to the data model that SQL++ is designed to serve. SQL was designed in the 1970's to interact with



the flat, schema-ified world of relational databases. SQL++ is designed for the nested, schema-less (or schema-optional, in Analytics) world of NoSQL systems. SQL++ offers a mostly familiar paradigm for experienced SQL users to use to query and manipulate data in Analytics. SQL++ is also related to N1QL, the current query language used in Couchbase Server. SQL++ is really a functional superset of N1QL that is a bit closer to SQL, and the differences between N1QL and SQL++ will eventually disappear in the future releases.

In this section, we introduce SQL++ via a set of example queries with their expected results, based on the data above, to help you get started. Many of the most important features of SQL++ are presented in this set of representative queries.

For more information on the query language, see SQL++ Language Reference and the list of built-in functions in the Function Reference. As you will learn, SQL++ is a highly composable expression language. Even the very simple expression `1 + 1` is a valid SQL++ query that evaluates to 2. Try it for yourself!

It's worth noting that each time you execute a SQL++ query, the Analytics query engine employs state-of-the-art parallel processing algorithms similar to those used by the parallel relational DBMSs that power many enterprise data warehouses. Unlike those systems, Analytics also works on rich, flexible schema data.

Let's go ahead and write some queries and start learning SQL++ through examples.

Query 0 - Key-Based Lookup

For your first query, let's find a particular brewery based on its Couchbase Server key. You can do this for the Kona Brewing company as follows:

```
SELECT meta(bw) AS meta, bw AS data
FROM `beer-data`. breweries bw
WHERE meta(bw).id = 'kona_brewing';
```

As in SQL, the query's FROM clause binds the variable `bw` incrementally to the data instances residing in the dataset named `breweries`. Its WHERE clause selects only those bindings having the primary key of interest; the key is accessed (as in N1QL) by using the `meta` function to get to the meta-information about the objects. The SELECT clause returns all of the meta-information plus the data value (the selected brewery object in this case) for each binding that satisfies the predicate.



The expected result for this query is as follows:

```
[
  {
    "meta": {
      "id": "kona_brewing",
      "vbid": 862,
      "seq": 5,
      "cas": 1499958777027625000,
      "flags": 33554432
    },
    "data": {
      "address": [
        "75-5629 Kuakini Highway"
      ],
      "city": "Kailua-Kona",
      "code": "96740",
      "country": "United States",
      "description": "",
      "geo": {
        "accuracy": "RANGE_INTERPOLATED",
        "lat": 19.642,
        "lon": -155.996
      },
      "name": "Kona Brewing",
      "phone": "1-808-334-1133",
      "state": "Hawaii",
      "type": "brewery",
      "updated": "2010-07-22 20:00:20",
      "website": "http://www.konabrewingco.com"
    }
  }
]
```

Notice how the resulting object of interest has two fields whose names were requested in the SELECT clause.



Query 1 - Exact-Match Lookup

The SQL++ language, like SQL, supports a variety of different predicates. For the next query, let's find the same brewery information but in a slightly simpler or cleaner way based only on the data:

```
SELECT VALUE bw
FROM `beer-data`.breweries bw
WHERE bw.name = 'Kona Brewing';
```

This query's expected result is:

```
[
  {
    "address": [
      "75-5629 Kuakini Highway"
    ],
    "city": "Kailua-Kona",
    "code": "96740",
    "country": "United States",
    "description": "",
    "geo": {
      "accuracy": "RANGE_INTERPOLATED",
      "lat": 19.642,
      "lon": -155.996
    },
    "name": "Kona Brewing",
    "phone": "1-808-334-1133",
    "state": "Hawaii",
    "type": "brewery",
    "updated": "2010-07-22 20:00:20",
    "website": "http://www.konabrewingco.com"
  }
]
```

In SQL++, you can select a single value (whether it be an atomic or scalar value or a object value or an array value) by using a **SELECT VALUE** clause as shown above. If you instead use the more SQL-familier **SELECT** clause, SQL++ will return objects instead of values, and since a given query may **SELECT** multiple values in its result (like our first query did), you get a slightly differently shaped result using **SELECT bw** instead of **SELECT VALUE bw**:

```
[
  {
    "bw": {
      "address": [
        "75-5629 Kuakini Highway"
      ],
      "city": "Kailua-Kona",
      "code": "96740",
      "country": "United States",
```



```

    "description": "",
    "geo": {
      "accuracy": "RANGE_INTERPOLATED",
      "lat": 19.642,
      "lon": -155.996
    },
    "name": "Kona Brewing",
    "phone": "1-808-334-1133",
    "state": "Hawaii",
    "type": "brewery",
    "updated": "2010-07-22 20:00:20",
    "website": "http://www.konabrewingco.com"
  }
}
]

```

Query 2 - Other Query Filters

SQL++ can apply ranges and other conditions on any data type that supports the appropriate set of comparators. As an example, the next query applies a range condition together with a string condition to select breweries:

```

SELECT VALUE bw
FROM `beer-data`.breweries bw
WHERE bw.geo.lat > 60.0
AND bw.name LIKE '%Brewing%'
ORDER BY bw.name;

```

The expected result for this query is as follows:

```

[
  {
    "address": [
      "238 North Boundary Street"
    ],
    "city": "Wasilla",
    "code": "99654",
    "country": "United States",
    "description": "",
    "geo": {
      "accuracy": "RANGE_INTERPOLATED",
      "lat": 61.5816,
      "lon": -149.439
    },
    "name": "Great Bear Brewing",

```



```

    "phone": "1-907-373-4782",
    "state": "Alaska",
    "type": "brewery",
    "updated": "2010-07-22 20:00:20",
    "website": ""
  },
  {
    "address": [
      "8111 Dimond Hook Drive"
    ],
    "city": "Anchorage",
    "code": "99507",
    "country": "United States",
    "description": "Since firing up its brew kettle in 1995, Midnight
Sun Brewing Company has become a serious yet creative force on the
American brewing front. From concept to glass, Midnight Sun relies on
an art marries science approach, mixing tradition with innovation, to
design and craft bold, distinctive beers for Alaska...and beyond. We
at Midnight Sun find inspiration in the untamed spirit and rugged
beauty of the Last Frontier and develop unique beers with equally
appealing names and labels. But the company's true focus remains in
its dedication to producing consistently high-quality beers that
provide satisfying refreshment in all seasons... for Alaskans and
visitors alike. From our Pacific Northwest locale, we offer our
wonderful beers on draft throughout Alaska and in 22-ounce bottles
throughout Alaska and Oregon. We invite you to visit our hardworking,
little brewery in South Anchorage every chance you get!",
    "geo": {
      "accuracy": "ROOFTOP",
      "lat": 61.1473,
      "lon": -149.844
    },
    "name": "Midnight Sun Brewing Co.",
    "phone": "1-907-344-1179",
    "state": "Alaska",
    "type": "brewery",
    "updated": "2010-07-22 20:00:20",
    "website": "http://www.midnightsunbrewing.com/"
  },
  {
    "address": [],
    "city": "Anchorage",
    "code": "",
    "country": "United States",
    "description": "",
    "geo": {
      "accuracy": "APPROXIMATE",
      "lat": 61.2181,
      "lon": -149.9
    },
  },

```



```

    "name": "Railway Brewing",
    "phone": "",
    "state": "Alaska",
    "type": "brewery",
    "updated": "2010-07-22 20:00:20",
    "website": ""
  },
  {
    "address": [
      "2195 Old Steese Highway"
    ],
    "city": "Fox",
    "code": "99708",
    "country": "United States",
    "description": "Silver Gulch Brewing and Bottling Co. has been in operation since February 1998 in the small mining community of Fox, Alaska, located about 10 miles north of Fairbanks on the Steese Highway. Silver Gulch Brewing grew from brewmaster Glenn Brady's home-brewing efforts in 5-gallon batches to its current capacity of 24-barrel (750 gallon) batches.",
    "geo": {
      "accuracy": "ROOFTOP",
      "lat": 64.9583,
      "lon": -147.622
    },
    "name": "Silver Gulch Brewing Company",
    "phone": "(907) 452-2739",
    "state": "Alaska",
    "type": "brewery",
    "updated": "2010-07-22 20:00:20",
    "website": "http://www.ptialaska.net/~gbrady/"
  },
  {
    "address": [
      "717 W. 3rd Ave"
    ],
    "city": "Anchorage",
    "code": "99501",
    "country": "United States",
    "description": "",
    "geo": {
      "accuracy": "RANGE_INTERPOLATED",
      "lat": 61.2196,
      "lon": -149.896
    },
    "name": "Sleeping Lady Brewing Company",
    "phone": "(907) 277-7727",
    "state": "Alaska",
    "type": "brewery",
    "updated": "2010-07-22 20:00:20",

```




```

    "website": "http://www.alaskabeers.com/"
  }
]

```

Query 3 (and friends) - Equijoin

In addition to simply binding variables to data instances and returning them whole, a SQL++ query can construct new objects to return based on combinations of variable bindings. This gives SQL++ the power to do projections and joins much like those done using multi-table FROM clauses in SQL. For example, suppose that you wanted a list of all breweries paired with their associated beers, with the list enumerating the brewery name and the beer name for each such pair. You can do this as follows in SQL++, while also limiting the answer set size to at most 3 results:

```

SELECT bw.name AS brewer, br.name AS beer
FROM `beer-data`.breweries bw, `beer-data`.beers br
WHERE br.brewery_id = meta(bw).id
ORDER BY bw.name, br.name
LIMIT 3;

```

The result of this query is a sequence of new objects, one for each brewery/beer pair. Each instance in the result will be a object containing two fields, "brewer" and "beer", containing the brewery's name and the beer's name, respectively, for each brewery/beer pair. Notice how the use of a SQL-style SELECT clause, as opposed to the new SQL++ SELECT VALUE clause, automatically results in the construction of a new object value for each result.

The expected result of this example SQL++ join query is:

```

[
  {
    "brewer": "(512) Brewing Company",
    "beer": "(512) ALT"
  },
  {
    "brewer": "(512) Brewing Company",
    "beer": "(512) Bruin"
  },
  {
    "brewer": "(512) Brewing Company",
    "beer": "(512) IPA"
  }
]

```



If we were feeling lazy, e.g., while browsing our data casually, we might use `SELECT *` in SQL++ to return all of the matching beer/brewery data:

```
SELECT *
FROM `beer-data`.breweries bw, `beer-data`.beers br
WHERE br.brewery_id = meta(bw).id
ORDER BY bw.name, br.name
LIMIT 3;
```

In SQL++, this `SELECT *` query will produce a new nested object for each brewery/beer pair. Each result object contains one field (named after the "breweries" variable) to hold the brewery object and another field (named after the "beers" variable) to hold the matching beer object. Note that the nested nature of this SQL++ `SELECT *` result is different than traditional SQL, as SQL was not designed to handle the richer, nested data model that underlies the design of SQL++.

The expected result of this version of the SQL++ join query for our sample data set is as follows:

```
[
  {
    "bw": {
      "address": [
        "407 Radam, F200"
      ],
      "city": "Austin",
      "code": "78745",
      "country": "United States",
      "description": "(512) Brewing Company is a microbrewery located
in the heart of Austin that brews for the community using as many
local, domestic and organic ingredients as possible.",
      "geo": {
        "accuracy": "ROOFTOP",
        "lat": 30.2234,
        "lon": -97.7697
      },
      "name": "(512) Brewing Company",
      "phone": "512.707.2337",
      "state": "Texas",
      "type": "brewery",
      "updated": "2010-07-22 20:00:20",
      "website": "http://512brewing.com/"
    },
    "br": {
      "abv": 6,
      "brewery_id": "512_brewing_company",
      "category": "German Ale",
      "description": "(512) ALT is a German-style amber ale that is
fermented cooler than typical ales and cold conditioned like a lager.
ALT means "old" in German and refers to a beer style made using ale
yeast after many German brewers had switched to newly discovered lager
```



yeast. This ale has a very smooth, yet pronounced, hop bitterness with a malty backbone and a characteristic German yeast character. Made with 98% Organic 2-row and Munch malts and US noble hops.",

```

    "ibu": 0,
    "name": "(512) ALT",
    "srm": 0,
    "style": "German-Style Brown Ale/Altbier",
    "type": "beer",
    "upc": 0,
    "updated": "2010-07-22 20:00:20"
  }
},
{
  "bw": {
    "address": [
      "407 Radam, F200"
    ],
    "city": "Austin",
    "code": "78745",
    "country": "United States",
    "description": "(512) Brewing Company is a microbrewery located
in the heart of Austin that brews for the community using as many
local, domestic and organic ingredients as possible.",
    "geo": {
      "accuracy": "ROOFTOP",
      "lat": 30.2234,
      "lon": -97.7697
    },
    "name": "(512) Brewing Company",
    "phone": "512.707.2337",
    "state": "Texas",
    "type": "brewery",
    "updated": "2010-07-22 20:00:20",
    "website": "http://512brewing.com/"
  },
  "br": {
    "abv": 7.6,
    "brewery_id": "512_brewing_company",
    "category": "North American Ale",
    "description": "At once cuddly and ferocious, (512) BRUIN
combines a smooth, rich maltiness and mahogany color with a solid hop
backbone and stealthy 7.6% alcohol. Made with Organic 2 Row and Munich
malts, plus Chocolate and Crystal malts, domestic hops, and a touch of
molasses, this brew has notes of raisins, dark sugars, and cocoa, and
pairs perfectly with food and the crisp fall air.",
    "ibu": 0,
    "name": "(512) Bruin",
    "srm": 0,
    "style": "American-Style Brown Ale",
    "type": "beer",

```



```

    "upc": 0,
    "updated": "2010-07-22 20:00:20"
  },
  {
    "bw": {
      "address": [
        "407 Radam, F200"
      ],
      "city": "Austin",
      "code": "78745",
      "country": "United States",
      "description": "(512) Brewing Company is a microbrewery located
in the heart of Austin that brews for the community using as many
local, domestic and organic ingredients as possible.",
      "geo": {
        "accuracy": "ROOFTOP",
        "lat": 30.2234,
        "lon": -97.7697
      },
      "name": "(512) Brewing Company",
      "phone": "512.707.2337",
      "state": "Texas",
      "type": "brewery",
      "updated": "2010-07-22 20:00:20",
      "website": "http://512brewing.com/"
    },
    "br": {
      "abv": 7,
      "brewery_id": "512_brewing_company",
      "category": "North American Ale",
      "description": "(512) India Pale Ale is a big, aggressively dry-
hopped American IPA with smooth bitterness (~65 IBU) balanced by
medium maltiness. Organic 2-row malted barley, loads of hops, and
great Austin water create an ale with apricot and vanilla aromatics
that lure you in for more.",
      "ibu": 0,
      "name": "(512) IPA",
      "srm": 0,
      "style": "American-Style India Pale Ale",
      "type": "beer",
      "upc": 0,
      "updated": "2010-07-22 20:00:20"
    }
  }
]

```

Die-hard SQL JOIN clause syntax fans will be happy to know that SQL++ hasn't forgotten them, so a result identical to the one immediately above can also be produced as follows:



```
SELECT *
FROM `beer-data`.breweries bw JOIN `beer-data`.beers br ON
br.brewery_id = meta(bw).id
ORDER BY bw.name, br.name
LIMIT 3;
```

Finally (for now :-)), another more explicit SQL++ way of achieving the very same result as above is:

```
SELECT VALUE {"bw": bw, "br": br}
FROM `beer-data`.breweries bw, `beer-data`.beers br
WHERE br.brewery_id = meta(bw).id
ORDER BY bw.name, br.name
LIMIT 3;
```

This version of the query uses an explicit object constructor to build each result object. Note that the string field names "bw" and "br" in the object constructor above are both simple SQL++ expressions themselves, so in the most general case, even the resulting field names can be computed as part of the query, making SQL++ a very powerful tool for slicing and dicing semistructured data.

(It is worth knowing, with respect to influencing Analytics' query evaluation, that FROM and JOIN clauses - also known as joins - are currently evaluated in order, with the left clause probing the data of the right clause.)

Query 4 - Nested Outer Join

In order to support joins between tables with missing or dangling join tuples, the designers of SQL ended up shoe-horning a subset of the relational algebra into SQL's FROM clause syntax and providing a variety of join types there for users to choose from (which SQL++ supports for SQL compatibility). Left outer joins are particularly important in SQL, for example, to print a summary of customers and orders, grouped by customer, without omitting those customers who haven't placed any orders yet.

The SQL++ language supports nesting, both of queries and of query results, and the combination allows for a cleaner and more natural approach to such queries. As an example, suppose you wanted for each brewery to produce an object that contains the brewery name along with a list of all of the brewery's offered beer names and alcohol percentages. In the flat (also known as 1NF) world of SQL, approximating this query would involve a left outer join between breweries and beers, ordered by brewery, with the brewery name being repeated along side each beer's information. In the richer (NoSQL) world of SQL++, this use case can be handled more naturally as follows:

```
SELECT bw.name AS brewer,
(SELECT br.name, br.abv FROM `beer-data`.beers br
WHERE br.brewery_id = meta(bw).id
ORDER BY br.name) AS beers
FROM `beer-data`.breweries bw
ORDER BY bw.name
LIMIT 2;
```

This SQL++ query binds the variable bw to the objects in breweries; for each brewery, it constructs a result object containing a "brewer" field with the brewery's name plus a "beers" field with a nested



collection of objects containing the beer name and alcohol percentage for each of the brewery's beers. The nested collection field for each brewery is created using a correlated subquery.

Note: While it looks like nested loops could be involved in computing the result, Analytics recognizes the equivalence of such a query to an outerjoin, so it will use an efficient parallel join strategy when actually computing the query's result.

Below is this example query's expected output:

```
[
  {
    "beers": [
      {
        "name": "(512) Bruin",
        "abv": 7.6
      },
      {
        "name": "(512) Pecan Porter",
        "abv": 6.8
      },
      {
        "name": "(512) Whiskey Barrel Aged Double Pecan Porter",
        "abv": 8.2
      },
      {
        "name": "(512) ALT",
        "abv": 6
      },
      {
        "name": "(512) IPA",
        "abv": 7
      },
      {
        "name": "(512) Pale",
        "abv": 5.8
      },
      {
        "name": "(512) Wit",
        "abv": 5.2
      },
      {
        "name": "One",
        "abv": 8
      }
    ],
    "brewer": "(512) Brewing Company"
  },
  {
    "beers": [
```



```

    {
      "name": "21A IPA",
      "abv": 7.2
    },
    {
      "name": "Amendment Pale Ale",
      "abv": 5.2
    },
    {
      "name": "Bitter American",
      "abv": 3.6
    },
    {
      "name": "General Pippo's Porter",
      "abv": 5.5
    },
    {
      "name": "North Star Red",
      "abv": 5.8
    },
    {
      "name": "Oyster Point Oyster Stout",
      "abv": 5.9
    },
    {
      "name": "South Park Blonde",
      "abv": 5
    },
    {
      "name": "563 Stout",
      "abv": 5
    },
    {
      "name": "Double Trouble IPA",
      "abv": 9.8
    },
    {
      "name": "Potrero ESB",
      "abv": 5.2
    },
    {
      "name": "Watermelon Wheat",
      "abv": 5.5
    }
  ],
  "brewer": "21st Amendment Brewery Cafe"
}
]

```



Query 5 - Theta Join

Not all joins are expressible as equijoins and computable using equijoin-oriented algorithms. The join predicates for some use cases involve predicates with functions; Analytics supports the expression of such queries and will still evaluate them as best it can using nested loop based techniques (and broadcast joins in the parallel case).

As an example of such a use case, suppose that you wanted for each Arizona brewery to get the brewery's name, location, and a list of competitors' names – where competitors are other breweries that are geographically close to their location. In SQL++, this can be accomplished in a manner similar to the previous query, but with locality plus name inequality instead of a simple key equality condition in the correlated query's WHERE clause:

```
SELECT bw1.name AS brewer, bw1.geo AS location,
(SELECT VALUE bw2.name FROM `beer-data`.breweries bw2
WHERE bw2.name != bw1.name
AND abs(bw1.geo.lat - bw2.geo.lat) <= 0.1
AND abs(bw2.geo.lon - bw1.geo.lon) <= 0.1
) AS competitors
FROM `beer-data`.breweries bw1
WHERE bw1.state = 'Arizona'
ORDER BY bw1.name
LIMIT 3;
```

Here is the expected result for this query:

```
[
  {
    "competitors": [
      "Mudshark Brewing"
    ],
    "brewer": "Barley Brothers Brewery and Grill",
    "location": {
      "accuracy": "RANGE_INTERPOLATED",
      "lat": 34.4702,
      "lon": -114.35
    }
  },
  {
    "competitors": [
      "Mogollon Brewing Company"
    ],
    "brewer": "Flagstaff Brewing",
    "location": {
      "accuracy": "ROOFTOP",
      "lat": 35.1973,
```




```

        "lon": -111.648
    }
},
{
    "competitors": [
        "Rio Salado Brewing"
    ],
    "brewer": "Four Peaks Brewing",
    "location": {
        "accuracy": "ROOFTOP",
        "lat": 33.4194,
        "lon": -111.916
    }
}
]

```

Query 6 - Existential Quantification

The expressive power of SQL++ includes support for queries involving some (existentially quantified) and all (universally quantified) query semantics. Quantified predicates are especially useful for querying datasets involving nested collections of objects, in order to find objects where some or all of their nested sets' objects satisfy a condition of interest. To illustrate their use in such situations, we start here by using another (orthogonal) feature of SQL++, its WITH clause, to create a temporarily nested view of breweries and their beers. We then use an existential (SOME) predicate to find those breweries whose beers include at least one IPA and return the brewery's name, phone number, and complete list of beer names and associated alcohol levels. Here is the resulting SQL++ query:

```

WITH nested_breweries AS
(
  SELECT bw.name AS brewer, bw.phone,
  (
    SELECT br.name, br.abv FROM `beer-data`.beers br
    WHERE br.brewery_id = meta(bw).id
    ORDER BY br.name
  ) AS beers
  FROM `beer-data`.breweries bw
)
SELECT VALUE nb FROM nested_breweries nb
WHERE (SOME b IN nb.beers SATISFIES b.name LIKE '%IPA%')
ORDER BY nb.brewer
LIMIT 2;

```

The expected result in this case is:

```

[
  {

```



```

"beers": [
  {
    "name": "(512) ALT",
    "abv": 6
  },
  {
    "name": "(512) Bruin",
    "abv": 7.6
  },
  {
    "name": "(512) IPA",
    "abv": 7
  },
  {
    "name": "(512) Pale",
    "abv": 5.8
  },
  {
    "name": "(512) Pecan Porter",
    "abv": 6.8
  },
  {
    "name": "(512) Whiskey Barrel Aged Double Pecan Porter",
    "abv": 8.2
  },
  {
    "name": "(512) Wit",
    "abv": 5.2
  },
  {
    "name": "One",
    "abv": 8
  }
],
"brewer": "(512) Brewing Company",
"phone": "512.707.2337"
},
{
  "beers": [
    {
      "name": "21A IPA",
      "abv": 7.2
    },
    {
      "name": "563 Stout",
      "abv": 5
    },
    {
      "name": "Amendment Pale Ale",
      "abv": 5.2
    }
  ]
}

```



```

    },
    {
      "name": "Bitter American",
      "abv": 3.6
    },
    {
      "name": "Double Trouble IPA",
      "abv": 9.8
    },
    {
      "name": "General Pippo's Porter",
      "abv": 5.5
    },
    {
      "name": "North Star Red",
      "abv": 5.8
    },
    {
      "name": "Oyster Point Oyster Stout",
      "abv": 5.9
    },
    {
      "name": "Potrero ESB",
      "abv": 5.2
    },
    {
      "name": "South Park Blonde",
      "abv": 5
    },
    {
      "name": "Watermelon Wheat",
      "abv": 5.5
    }
  ],
  "brewer": "21st Amendment Brewery Cafe",
  "phone": "1-415-369-0900"
}
]

```



Query 7 - Universal Quantification

As an example of a universally quantified SQL++ query, we can find those breweries that only have seriously strong beers:

```
WITH nested_breweries AS
(
  SELECT bw.name AS brewer, bw.phone,
  (
    SELECT br.name, br.abv FROM `beer-data`.beers br
    WHERE br.brewery_id = meta(bw).id
    ORDER BY br.name) AS beers
  FROM `beer-data`.breweries bw
)
SELECT VALUE nb FROM nested_breweries nb
WHERE (EVERY b IN nb.beers SATISFIES b.abv >= 10)
AND ARRAY_COUNT(nb.beers) > 0
ORDER BY nb.brewer
LIMIT 5;
```

Notice how the brewery predicate also makes sure that the set of beers for each qualifying brewery is non-empty; this is needed if we want our results to exclude breweries that currently offer no beers, as an empty set trivially satisfies a universal predicate. The expected result in this case is:

```
[
  {
    "beers": [
      {
        "name": "Podge Belgian Imperial Stout",
        "abv": 10.5
      }
    ],
    "brewer": "Alvinne Picobrouwerij",
    "phone": "32-051-/-30-55-17"
  },
  {
    "beers": [
      {
        "name": "Belzebuth",
        "abv": 13
      }
    ],
    "brewer": "Brasserie Grain D'Orge",
    "phone": ""
  },
  {
    "beers": [
      {
        "name": "Malheur 10",
        "abv": 10
      }
    ]
  }
]
```



```

    },
    {
      "name": "Malheur 12",
      "abv": 12
    },
    {
      "name": "Malheur Black Chocolate 2003",
      "abv": 12
    },
    {
      "name": "Malheur Brut Reserve",
      "abv": 11
    },
    {
      "name": "Malheur MM",
      "abv": 10
    }
  ],
  "brewer": "Brouwerij De Landtsheer",
  "phone": "32-052-33-39-11"
},
{
  "beers": [
    {
      "name": "Maredsous 10 Tripple",
      "abv": 10
    }
  ],
  "brewer": "Brouwerij Duvel Moortgat",
  "phone": ""
},
{
  "beers": [
    {
      "name": "Sexual Chocolate",
      "abv": 10
    }
  ],
  "brewer": "Foothills Brewing Company",
  "phone": "(336) 777-3348"
}
]

```



Query 8 - Simple Aggregation

Like SQL, the SQL++ language of Analytics provides support for computing aggregates over large amounts of data. As a very simple example, the following SQL++ query computes the total number of beers in a SQL-like way:

```
SELECT COUNT(*) AS num_beers FROM `beer-data`.beers;
```

This query's result will be:

```
[ {  
  "num_beers" : 5891  
} ]
```

If an unwrapped value is preferred, the following variant could be used instead:

```
SELECT VALUE COUNT(b) FROM `beer-data`.beers b;
```

This time the result will simply be:

```
[  
  5891  
]
```

In SQL++, aggregate functions can be applied to arbitrary collections, including subquery results. To illustrate, here is a less SQL-like, and also more explicit, way to express the query above:

```
SELECT VALUE ARRAY_COUNT((SELECT b FROM `beer-data`.beers b));
```

For each traditional SQL aggregate function *F*, SQL++ has a corresponding function *ARRAY_F* that can be used to perform the desired aggregate calculation. Each such function is a regular function that takes a collection-valued argument to aggregate over. Thus, the query above counts the results produced by the beer selection subquery, and the previous, more SQL-like versions are just syntactic sugar for SQL++ queries that logically use *ARRAY_COUNT*.



Query 9 (and friends) - Grouping and Aggregation

Also like SQL, SQL++ supports grouped aggregation. For each brewery that offers more than 30 beers, the following group-by or aggregate query reports the number of beers that it offers.

```
SELECT br.brewery_id, COUNT(*) AS num_beers
FROM `beer-data`.beers br
GROUP BY br.brewery_id
HAVING COUNT(*) > 30
ORDER BY COUNT(*) DESC;
```

The FROM clause incrementally binds the variable `br` to beers, and the GROUP BY clause groups the beers by their associated brewery id. Unlike SQL, where data is tabular (flat), the data model underlying SQL++ allows for nesting. Thus, due to the GROUP BY clause, the SELECT clause in this query sees a sequence of `br` groups, with each such group having an associated `brewery_id` variable value (that is the producing brewery's id). In the context of the SELECT clause, `brewery_id` is bound to the brewery's id and `br` is now re-bound (due to grouping) to the set of beers issued by that brewery. The SELECT clause yields a result object containing the brewery's id and the count of the items in the associated beer set. The query result will contain one such object per brewery id.

Below is the expected result for this query over the sample data:

```
[
  {
    "num_beers": 57,
    "brewery_id": "midnight_sun_brewing_co"
  },
  {
    "num_beers": 49,
    "brewery_id": "rogue_ales"
  },
  {
    "num_beers": 38,
    "brewery_id": "anheuser_busch"
  },
  {
    "num_beers": 37,
    "brewery_id": "egan_brewing"
  },
  {
    "num_beers": 37,
    "brewery_id": "troegs_brewing"
  }
]
```



```

    },
    {
      "num_beers": 36,
      "brewery_id": "boston_beer_company"
    },
    {
      "num_beers": 34,
      "brewery_id": "f_x_matt_brewing"
    },
    {
      "num_beers": 34,
      "brewery_id": "titledtown_brewing"
    },
    {
      "num_beers": 33,
      "brewery_id": "sierra_nevada_brewing_co"
    },
    {
      "num_beers": 32,
      "brewery_id": "stone_brewing_co"
    },
    {
      "num_beers": 31,
      "brewery_id": "southern_tier_brewing_co"
    }
  ]

```

Analytics has multiple evaluation strategies available for processing grouped aggregate queries. For grouped aggregation, the system knows how to employ both sort-based and hash-based parallel aggregation methods, with sort-based methods being used by default and a hint being available to suggest that a different approach (hashing) be used in processing a particular SQL++ query.

The following query is nearly identical to the previous one, but adds a hash-based aggregation hint (though that's not necessarily the more efficient approach here):

```

SELECT br.brewery_id, COUNT(*) AS num_beers
FROM `beer-data`.beers br
/*+ hash */
GROUP BY br.brewery_id
HAVING COUNT(*) > 30
ORDER BY COUNT(*) DESC;

```

Here is the expected result (the same result, but in a slightly different order):

```

[
  {
    "num_beers": 57,
    "brewery_id": "midnight_sun_brewing_co"
  },
  {

```




```
    "num_beers": 49,
    "brewery_id": "rogue_ales"
  },
  {
    "num_beers": 38,
    "brewery_id": "anheuser_busch"
  },
  {
    "num_beers": 37,
    "brewery_id": "troegs_brewing"
  },
  {
    "num_beers": 37,
    "brewery_id": "egan_brewing"
  },
  {
    "num_beers": 36,
    "brewery_id": "boston_beer_company"
  },
  {
    "num_beers": 34,
    "brewery_id": "f_x_matt_brewing"
  },
  {
    "num_beers": 34,
    "brewery_id": "titledtown_brewing"
  },
  {
    "num_beers": 33,
    "brewery_id": "sierra_nevada_brewing_co"
  },
  {
    "num_beers": 32,
    "brewery_id": "stone_brewing_co"
  },
  {
    "num_beers": 31,
    "brewery_id": "southern_tier_brewing_co"
  }
]
```



Query 10 - Grouping and Limits

In some use cases it is not necessary to compute the entire answer to a query. In some cases, just having the first N or top N results are sufficient. This is expressible in SQL++ using the LIMIT clause combined with the ORDER BY clause. (You may have noticed that we have used the LIMIT clause all along to keep the result set sizes in this document manageable.)

The following SQL++ query returns the top three breweries based on their numbers of offered beers. It also illustrates the use of multiple aggregate functions to compute various alcohol content statistics for their beers:

```
SELECT bw.name,  
COUNT(*) AS num_beers,  
AVG(br.abv) AS abv_avg,  
MIN(br.abv) AS abv_min,  
MAX(br.abv) AS abv_max  
FROM `beer-data`.breweries bw, `beer-data`.beers br  
WHERE br.brewery_id = meta(bw).id  
GROUP BY bw.name  
ORDER BY num_beers DESC  
LIMIT 3;
```

The expected result for this query is:

```
[  
  {  
    "num_beers": 57,  
    "abv_avg": 7.7675438596491215,  
    "name": "Midnight Sun Brewing Co.",  
    "abv_min": 0,  
    "abv_max": 16  
  },  
  {  
    "num_beers": 49,  
    "abv_avg": 4.688775510204083,  
    "name": "Rogue Ales",  
    "abv_min": 0,  
    "abv_max": 11.5  
  },  
  {  
    "num_beers": 38,  
    "abv_avg": 3.8052631578947373,
```



```

    "name": "Anheuser-Busch",
    "abv_min": 0,
    "abv_max": 8
  }
]

```

Everything Must Change

So far you have been walking through the SQL++ query capabilities of Analytics. What really makes Analytics interesting, however, is that it brings this query power to bear on your nearly-current Couchbase Server data, enabling you to harness the power of parallelism in Analytics to analyze what's going on with your data "up front" in essentially real time, without perturbing your Couchbase Server applications' performance (or the resulting end user experience). Before closing this tutorial, let's take a very quick look at that aspect of Analytics.

To start, the following SQL++ query lists all of the Kona Brewery's current beer offerings:

```

SELECT meta(b).id, b as beer FROM `beer-data`.beers b
WHERE b.brewery_id = "kona_brewing"
ORDER BY meta(b).id;

```

The result of this query will be a list of the following seven beers:

```

[
  {
    "id": "kona_brewing-black_sand_porter",
    "beer": {
      "abv": 0,
      "brewery_id": "kona_brewing",
      "category": "Irish Ale",
      "description": "",
      "ibu": 0,
      "name": "Black Sand Porter",
      "srm": 0,
      "style": "Porter",
      "type": "beer",
      "upc": 0,
      "updated": "2010-07-22 20:00:20"
    }
  },
  {
    "id": "kona_brewing-fire_rock_pale_ale",
    "beer": {
      "abv": 5.8,
      "brewery_id": "kona_brewing",
      "category": "North American Ale",
      "description": "",
      "ibu": 0,

```



```

    "name": "Fire Rock Pale Ale",
    "srm": 0,
    "style": "American-Style Pale Ale",
    "type": "beer",
    "upc": 0,
    "updated": "2010-07-22 20:00:20"
  }
},
{
  "id": "kona_brewing-lilikoi_wheat_ale",
  "beer": {
    "abv": 0,
    "brewery_id": "kona_brewing",
    "category": "Other Style",
    "description": "",
    "ibu": 0,
    "name": "Lilikoi Wheat Ale",
    "srm": 0,
    "style": "Light American Wheat Ale or Lager",
    "type": "beer",
    "upc": 0,
    "updated": "2010-07-22 20:00:20"
  }
},
{
  "id": "kona_brewing-longboard_lager",
  "beer": {
    "abv": 5.5,
    "brewery_id": "kona_brewing",
    "category": "North American Lager",
    "description": "",
    "ibu": 0,
    "name": "Longboard Lager",
    "srm": 0,
    "style": "American-Style Lager",
    "type": "beer",
    "upc": 0,
    "updated": "2010-07-22 20:00:20"
  }
},
{
  "id": "kona_brewing-pipeline_porter",
  "beer": {
    "abv": 0,
    "brewery_id": "kona_brewing",
    "category": "Irish Ale",
    "description": "Pipeline Porter is smooth and dark with a
distinctive roasty aroma and earthy complexity from its diverse blends
of premium malted barley. This celebration of malt unites with freshly
roasted 100% Kona coffee grown at Cornwell Estate on Hawaii's Big

```



Island, lending a unique roasted aroma and flavor. A delicate blend of hops rounds out this palate-pleasing brew.",

```

    "ibu": 0,
    "name": "Pipeline Porter",
    "srm": 0,
    "style": "Porter",
    "type": "beer",
    "upc": 0,
    "updated": "2010-07-22 20:00:20"
  }
},
{
  "id": "kona_brewing-stout",
  "beer": {
    "abv": 0,
    "brewery_id": "kona_brewing",
    "category": "North American Ale",
    "description": "",
    "ibu": 0,
    "name": "Stout",
    "srm": 0,
    "style": "American-Style Stout",
    "type": "beer",
    "upc": 0,
    "updated": "2010-07-22 20:00:20"
  }
},
{
  "id": "kona_brewing-wailua",
  "beer": {
    "abv": 0,
    "brewery_id": "kona_brewing",
    "category": "Other Style",
    "description": "Wailua is Hawaiian for two fresh water streams
mingling. This was just the inspiration we needed for our Limited
Release wheat ale brewed with tropical passion Fruit. A refreshing
citrusy, sun-colored ale with the cool taste of Hawaii.",
    "ibu": 0,
    "name": "Wailua",
    "srm": 0,
    "style": "Light American Wheat Ale or Lager",
    "type": "beer",
    "upc": 0,
    "updated": "2010-07-22 20:00:20"
  }
}
]

```



To illustrate Analytics in action, suppose that Kona's marketing department determines that a light beer is needed. You can use your favorite Couchbase Server interface to modify the server's beer-sample content accordingly, for example, **a N1QL insert query**:

```
INSERT INTO `beer-sample` ( KEY, VALUE )
VALUES
(
  "kona_brewing-skimboard_light_ale",
  {"name": "Skimboard Light Ale", "abv": 4.0, "ibu": 0.0, "srm":
0.0, "upc": 0, "type": "beer", "brewery_id": "kona_brewing",
"updated": "2010-07-22 20:00:20", "description": "", "style": "Light
Beer", "category": "North American Ale"}
)
RETURNING META().id as docid, *;
```

Analytics will shadow this change, updating the beers dataset as a result. Go ahead and rerun the Analytics SQL++ query that lists the Kona Brewery's beer offerings:

```
SELECT meta(b).id, b as beer FROM `beer-data`.beers b
WHERE b.brewery_id = "kona_brewing"
ORDER BY meta(b).id;
```

The result of the query now reflects the new beer offering:

```
[
  {
    "id": "kona_brewing-black_sand_porter",
    "beer": {
      "abv": 0,
      "brewery_id": "kona_brewing",
      "category": "Irish Ale",
      "description": "",
      "ibu": 0,
      "name": "Black Sand Porter",
      "srm": 0,
      "style": "Porter",
      "type": "beer",
      "upc": 0,
      "updated": "2010-07-22 20:00:20"
    }
  },
  {
    "id": "kona_brewing-fire_rock_pale_ale",
    "beer": {
      "abv": 5.8,
      "brewery_id": "kona_brewing",
```



```

    "category": "North American Ale",
    "description": "",
    "ibu": 0,
    "name": "Fire Rock Pale Ale",
    "srm": 0,
    "style": "American-Style Pale Ale",
    "type": "beer",
    "upc": 0,
    "updated": "2010-07-22 20:00:20"
  }
},
{
  "id": "kona_brewing-lilikoi_wheat_ale",
  "beer": {
    "abv": 0,
    "brewery_id": "kona_brewing",
    "category": "Other Style",
    "description": "",
    "ibu": 0,
    "name": "Lilikoi Wheat Ale",
    "srm": 0,
    "style": "Light American Wheat Ale or Lager",
    "type": "beer",
    "upc": 0,
    "updated": "2010-07-22 20:00:20"
  }
},
{
  "id": "kona_brewing-longboard_lager",
  "beer": {
    "abv": 5.5,
    "brewery_id": "kona_brewing",
    "category": "North American Lager",
    "description": "",
    "ibu": 0,
    "name": "Longboard Lager",
    "srm": 0,
    "style": "American-Style Lager",
    "type": "beer",
    "upc": 0,
    "updated": "2010-07-22 20:00:20"
  }
},
{
  "id": "kona_brewing-pipeline_porter",
  "beer": {
    "abv": 0,
    "brewery_id": "kona_brewing",
    "category": "Irish Ale",

```



"description": "Pipeline Porter is smooth and dark with a distinctive roasty aroma and earthy complexity from its diverse blends of premium malted barley. This celebration of malt unites with freshly roasted 100% Kona coffee grown at Cornwell Estate on Hawaii's Big Island, lending a unique roasted aroma and flavor. A delicate blend of hops rounds out this palate-pleasing brew.",

```

    "ibu": 0,
    "name": "Pipeline Porter",
    "srm": 0,
    "style": "Porter",
    "type": "beer",
    "upc": 0,
    "updated": "2010-07-22 20:00:20"
  }
},
{
  "id": "kona_brewing-skimboard_light_ale",
  "beer": {
    "abv": 4,
    "brewery_id": "kona_brewing",
    "category": "North American Ale",
    "description": "",
    "ibu": 0,
    "name": "Skimboard Light Ale",
    "srm": 0,
    "style": "Light Beer",
    "type": "beer",
    "upc": 0,
    "updated": "2010-07-22 20:00:20"
  }
},
{
  "id": "kona_brewing-stout",
  "beer": {
    "abv": 0,
    "brewery_id": "kona_brewing",
    "category": "North American Ale",
    "description": "",
    "ibu": 0,
    "name": "Stout",
    "srm": 0,
    "style": "American-Style Stout",
    "type": "beer",
    "upc": 0,
    "updated": "2010-07-22 20:00:20"
  }
},
{
  "id": "kona_brewing-wailua",
  "beer": {

```




```

    "abv": 0,
    "brewery_id": "kona_brewing",
    "category": "Other Style",
    "description": "Wailua is Hawaiian for two fresh water streams
mingling. This was just the inspiration we needed for our Limited
Release wheat ale brewed with tropical passion Fruit. A refreshing
citrusy, sun-colored ale with the cool taste of Hawaii.",
    "ibu": 0,
    "name": "Wailua",
    "srm": 0,
    "style": "Light American Wheat Ale or Lager",
    "type": "beer",
    "upc": 0,
    "updated": "2010-07-22 20:00:20"
  }
}
]

```

To further illustrate Analytics in action, suppose that Kona's CEO determines that a light beer is in fact NOT needed. You can use your favorite [N1QL interface again](#) to modify the Couchbase Server's beer-sample content accordingly, that is:

```
DELETE FROM `beer-sample` b USE KEYS "kona_brewing-
skimboard_light_ale";
```

Finally, if you run the SQL++ Kona beer list query on Analytics once again, you will find that the Kona CEO's wishes have been shadowed in Analytics as well.

```
SELECT meta(b).id, b as beer FROM `beer-data`.beers b
WHERE b.brewery_id = "kona_brewing"
ORDER BY meta(b).id;
```

And of Course, Indexing



Last but not least, if you're a database fan who has come this far, you may be wondering about indexing. Indeed, in addition to the use of efficient parallel execution strategies for large analytical queries, Couchbase Analytics provides support for the use of indexes to speed the execution of smaller queries. For example, consider the following query:

```
SELECT VALUE br
FROM `beer-data`.beers br
WHERE br.brewery_id = 'kona_brewing';
```

The following DDL statements could be used to accelerate its execution for a large collection of beers:

```
DISCONNECT LINK `beer-data`.Local;

CREATE INDEX brewerIndex ON `beer-data`.beers (brewery_id: STRING);

CONNECT LINK `beer-data`.Local;
```

Note that index creation requires temporary suspension of the shadowing process, although queries on the accumulated data can continue to be run while shadowing is on hold and the index is being built. Note also that a CREATE INDEX statement needs to specify the type as well as the name of the field to be indexed; the above brewerIndex will be used to speed queries that have string-based brewery_id predicates.

Further Help

For more information, see SQL++ Language Reference, or consult the complete list of built-in functions in the Function Reference.

Couchbase Analytics lets you bring a powerful new NoSQL parallel query engine to bear on your data, using the latest state of the art parallel query processing techniques under the hood.

We hope you find it useful in exploring and analyzing your Couchbase Server data - without having to worry about end user performance impact or doing ETL grunt work to make your analyses possible.

(Note that the DP demo set up does not effectively demonstrate the performance aspect of Analytics.)



Do the following:

The last thing to do to prepare for the upcoming labs is to remove nodes 7 and 8 from your cluster.

Nodes 7 and 8 will be used to construct and new 2 node cluster for the XDCR labs.

If you do not remember how to remove a node gracefully from your cluster consult lab 3 or 4 for the step by step procedures

After you remove the Eventing/Analytics nodes from your cluster, rename your original cluster back to "6 Node NYC-Cluster".

End of Lab.