QUERYING WITH SQL++

Couchbase

This cheatsheet uses the dataset found in the online tutorial at http://query-tutorial.couchbase.com.

BASICS

SELECT Statement

```
SELECT count(*), state
FROM customer
WHERE customer.ccInfo.cardType="discover"
GROUP BY customer.state
ORDER BY customer.state
LIMIT 5 OFFSET 5
```

This query counts the number of customers per state who have a Discover credit card. The result set is grouped and ordered by state. Output is limited to 5 documents, after skipping the first 5.

Simple Arithmetic

Normalized, rounded, and truncated ratings:

```
SELECT AVG(reviews.rating) / 5
AS normalizedRating,
ROUND((AVG(reviews.rating) / 5), 2)
AS roundedRating,
TRUNC((AVG(reviews.rating) / 5), 3)
AS truncRating
FROM reviews AS reviews
WHERE reviews.customerId = "customer62"
```

With SQL++, you can use the +, -, * and / operators. SQL++ has functions for rounding (ROUND) and truncation (TRUNC). See the result set for this query below:

```
"results": [
  {
    "normalizedRating": 0.65,
    "roundedRating": 0.65,
    "truncRating": 0.65
  }
]
```

Try using other aggregation functions like SUM, MIN, and MAX.

String Concatenation and Matching

The || operator concatenates the first and last names to form the full name. The LIKE operator filters customers with email addresses ending in .biz.

```
SELECT firstName || " " || lastName AS fullName
FROM customer
WHERE emailAddress LIKE "%.biz"
```

```
"results": [
  {
    "fullName": "Joyce Murazik"
  }, // ...
]
```

SQL++ also provides string functions such as LOWER, UPPER, SUBSTR, and LENGTH.

DISTINCT

The DISTINCT keyword enables you to remove duplicate results.

To count the number of unique customers who have purchased something:

```
SELECT COUNT( DISTINCT customerId )
FROM purchases
```

NULL and MISSING Values

JSON documents can contain NULL values or omit fields entirely. The NULL/MISSING operators let you test for these conditions.

```
SELECT fname, children
FROM tutorial
WHERE children IS NULL
```

Now, try changing IS NULL to IS MISSING.

Indexes

SQL++ uses indexes to perform queries. You can create primary indexes and global secondary indexes.

```
CREATE INDEX idx ON `customer`(`emailAddress`)
```

EXPLAIN

EXPLAIN shows how a statement will operate.

```
EXPLAIN <Query Statement>
```

DATA STRUCTURES

Arrays and Objects

SQL++ supports nested JSON arrays and objects. You can use the dot "." operator to access fields inside nested objects, and the bracket [index] to access elements inside an array.

For example, consider the following object:

```
{ "address" : { "city": "Toronto"}, "revision":
[2014] }
```

`address.city` will return "Toronto" and `revision[0]` will return 2014.

These are some of the additional array functions:

ARRAY_LENGTH(<array>)
ARRAY_PREPEND(<value>,<array>)
ARRAY_APPEND(<array>,<value>)
ARRAY_CONCAT(<array1>,<array2>)

Range Predicates

SQL++ provides operators to work with arrays of nested objects. Range predicates allow you to test a boolean condition over the elements of an array.

The ANY operator allows you to search through an array, returning TRUE when at least one match is found. With the EVERY operator, every single element needs to match.

To search for purchase orders with a particular item purchased 5 times or more:

```
SELECT *
FROM purchases
WHERE ANY item IN purchases.lineItems SATISFIES
item.count >= 5 END
```

Try changing ANY to EVERY.

Range Transformations

To map and filter elements of an array, you can use the ARRAY and FIRST operators.

To get an array of products for each purchase order:

```
SELECT ARRAY item.product
FOR item IN purchases.lineItems END
AS product_ids
FROM purchases
```

Changing ARRAY to FIRST will produce the first product in each purchase order.

JOINS

JOIN, NEST, and UNNEST

A JOIN in SQL++ is similar to SQL; a single result is produced for each matching left and right-hand input.

NEST produces a single result for each left-hand input, while the right-hand input is collected and nested into a single array-valued field in the result.

To assemble a complete list of products purchased by a customer:

```
SELECT c, pr
FROM purchases pu
JOIN customer c ON KEYS pu.customerId
NEST product pr ON KEYS ARRAY li.product FOR li
IN pu.lineItems END
WHERE pu.customerId = "customer1"
```

The UNNEST clause allows you to take contents of a nested array and join them with the parent object.

To list products belonging to a particular category:

```
SELECT p
FROM product p
UNNEST p.categories AS category
WHERE category= "Appliances"
```