N1QL: A PRACTICAL GUIDE

N1QL = SQL + JSON

Gerald Sangudi
Marco Greco
Isha Kandaswamy
Eben Haber
Keshav Murthy
Sitaram Vemulapalli
Johan Larson
Prasad Varakur
Manuel Hurtado
Tai Tran

Foreword by Gerald Sangudi, father of N1QL

To Couchbase & N1QL
Customers & Community

Foreword

We are excited to present this eBook to you, our N1QL customers, users and readers. We started developing N1QL way back in 2013, when we realized that people were managing more of their data in JSON, but in doing so, they were having to give up the benefits of SQL querying. We appreciate both SQL and JSON, and we thought users should have the benefits of both.

After a few developer previews and much helpful feedback from early users and customers, N1QL was released in 2015. As of this writing in March 2017, N1QL is being used by over 100 enterprise customers, and many times that number of community users and developers. N1QL has steadily grown in sophistication, functionality, and performance.

This book is a compilation of some of our DZone articles through 2016. We have found DZone to be a great medium for N1QL blogs and articles. In addition to the initial reading, the articles serve as a continuous resource for questions, learning, reference, and discussion. A special thanks to our Couchbase and community colleagues, including those who contributed articles after 2016.

We hope you enjoy this eBook. If you have feedback or suggestions for future topics, please share them on our forums, on Stack Overflow, and of course, on DZone.

Gerald Sangudi
For Team N1QL
Mountain View, California
March, 14th, 2017

Table of Contents

Enterprise Jobs To Be DONE

Enterprises on Getting the Jobs Done

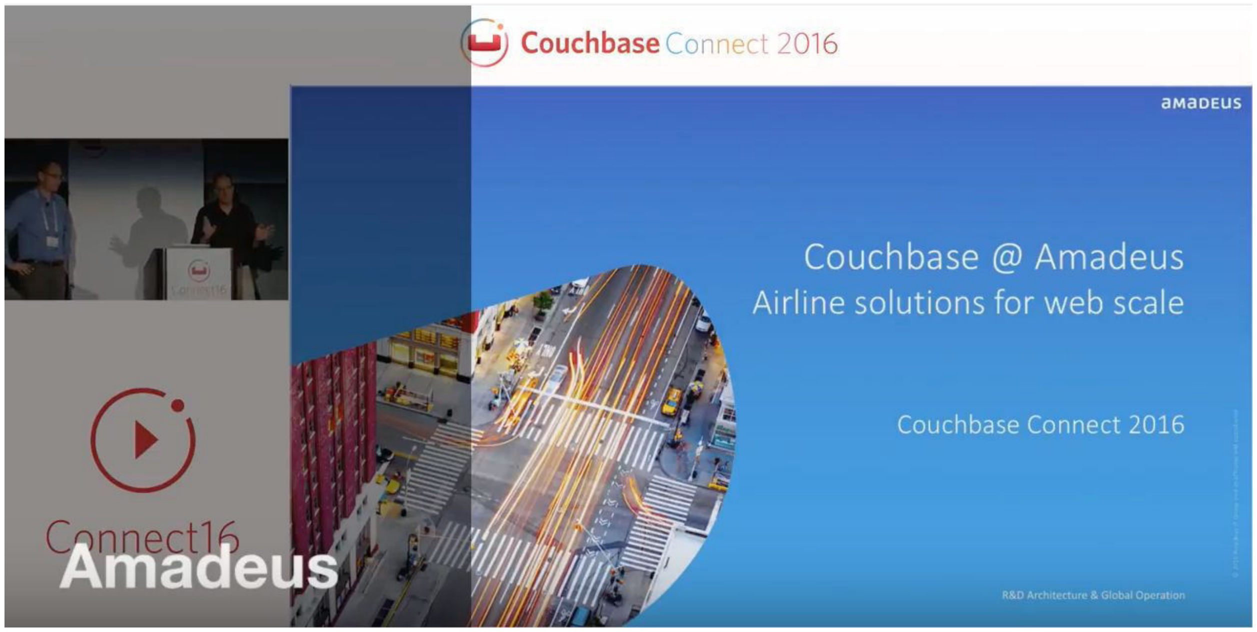
Authors: Gerald Sangudi, Keshav Murthy

Every enterprise, from the biggest and most familiar brand to the up and coming innovator, has important jobs to get done. Cisco is a networking company and its jobs include delivering VOIP solutions, video infrastructure, and more. Marriott is a hotel company and its jobs include customer management, reservation management, inventory and revenue management, and more.

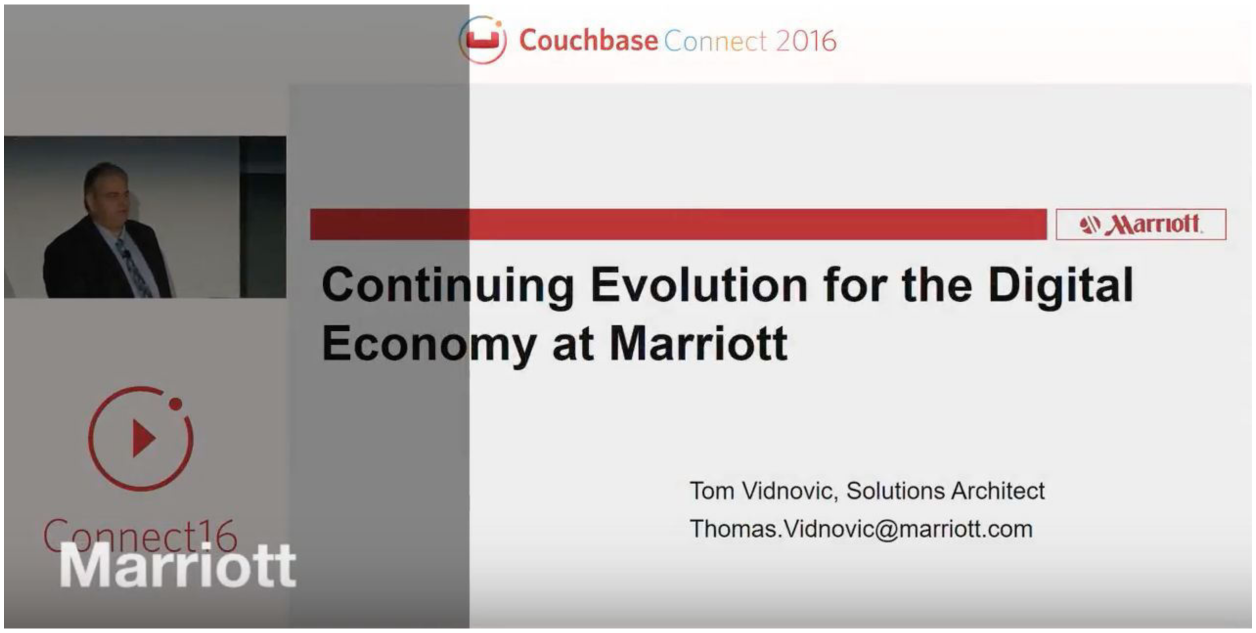
At the 2016 Couchbase Connect conference, many of these enterprises came and spoke about their jobs and how they are getting those jobs done. The Couchbase channel has a full 45-minute video of each talk. The typical talk introduces the speaker, the customer, the jobs (use cases), and how the customer is using Couchbase and other technologies to get those jobs done.

In the clips below, we have distilled each video to highlight the introductions and the customer's jobs, challenges, and learnings. Each video is about 10 minutes long, with three customer speakers in their own words. We enjoyed the videos, and we hope you will too. Here is the YouTube playlist to listen to all the videos, and here are the individual videos.

Video 1: Amadeus, Paypal, United Airlines



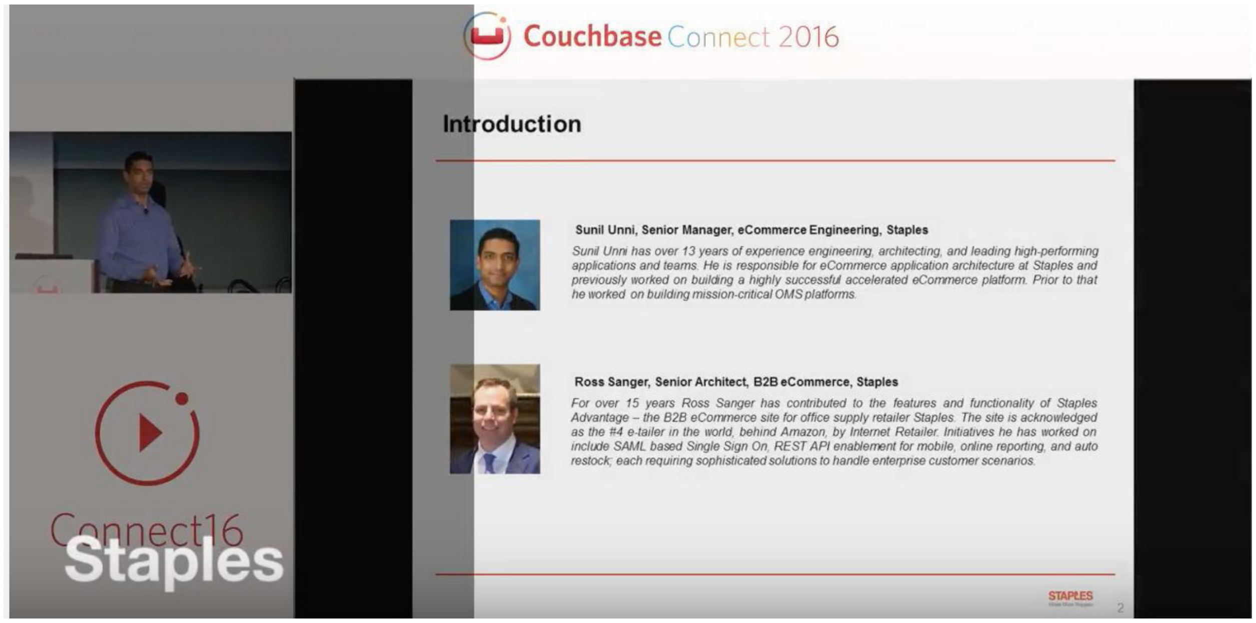
Video 2: Marriott, Sky, cars.com



Continuing Evolution for the Digital Economy at Marriott

Tom Vidnovic, Solutions Architect
Thomas.Vidnovic@marriott.com

Video 3: [Staples, Verizon, Doddle]



Introduction

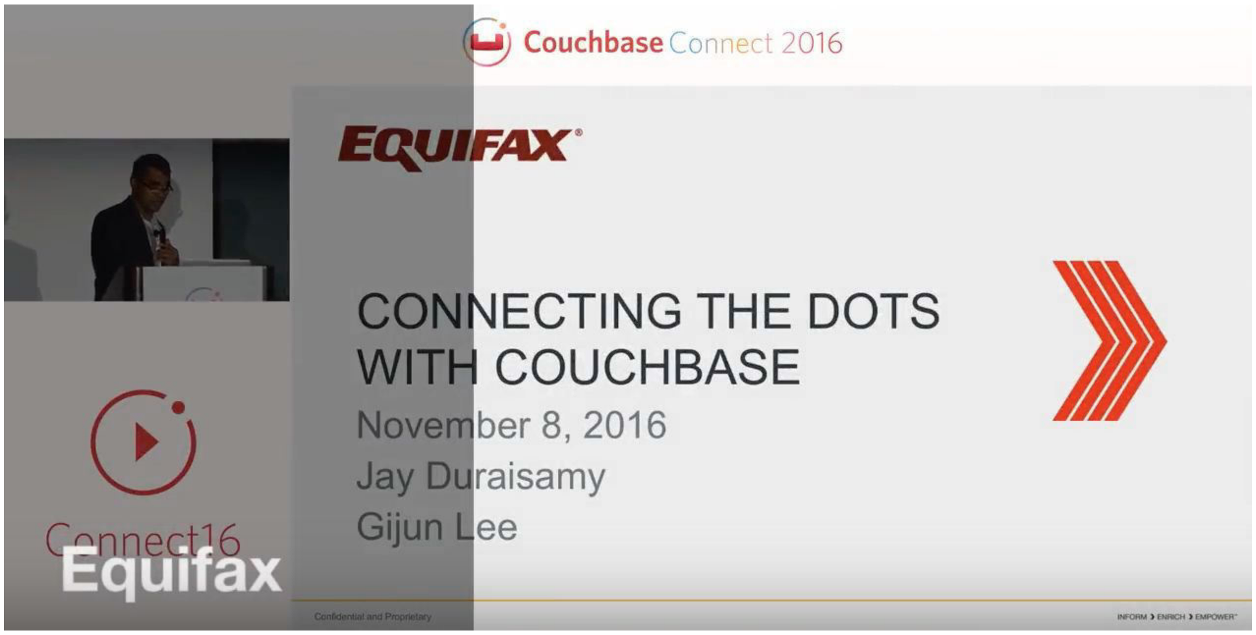
Sunil Unni, Senior Manager, eCommerce Engineering, Staples

Sunil Unni has over 13 years of experience engineering, architecting, and leading high-performing applications and teams. He is responsible for eCommerce application architecture at Staples and previously worked on building a highly successful accelerated eCommerce platform. Prior to that he worked on building mission-critical OMS platforms.

Ross Sanger, Senior Architect, B2B eCommerce, Staples

For over 15 years Ross Sanger has contributed to the features and functionality of Staples Advantage – the B2B eCommerce site for office supply retailer Staples. The site is acknowledged as the #4 e-tailer in the world, behind Amazon, by Internet Retailer. Initiatives he has worked on include SAML based Single Sign On, REST API enablement for mobile, online reporting, and auto restock; each requiring sophisticated solutions to handle enterprise customer scenarios.

Video 4: [Equifax, eBay, CenterEdge Software]


Couchbase Connect 2016

EQUIFAX

CONNECTING THE DOTS
WITH COUCHBASE

November 8, 2016

Jay Duraisamy

Gijun Lee

Confidential and Proprietary

INFORM › ENRICH › EMPOWER

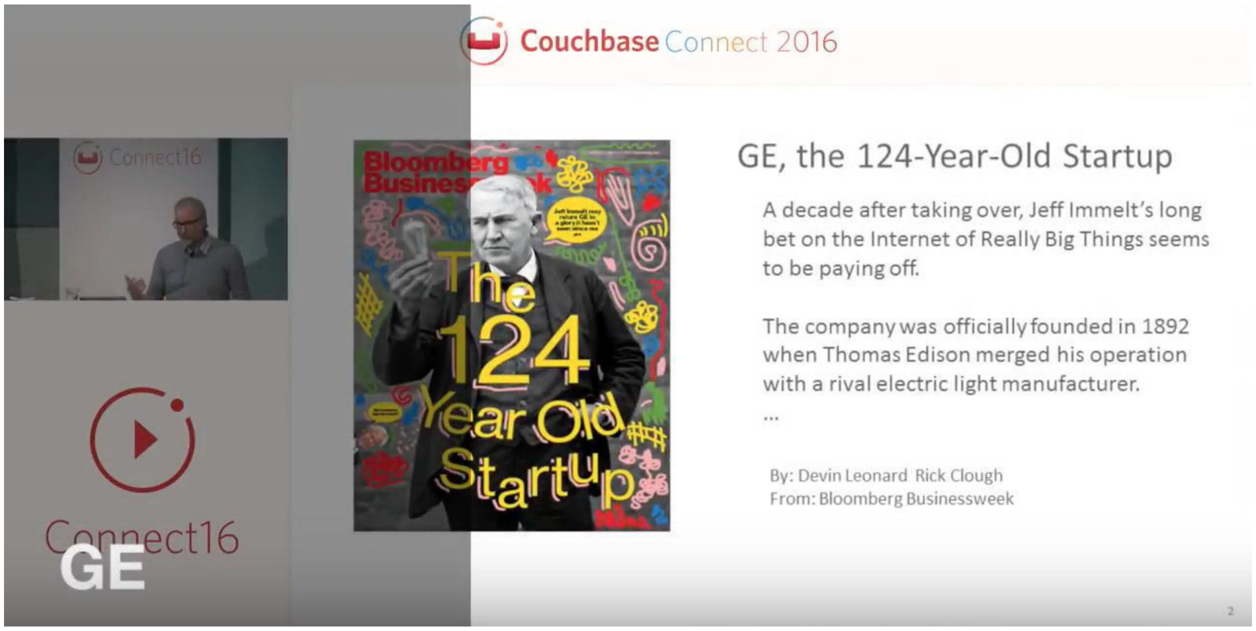
Video 5: LinkedIn, Cisco, Cvent

Couchbase Connect 2016

Couchbase Application
Clustering

Clint Ricker

Video 6: GE, SHOP.COM, Seenit



GE, the 124-Year-Old Startup

A decade after taking over, Jeff Immelt's long bet on the Internet of Really Big Things seems to be paying off.

The company was officially founded in 1892 when Thomas Edison merged his operation with a rival electric light manufacturer.
...

By: Devin Leonard Rick Clough
From: Bloomberg Businessweek

Query on JSON

Keep Calm and JSON

Author: Keshav Murthy

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for software to parse and generate. Here’s a JSON representation of a customer data in JSON. This has served well for interchange and integration.

```
{
   "Name" : "Jane Smith",
   "DOB"  : "1990-01-30",
   "Billing" : [
       {
           "type"    : "visa",
           "cardnum" : "5827-2842-2847-3909",
           "expiry"  : "2019-03"
       },
       {
           "type"    : "master",
           "cardnum" : "6274-2542-5847-3949",
           "expiry"  : "2018-12"
       }
   ],
  "addressz":
     {
           "street"  : "9462, Fillmore",
           "city"    : "San Francisco",
           "state"   : "California",
           "zip"     : 94401
     }
}
```

So far, so good.

As long as JSON was used for data interchange between multiple layers of application, everything was good. The moment people started using JSON as the database storage format, all hell broke loose. When I first saw JSON as the data format in database, I was surprised databases would pay the cost of keeping both key names and values for every document and process them for every query. I've heard this question numerous times in my talks and articles.

Here are the common objections:

1. JSON is text. It's inefficient.
2. JSON has no enforceable structure. Data quality is gone.
3. Key-value pairs for every document (row)? You're nuts.

Let's discuss these objections one by one.

1. JSON is TEXT. It's Inefficient.

Online transactions provided by RDBMS was key to life as we know it even with the advent of Web 2.0, RDBMS was running the all the websites in the back. The cost of transaction had come down from $5 when mankind went to the moon down to $0.02 by the time Jim Gray took his last sailing trip. This is because of tremendous effort on RDBMS and hardware vendors to continuously improve to win TPC benchmark wars in the '90s. Oracle would publish on SUN hardware. Sybase would beat the numbers on HP. Informix would beat it on SGI. IBM would beat it on IBM. Then, Oracle will come back. Everyone would spend $25,000 to boast in a full page ad on WSJ — yes, people read newspapers on paper in the '90s.

For this effort, databases would optimize the physical design, access paths, I/O, data loading, transactions, et cetera. This drove many innovations in hardware such as, microprocessor instruction sets to SMP system design, InfiniBand.

Databases squeezed every clock cycle, every IO, took every locking, logging advantage. Every bottleneck was attacked vigorously. Every hardware vendor tried to have the best TPC number.

RDBMS is efficient.

RDBMS is cost conscious — space, time, spacetime, everything.

Compared to RDBMS storage formats, JSON is inefficient.

- All the data is in character form of JSON requires conversion processing.
- Even numerical data is stored in text.
- Each value comes with its key name (key-value) in each document. More stored & processing.
- Allows complex structure to be represented resulting in a runtime overhead.
- Officially supports small set of data types. No decimal, no timestamp, etc.

Having JSON in a database means:

- Database engine has to understand each document's schema and process them instead of handling regular tuples and attributes (rows & columns).
- The data types for each key-value in JSON has to be interpreted and processed
- JSON may be a skinnier version of XML, but it's still fat.

With all these inefficiencies, why should any database use JSON as the data model? Or implement a JSON data type?

Yes. JSON was invented as data interchange format and now is the preferred format for public APIs, data integration scenarios. Most devices, most layers of software can consume and product data in the form of JSON. Storing and publishing data in JSON means, APIs and applications can interact with the database in the medium they're familiar with, in addition to the advantages below. You've to think in term of n-tier application efficiency instead of comparing it to tuple insert and update. Most RDBMS have added JSON as a data type now. As the JSON usage grows, the storage and processing efficiency will increase.

2. JSON has no enforceable structure in the database. Data quality is gone.

Since the JSON databases do not enforce any structure. Application, intentionally or unintentionally, can insert data in any format as long as it's a valid JSON.

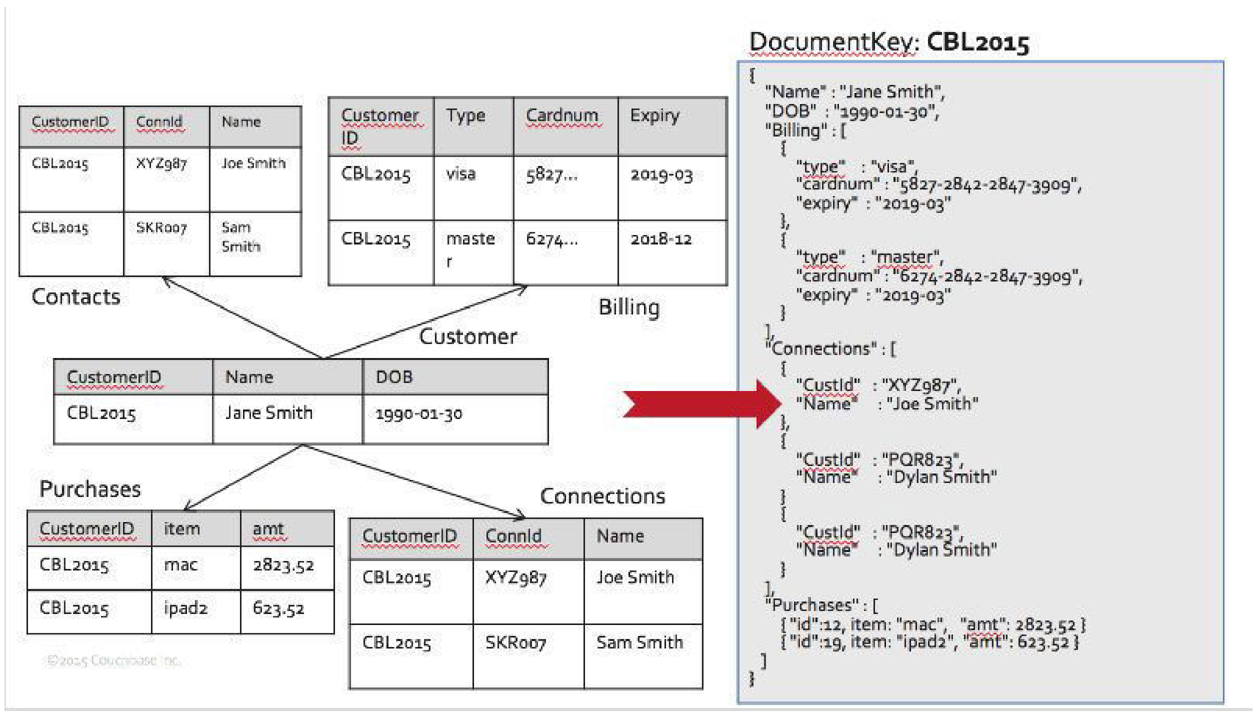
JSON based databases prefer a "schema on read" model. This means, reader has responsibility to understand and interpret the schema of each document or result from the query. Query language like N1QL has many feature for introspection of the document itself.

Having said that, there are tools like Ottoman and Mongoose help you to validate a given JSON against a predefined structure.While this isn't a fool proof method for data loading into JSON database, right controls in the application layer help you enforce consistency.

On the other hand, the flexible, nested structure of JSON helps you map object data into a single JSON document.

Using JSON also avoids the overhead of object-relational mapping. Applications model and modify objects. To store these in relational schema, you need to normalize the data into multiple relations and store them in discrete tables. When you need to reconstruct the object, you need to join data in these tables, and construct the object back. While object-relational mapping layers like Hibernate automate this process, you still pay in performance because you issue multiple queries for each step.

By representing the object in hierarchical nested structure of JSON, you avoid most of the overhead of object-relational mapping.



Contacts

CustomerID	ConnId	Name
CBL2015	XYZ987	Joe Smith
CBL2015	SKR007	Sam Smith

Billing

Customer ID	Type	Cardnum	Expiry
CBL2015	visa	5827...	2019-03
CBL2015	master	6274...	2018-12

Customer

CustomerID	Name	DOB
CBL2015	Jane Smith	1990-01-30

Purchases

CustomerID	item	amt
CBL2015	mac	2823.52
CBL2015	ipad2	623.52

Connections

CustomerID	ConnId	Name
CBL2015	XYZ987	Joe Smith
CBL2015	SKR007	Sam Smith

©2015 Couchbase Inc.

DocumentKey: **CBL2015**

```
{
  "Name" : "Jane Smith",
  "DOB"  : "1990-01-30",
  "Billing" : [
    {
      "type"    : "visa",
      "cardnum" : "5827-2842-2847-3909",
      "expiry"  : "2019-03"
    },
    {
      "type"    : "master",
      "cardnum" : "6274-2842-2847-3909",
      "expiry"  : "2019-03"
    }
  ],
  "Connections" : [
    {
      "CustId"  : "XYZ987",
      "Name"    : "Joe Smith"
    },
    {
      "CustId"  : "PQR823",
      "Name"    : "Dylan Smith"
    }
    {
      "CustId"  : "PQR823",
      "Name"    : "Dylan Smith"
    }
  ],
  "Purchases" : [
    { "id":12, item: "mac",   "amt": 2823.52 }
    { "id":19, item: "ipad2", "amt": 623.52 }
  ]
}
```

3. Key-value pairs for every document? You're nuts!

The answer is, the bottleneck has moved. Bottleneck has moved from database core performance to database flexibility and change management for iterative application development.

In the '80s and '90s, as RDBMS was used for many more applications and use cases, it was important for RDBMS to keep up with the scale and all the hardware innovations. Higher clock speed, new instructions, more cores, larger memory, all were exploited. As business grew, demand for more automation, applications, processes increased. As the apps got more popular and valuable, businesses want to do more with this. Business applications went from back office automation to real business change drivers.

RDBMS is great at many things. But, not CHANGE themselves. Issues are both organizational and technical.

RDBMS use a "schema on write" approach. Schema is determined at CREATE TABLE time, checked for at every INSERT, UPDATE, MERGE. Changing the table structure typically involves multiple stakeholders, multiple applications, testing all the dependent modules.

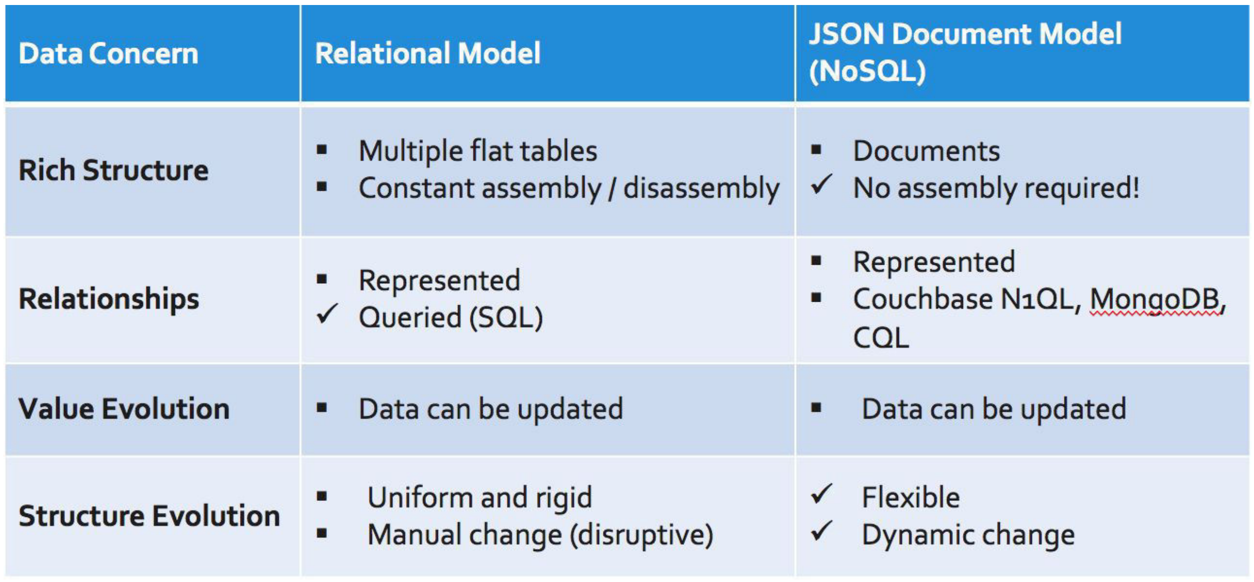
Typical ALTER table to add columns, change data types requires exclusive lock on the table resulting in application and business scheduled downtime. If you need a column to store distinct types of data or structure, you're out of luck. this trend, there have been attempts alleviate small subset of the issues by allowing users to add columns ONLINE.

All these takes time, slowing the pace of development and delivery. You can see further requirements of schema evolution on JSON hands it in this presentation [starting slide 17].

Using self-describing JSON model via {"key":value} pairs to store the data makes each document self-describing. So, you can avoid enforcing the schema on write. When the documents is read by the query processing engine or the application, they'll have to read and interpret the document. Applications typically version the documents so they know how what to expect as the schema of the document itself evolves. Query languages like N1QL, SQL++ have extended query languages to introspect and process flexible schema.

Summary

Here's the comparison of modeling data in relational model vs. JSON.



Data Concern	Relational Model	JSON Document Model (NoSQL)
Rich Structure	▪ Multiple flat tables ▪ Constant assembly / disassembly	▪ Documents ✓ No assembly required!
Relationships	▪ Represented ✓ Queried (SQL)	▪ Represented ▪ Couchbase N1QL, MongoDB, CQL
Value Evolution	▪ Data can be updated	▪ Data can be updated
Structure Evolution	▪ Uniform and rigid ▪ Manual change (disruptive)	✓ Flexible ✓ Dynamic change

Yes. JSON is text, makes schema enforcement challenging and takes more space. But, you get easier application development, schema evolution, and object-relational mapping.

References

- Keep Calm and Query JSON: https://dzone.com/articles/keep-calm-and-query-json
- N1QL: http://query.couchbase.com
- Couchbase: http://www.couchbase.com
- Couchbase Documentation: http://developer.couchbase.com

KEEP CALM and QUERY JSON

Author: Keshav Murthy

You've seen why and how to model data in JSON. Once you have the data in JSON, you can store and retrieve the JSON documents in any simple key-value databases. If your application only has to set and get the documents based on the key, use any key-value store. you're all set.

That's seldom the case.

You don't use a database just to store and retrieve JSON. You're trying to solve a real business problem. You're developing a business application. For a business application, the questions, operations, and reports don't change just because your database doesn't support query.

Business applications have to do many things requiring multi-object operation:

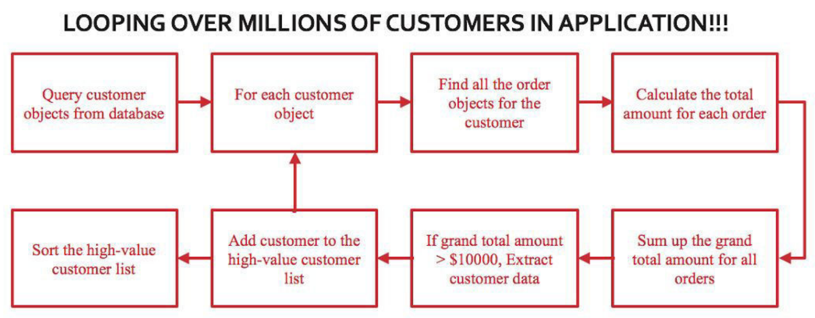
- Process Order Checkout.
- Search stores data for the shoe customer is looking for.
- How many new customers did we get last month?
- Generate the outstanding list of shipments due for today.
- Retrieve the customer order using case insensitive customer name.
- Load the new inventory data.
- Merge customer lists.

For doing each of these tasks, you need to search your database efficiently. Do the select-join-group-project-aggregate-order processing of the data to produce a report. Similarly, you'll have to insert, update, delete, and merge to represent the real-world business processes.

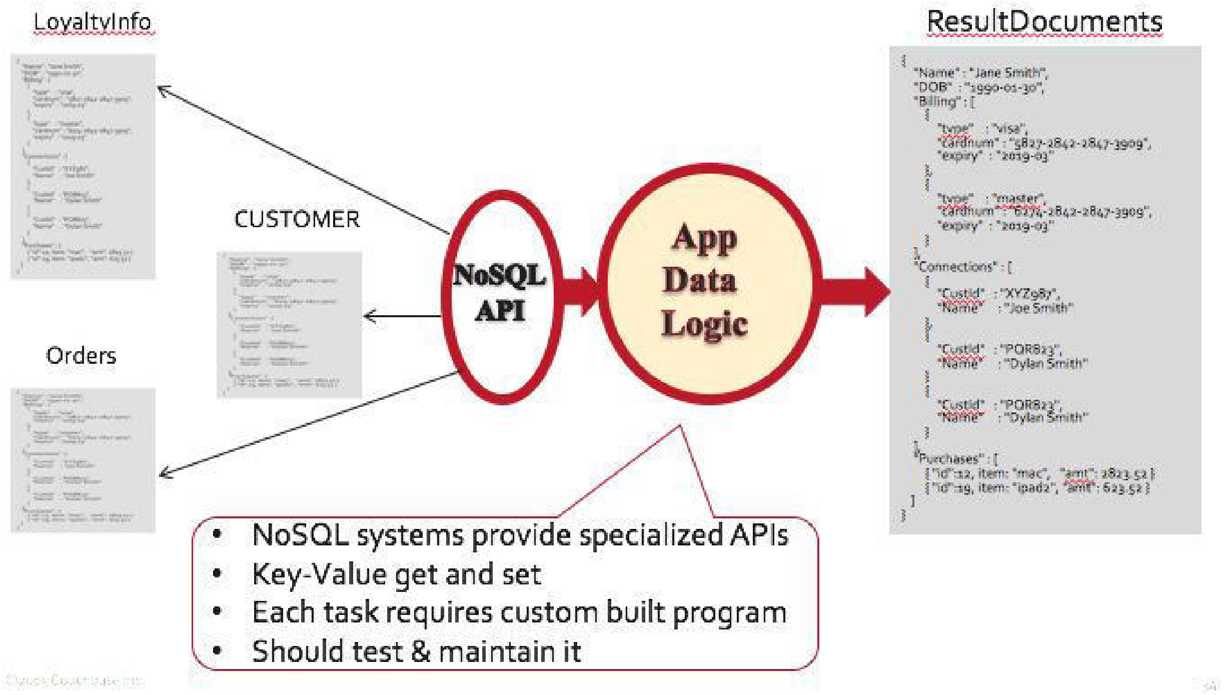
Now, the question is, how would you go about implementing all these for a NoSQL key-value systems with GET and SET APIs? You do this by writing a program to do each task. Of course, writing the application is expensive. Scaling, debugging, and maintaining it makes it even more expensive.

Imagine a use case: Find high-value customers with orders > $10,000. With a NoSQL API approach, you write a program to retrieve the objects and do the filtering, grouping, aggregation in your program.

Find High-Value Customers with Orders > $10000


LOOPING OVER MILLIONS OF CUSTOMERS IN APPLICATION!!!
Query customer objects from database
For each customer object
Find all the order objects for the customer
Calculate the total amount for each order
Sum up the grand total amount for all orders
If grand total amount > $10000, Extract customer data
Add customer to the high-value customer list
Sort the high-value customer list

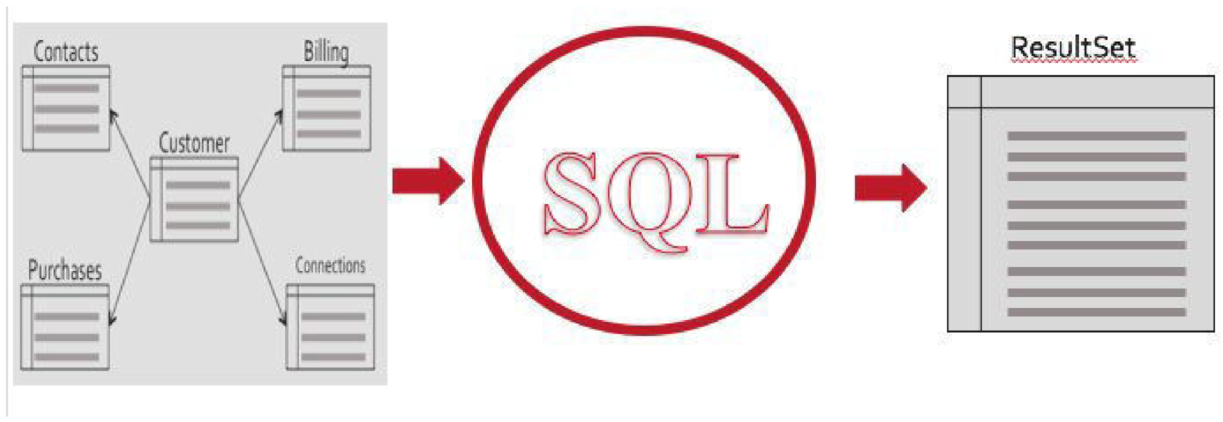
Now, the question is, how would you go about implementing all these in a NoSQL key-value systems with GET and SET APIs? You do this by writing a program to do each task. Of course, writing the application is expensive, scaling, debugging and maintaining it makes it even more expensive.



```
{
  "Name" : "Jane Smith",
  "DOB"  : "1990-01-30",
  "Billing" : [
    {
      "type"    : "visa",
      "cardnum" : "5827-2842-2847-3909",
      "expiry"  : "2019-03"
    },
    {
      "type"    : "master",
      "cardnum" : "6274-2842-2847-3909",
      "expiry"  : "2019-03"
    }
  ],
  "Connections" : [
    {
      "CustId"  : "XYZ987",
      "Name"    : "Joe Smith"
    },
    {
      "CustId"  : "PQR823",
      "Name"    : "Dylan Smith"
    },
    {
      "CustId"  : "PQR823",
      "Name"    : "Dylan Smith"
    }
  ],
  "Purchases" : [
    { "id":12, item: "mac", "amt": 2823.52 }
    { "id":19, item: "ipad2", "amt": 623.52 }
  ]
}
```

- NoSQL systems provide specialized APIs
- Key-Value get and set
- Each task requires custom built program
- Should test & maintain it

This is the reason RDBMS and SQL were invented. SQL does set processing naturally. On the relational model, each of these tasks can be done via a series of statements. Each SQL statement takes set of tuples, does the processing, and produces another set of tuples.

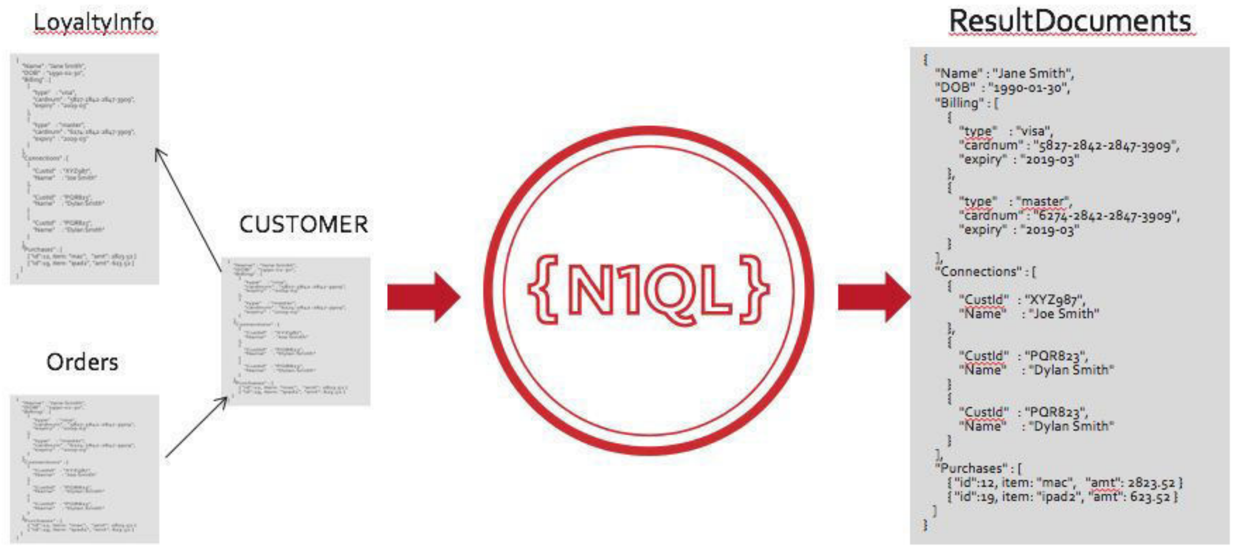


```
SELECT c.CustID, Customers.Name, SUM(o.Amount)
FROM Orders o INNER JOIN LineItems l ON (o.CustID = l.CustID)
INNER JOIN Customers c (o.CustID = c.CustID)
GROUP BY c.CustID, c.Name
HAVING SUM(o.Amount) > 10000
ORDER BY SUM(o.Amount) DESC
```

As the NoSQL key-value databases evolved, they added the map-reduce frameworks so developers can use and cascade them to do complex work. Cassandra, MongoDB, and Couchbase introduced SQL-like languages on top distributed NoSQL database to make the query easier. MongoDB API looks different, but you'll still see something like SQL.

Couchbase N1QL (Non-First Normal Form Query Language) provides all of the select-join-project operations in SQL on the flexible JSON data model. N1QL defines a four-valued boolean logic for boolean expressions that include the new MISSING value in a flexible schema and the collation sequence for data types. In addition, N1QL extends SQL to access and manipulate all parts of nested JSON structure.

N1QL takes sets of JSON documents as input, processes them and gives you a set of JSON documents. Just replace tuples with JSON in SQL definition, you get N1QL.



```
{
  "Name" : "Jane Smith",
  "DOB"  : "1990-01-30",
  "Billing" : [
    {
      "type"    : "visa",
      "cardnum" : "5827-2842-2847-3909",
      "expiry"  : "2019-03"
    },
    {
      "type"    : "master",
      "cardnum" : "6274-2842-2847-3909",
      "expiry"  : "2019-03"
    }
  ],
  "Connections" : [
    {
      "CustId"  : "XYZ987",
      "Name"    : "Joe Smith"
    },
    {
      "CustId"  : "PQR823",
      "Name"    : "Dylan Smith"
    }
    {
      "CustId"  : "PQR823",
      "Name"    : "Dylan Smith"
    }
  ],
  "Purchases" : [
    { "id":12, item: "mac",   "amt": 2823.52 }
    { "id":19, item: "ipad2", "amt": 623.52 }
  ]
}
```

Let's see how we can write the previous query in N1QL. Pretty close to SQL, but it has access to nested data and does the UNNESTING of arrays within ORDERS document.

```
SELECT Customers.ID, Customers.Name, SUM(OrderLine.Amount)
FROM Orders UNNEST Orders.LineItems AS OrderLine
            INNER JOIN Customers ON KEYS Orders.CustID
GROUP BY Customers.ID, Customers.Name
HAVING SUM(OrderLine.Amount) > 10000
ORDER BY SUM(OrderLine.Amount) DESC
```

In addition to SELECT, Couchbase N1QL supports the traditional SQL statements: INSERT, UPDATE, DELETE, and MERGE. Here are some examples:

```
INSERT INTO ORDERS (KEY, VALUE)
     VALUES ("1.ABC.X382", {"O_ID":482, "O_D_ID":3, "O_W_ID":4});

UPDATE ORDERS SET O_CARRIER_ID = ”ABC987”
     WHERE O_ID = 482 AND O_D_ID = 3 AND O_W_ID = 4

DELETE FROM NEW_ORDER
     WHERE NO_D_ID = 291 AND NO_W_ID = 3482 AND NO_O_ID = 2483

MERGE INTO CUSTOMER USING
      (SELECT CID FROM NEWCUST WHERE STATE = 'CA') as NC
ON KEY NC.CID
     WHEN MATCHED THEN UPDATE SET CUSTOMER.STATE= 'CA';
```

Here's the comparison between SQL based RDBMS and N1QL:

Query Features	**SQL on RDBMS**	**N1QL: SQL on JSON**
Objects	Tables, tuples, columns	Keyspaces, JSON Documents, key-value
References	Flat references:Table.column E.g. customer.name, customer.zip	Flat & Nested: keyspace.keyvalue E.g. customer.name, customer.address.zip, Customer.contact[0].phone.mobile
Statements	SELECT, INSERT, UPDATE, DELETE, MERGE	SELECT, INSERT, UPDATE, DELETE, MERGE
Query Operations	● Select, Join, Project, Subqueries ● Strict Schema ● Strict Type checking	● Select, Join, Project, Subqueries ● Nest & Unnest ● Look Ma! No Type Mismatch Errors! ● JSON keys act as columns

Schema	Predetermined Columns	● Fully addressable JSON ● Flexible document structure
Data Types	SQL Data types Conversion Functions	● JSON Data types: Strings, Number Boolean, Objects, Arrays, null ● Conversion Functions
Query Processing	INPUT: Sets of Tuples OUPUT: Set of Tuples	INPUT: Sets of JSON OUTPUT: Set of JSON

Summary:

NoSQL databases give you the ability to read and write massive amounts of data at a substantially low latency and substantially low cost compared to RDBMS.Not having a query is like having a great engine without a steering wheel. With query on JSON from Couchbase, you can ask important questions like -- where can I find my favorite beer?.

```
SELECT          {"Mybeer": "My Beer is " || beer.name || "."},
     Array_agg({"name":brewery.name})  brewery,
     Array_agg({"name":brewery.name, "state":brewery.state,
"city":brewery.city, "location":brewery.geo})    locations,
      Array_count(Array_agg(brewery.NAME))  AS brewery_count
FROM            `beer-sample` beer
LEFT OUTER JOIN `beer-sample` brewery
ON              keys beer.brewery_id
WHERE           beer.type = 'beer'
AND             brewery.type = 'brewery'
AND             brewery.state = 'California'
GROUP BY        beer.NAME
ORDER BY        array_count(array_agg(brewery.NAME)) DESC,
                                 beer.NAME ASC limit 5 ;
```

You have your steering wheel back, now!

NEST and UNNEST: Normalizing and Denormalizing JSON on the Fly

Author: Johan Larson

When modelling data in a JSON database like Couchbase, developers and architects have two options for representing hierarchical data. The first option is to embed the related objects in the parent object, like this:

```
"1" : {"name":"doc 1","subdocs":[{"name":"doc 1.1"},{"name":"doc 1.2"},{"name":"doc 1.3"}]}
"2" : {"name":"doc 2","subdocs":[{"name":"doc 2.1"},{"name":"doc 2.2"}]}
```

The alternative is to represent the related objects separately, and refer to them by ID, like this:

```
"1" : {"name":"doc 1","subdocs":["1.1","1.2","1.3"]}
"2" : {"name":"doc 2","subdocs":["2.1","2.2"]}
"1.1" : {"name":"doc 1.1"}
"1.2" : {"name":"doc 1.2"}
"1.3" : {"name":"doc 1.3"}
"2.1" : {"name":"doc 2.1"}
"2.1" : {"name":"doc 2.2"}
```

Exactly when to use one form or the other is beyond the scope of this article. But whichever is chosen, the N1QL language lets developers convert to the other on the fly. These conversions are done with two operators, NEST and UNNEST.

“UNNEST”-ing Nested Structures

Suppose we have an order-tracking application and we have designed our database object model to have the line items stored within the order document itself (nested), like this:

```
"1" : {
 "order_id": "1",
 "type": "order",
 "customer_id": "24601",
 "total_price": 30.3,
 "lineitems": [
   { "item_id": 576, "quantity": 3, "item_price": 4.99, "base_price": 14.97, "tax": 0.75,
       "final_price": 15.72 },
   { "item_id": 234, "quantity": 1, "item_price": 12.95, "base_price": 12.95, "tax": 0.65,
       "final_price": 13.6 },
   { "item_id": 122, "quantity": 2, "item_price": 0.49, "base_price": 0.98, "final_price":
0.98 }
 ]
}
"5" : {
 "order_id": "5",
 "type": "order",
 "customer_id": "98732",
 "total_price": 428.04,
 "lineitems": [
   { "item_id": 770, "quantity": 3, "item_price": 95.97, "base_price": 287.91, "tax": 14.4,
       "final_price": 302.31 },
   { "item_id": 712, "quantity": 1, "item_price": 125.73, "base_price": 125.73,
       "final_price": 125.73 }
 ]

}
```

Now we want to compute the tax payable for each order. How would we do it? The tax information is not available at the order level; we only know the tax for each line item.

We have to do two things. First, we have to extract the line items from the orders, and then we have to aggregate the taxes by order. The extraction will be done with an UNNEST operation and the aggregation with an aggregation function and a GROUP BY.

We can extract the line items with this query:

```
SELECT * FROM demo UNNEST lineitems
```

That produces this result:

```
[
 {
   "demo": {
     "customer_id": "24601",
     "lineitems": [
       { "base_price": 14.97, "final_price": 15.72, "item_id": 576, "item_price": 4.99,
"quantity": 3,
           "tax": 0.75 },
       { "base_price": 12.95, "final_price": 13.6, "item_id": 234, "item_price": 12.95,
"quantity": 1,
           "tax": 0.65 },
        { "base_price": 0.98, "final_price": 0.98, "item_id": 122, "item_price": 0.49,
"quantity": 2 }
      ],
      "order_id": "1",
      "total_price": 30.3,
      "type": "order"
    },
    "lineitems": { "base_price": 14.97, "final_price": 15.72, "item_id": 576, "item_price":
4.99,
          "quantity": 3, "tax": 0.75 }
  },
  {
    "demo": {
      "customer_id": "24601",
      "lineitems": [
        { "base_price": 14.97, "final_price": 15.72, "item_id": 576, "item_price": 4.99,
"quantity": 3,
          "tax": 0.75
        },
        { "base_price": 12.95, "final_price": 13.6, "item_id": 234, "item_price": 12.95,
"quantity": 1,
          "tax": 0.65
        },
        { "base_price": 0.98, "final_price": 0.98, "item_id": 122, "item_price": 0.49,
"quantity": 2 }
      ],
      "order_id": "1",
      "total_price": 30.3,
      "type": "order"
    },
    "lineitems": { "base_price": 12.95, "final_price": 13.6, "item_id": 234, "item_price":
12.95,
        "quantity": 1, "tax": 0.65 }
  },
  {
    "demo": {
      "customer_id": "24601",
      "lineitems": [
        { "base_price": 14.97, "final_price": 15.72, "item_id": 576, "item_price": 4.99,
"quantity": 3,
            "tax": 0.75 },
        { "base_price": 12.95, "final_price": 13.6, "item_id": 234, "item_price": 12.95,
"quantity": 1,
            "tax": 0.65
        },
        { "base_price": 0.98, "final_price": 0.98, "item_id": 122, "item_price": 0.49,
"quantity": 2 }
      ],
      "order_id": "1",
      "total_price": 30.3,
      "type": "order"
    },
    "lineitems": { "base_price": 0.98, "final_price": 0.98, "item_id": 122, "item_price":
0.49,
        "quantity": 2 }
  },
  {
    "demo": {
        "customer_id": "98732",
        "lineitems": [
          { "base_price": 287.91, "final_price": 302.31, "item_id": 770, "item_price": 95.97,
              "quantity": 3, "tax": 14.4 },
          { "base_price": 125.73, "final_price": 125.73, "item_id": 712, "item_price": 125.73,
              "quantity": 1 }
        ],
        "order_id": "5",
        "total_price": 428.04,
        "type": "order"
      },
      "lineitems": { "base_price": 287.91, "final_price": 302.31, "item_id": 770,
   "item_price": 95.97,
          "quantity": 3, "tax": 14.4 }
    },
    {
      "demo": {
        "customer_id": "98732",
        "lineitems": [
          { "base_price": 287.91, "final_price": 302.31, "item_id": 770, "item_price": 95.97,
              "quantity": 3, "tax": 14.4 },
          { "base_price": 125.73, "final_price": 125.73, "item_id": 712, "item_price": 125.73,
              "quantity": 1 }
        ],
        "order_id": "5",
        "total_price": 428.04,
        "type": "order"
      },
      "lineitems": { "base_price": 125.73, "final_price": 125.73, "item_id": 712,
   "item_price": 125.73,
          "quantity": 1 }
    }
   ]
```

That’s a big result. Let’s look more closely. The result has five objects, each with two fields: “demo” contains the entire original (parent) object, and “lineitems” contains one of the elements of the “lineitems” array in the original object. So we’re extracted each line item, and we’ve kept around each original (parent) object, in case we need it.

We can then group by demo.order_id and sum up lineitems.tax, with this query:

```
SELECT demo.order_id, SUM(lineitems.tax) AS total_tax
FROM demo UNNEST lineitems
GROUP BY demo.order_id
```

Which produces this trim result:

```
[
 {
  "order_id": "1",
  "total_tax": 1.4
 },
 {
  "order_id": "5",
  "total_tax": 14.4
 }
]
```

Using Indexes for UNNEST

Let's think about how the basic query is executed.

```
EXPLAIN SELECT * FROM demo UNNEST lineitems
```

The plan produced is this:

```
[
 {
   "plan": {
     "#operator": "Sequence",
     "~children": [
       {
         "#operator": "PrimaryScan",
         "index": "#primary",
         "keyspace": "demo",
         "namespace": "default",
         "using": "gsi"
       },
       {
         "#operator": "Fetch",
         "keyspace": "demo",
         "namespace": "default"
       },
       {
         "#operator": "Parallel",
         "~child": {
           "#operator": "Sequence",
           "~children": [
             {
               "#operator": "Unnest",
               "as": "lineitems",
               "expr": "(`demo`.`lineitems`)"
             }
           ]
         }
       },
       {
         "#operator": "Parallel",
         "~child": {
           "#operator": "Sequence",
           "~children": [
             {
               "#operator": "InitialProject",
               "result_terms": [
                 {
                   "expr": "self",
                   "star": true
                 }
               ]
             },
             {
               "#operator": "FinalProject"
             }
           ]
         }
       }
     ]
   },
   "text": "SELECT * FROM demo UNNEST lineitems"
 }
]
```

This plan is going to scan the primary index for the demo bucket, fetch every record it references, and feed it into a scan operator.

If we want to be more selective about the orders, that's easy enough with an index. We add a predicate to the query,

```
SELECT * FROM demo UNNEST lineitems WHERE demo.customer_id = "999"
```

We also add an index on the new field,

```
CREATE INDEX demo_customer_id(customer_id)
```

And now the plan contains a shiny new IndexScan operator that uses the new index.

```
        {
          "#operator": "IndexScan",
          "index": "demo_customer_id",
          "index_id": "30061046820ccba9",
          "keyspace": "demo",
          "namespace": "default",
          "spans": [
            {
              "Range": {
                "High": [
                  "\"999\""
                ],
                "Inclusion": 3,
                "Low": [
                  "\"999\""
                ]
              }
            }
          ],
          "using": "gsi"
        }
```

But what if we want to be selective about the line items? Suppose we only want line items with item_id = 567, like this:

```
SELECT * FROM demo UNNEST lineitems WHERE lineitems.item_id = 576
```

For that, we'll need an array index, like this

```
CREATE INDEX demo_lineitem_item_id ON


demo(DISTINCT ARRAY l.item_id FOR l IN lineitems END)
```

This is an index on the item_id fields of the lineitems array in each order. Then we try an EXPLAIN, and nothing has happened. The index doesn't get selected.

The problem is the original query. The current version of Couchbase is very particular about how it uses array indexes, and the UNNEST clause has to be in a particular format to match the index. In particular, look at the "l.item_id" in the index definition. That has to exactly match the field name and prefix in the query. Things work correctly if we change the query like this:

```
SELECT * FROM demo UNNEST lineitems l WHERE l.item_id = 576
```

Then the EXPLAIN shows us a different operator, indicating the index is in use:

```
{
  "#operator": "DistinctScan",
  "scan": {
    "#operator": "IndexScan",
    "index": "demo_lineitem_item_id",
    "index_id": "f6b19179f6dc0496",
    "keyspace": "demo",
    "namespace": "default",
    "spans": [
      {
        "Range": {
          "High": [
            "576"
```

```
                    ],
                    "Inclusion": 3,
                    "Low": [
                      "576"
                    ]
                  }
                }
              ],
              "using": "gsi"
            }
          }
```

More information about array indexes is available here:

Left Outer Unnest

Now, suppose we add a third document to the demo bucket, like this:

```
INSERT INTO demo VALUES ("6", {
 "customer_id": "77077",
 "order_id": "6",
 "total_price": 0,
 "type": "order"
})
```

If we then rerun the query...

```
SELECT * FROM demo UNNEST lineitems
```

...we see a result, but the new order is not included. That's because the new order has no line items, and therefore does not participate in the join.

We can include the new order in the result by switching to a LEFT OUTER UNNEST, which includes documents even when they have no sub-objects.

The query...

```
SELECT * FROM demo LEFT OUTER UNNEST lineitems
```

...yields the same results as the earlier one, but this time with the new order included:

```
{
  "demo": {
    "customer_id": "77077",
    "order_id": "6",
    "total_price": 0,
    "type": "order"
  }
}
```

JOIN With UNNEST

Now let us suppose we have decomposed the orders into main and subordinate objects, perhaps because the principal object had grown too large,and was generating too much network traffic, like this:

```
"1" : {
 "order_id": "1",
 "type": "order",
 "customer_id": "24601",
 "sub" : "1A"
}
"1A" : {
 "type" : "order_sub",
 "order_sub_id" : "1A",
 "total_price": 30.3,
 "lineitems": [
   { "item_id": 576, "quantity": 3, "item_price": 4.99, "base_price": 14.97, "tax": 0.75,
       "final_price": 15.72 },
   { "item_id": 234, "quantity": 1, "item_price": 12.95, "base_price": 12.95, "tax": 0.65,
       "final_price": 13.6 },
   { "item_id": 122, "quantity": 2, "item_price": 0.49, "base_price": 0.98, "final_price":
0.98 }
 ]
}
"5" : {
 "order_id": "5",
 "type": "order",
 "customer_id": "98732",
 "sub" : "5A",
}
"5A" : {
 "type" : "order_sub",
 "order_sub_id" : "5A",
 "total_price": 428.04,
 "lineitems": [
   { "item_id": 770, "quantity": 3, "item_price": 95.97, "base_price": 287.91, "tax": 14.4,
       "final_price": 302.31 },
   { "item_id": 712, "quantity": 1, "item_price": 125.73, "base_price": 125.73,
       "final_price": 125.73 }
 ]
}
```

How does that force us to modify our query? We can join up the objects and subordinate objects like this:

```
SELECT * FROM demo ordr JOIN demo subdata ON KEYS ordr.sub
```

That yields this result:

```
[
 {
   "ordr": {
     "customer_id": "24601",
     "order_id": "1",
     "sub": "1A",
     "type": "order"
   },
   "subdata": {
     "lineitems": [
       {
         "base_price": 14.97,
         "final_price": 15.72,
         "item_id": 576,
         "item_price": 4.99,
         "quantity": 3,
         "tax": 0.75
       },
       {
         "base_price": 12.95,
         "final_price": 13.6,
         "item_id": 234,
         "item_price": 12.95,
         "quantity": 1,
         "tax": 0.65
       },
       {
         "base_price": 0.98,
         "final_price": 0.98,
         "item_id": 122,
         "item_price": 0.49,
         "quantity": 2
```

```
        }
      ],
      "order_sub_id": "1A",
      "total_price": 30.3,
      "type": "order_sub"
    }
  },
  {
    "ordr": {
      "customer_id": "98732",
      "order_id": "5",
      "sub": "5A",
      "type": "order"
    },
    "subdata": {
      "lineitems": [
        {
          "base_price": 287.91,
          "final_price": 302.31,
          "item_id": 770,
          "item_price": 95.97,
          "quantity": 3,
          "tax": 14.4
        },
        {
          "base_price": 125.73,
          "final_price": 125.73,
          "item_id": 712,
          "item_price": 125.73,
          "quantity": 1
        }
      ],
        "order_sub_id": "5A",
        "total_price": 428.04,
        "type": "order_sub"
      }
    }
  ]
```

We can then unnest on subdata.lineitems to split out the individual lineitems:

```
SELECT * FROM demo ordr JOIN demo subdata ON KEYS ordr.sub
UNNEST subdata.lineitems
```

That produces five results (one for each line item), structured like this:

```
{
  "lineitems": {
    "base_price": 14.97,
    "final_price": 15.72,
    "item_id": 576,
    "item_price": 4.99,
    "quantity": 3,
    "tax": 0.75
  },
  "ordr": {
    "customer_id": "24601",
    "order_id": "1",
    "sub": "1A",
    "type": "order"
  },
  "subdata": {
    "lineitems": [
      {
        "base_price": 14.97,
        "final_price": 15.72,
        "item_id": 576,
        "item_price": 4.99,
        "quantity": 3,
        "tax": 0.75
      },
      {
        "base_price": 12.95,
        "final_price": 13.6,
        "item_id": 234,
        "item_price": 12.95,
        "quantity": 1,
        "tax": 0.65
      },
      {
        "base_price": 0.98,
        "final_price": 0.98,
        "item_id": 122,
        "item_price": 0.49,
        "quantity": 2
      }
    ],
    "order_sub_id": "1A",
    "total_price": 30.3,
    "type": "order_sub"
  }
}
```

From there, we can aggregate as before.

“NEST”-ing an Unnested Structure

Now let’s flip the problem. What if the data starts out with orders and line items as separate entries in the database (unnested), and we want to group them together with line items under the documents. For example, we may want for each order, the items included and the quantity of each item.

In this case, the data might be represented like this, using seven separate objects:

```
"1" : { "order_id": "1", "type": "order", "customer_id": "24601", "total_price": 30.3,
       "lineitems": [ "11", "12", "13" ] }
"11" : { "lineitem_id": "11", "type": "lineitem", "item_id": 576, "quantity": 3,
"item_price": 4.99,
       "base_price": 14.97, "tax": 0.75, "final_price": 15.72 }
"12" : { "lineitem_id": "12", "type": "lineitem", "item_id": 234, "quantity": 1,
"item_price": 12.95,
       "base_price": 12.95, "tax": 0.65, "final_price": 13.6 }
"13" : { "lineitem_id": "13", "type": "lineitem", "item_id": 122, "quantity": 2,
"item_price": 0.49,
       "base_price": 0.98, "final_price": 0.98 }
"5" : { "order_id": "5", "type": "order", "customer_id": "98732", "total_price": 428.04,
       "lineitems" : [ "51", "52" ] }
"51" : { "lineitem_id": "51", "type": "lineitem", "item_id": 770, "quantity": 2,
"item_price": 95.97,
       "base_price": 287.91, "tax": 14.4, "final_price": 302.31 }
"52" : { "lineitem_id": "52", "type": "lineitem", "item_id": 712, "quantity": 1,
"item_price": 125.73,
       "base_price": 125.73, "final_price": 125.73 }
```

From these documents, we can use the NEST operator to transform the individual objects into nested structures using this query:

```
SELECT * FROM demo ordr NEST demo li ON KEYS ordr.lineitems
```

The query yields this result:

```
[
 {
   "li": [
     { "base_price": 14.97, "final_price": 15.72, "item_id": 576,
"item_price": 4.99,
       "lineitem_id": "11", "quantity": 3, "tax": 0.75, "type": "lineitem"
},
     { "base_price": 0.98, "final_price": 0.98, "item_id": 122,
"item_price": 0.49, "lineitem_id": "13",
       "quantity": 2, "type": "lineitem" },
     { "base_price": 12.95, "final_price": 13.6, "item_id": 234,
"item_price": 12.95,
       "lineitem_id": "12", "quantity": 1, "tax": 0.65, "type": "lineitem" }
   ],
   "ordr": {
     "customer_id": "24601",
     "lineitems": [
       "11",
       "12",
       "13"
     ],
     "order_id": "1",
     "total_price": 30.3,
     "type": "order"
   }
 },
 {
   "li": [
     { "base_price": 287.91, "final_price": 302.31, "item_id": 770,
"item_price": 95.97,
       "lineitem_id": "51", "quantity": 2, "tax": 14.4, "type": "lineitem"
},
     { "base_price": 125.73, "final_price": 125.73, "item_id":
712,"item_price": 125.73,
       "lineitem_id": "52", "quantity": 1, "type": "lineitem" }
   ],
   "ordr": {
     "customer_id": "98732",
     "lineitems": [
       "51",
       "52"
     ],
     "order_id": "5",
     "total_price": 428.04,
     "type": "order"
   }
 }
]
```

Again, this is quite a big result. But let’s look more closely. There are two objects in the result, one for each order. Each object has two fields. The order fields are under “ordr”, and the line items are in an array under “li”.

But we can simplify further. We just need the “order_id” of each order, and we only need the “item_id” and "quantity" for each line item. We can get the “order_id” from ordr.order_id, and we can extract the “item_id” and "quantity" from the “li” array using an array comprehension, like this:

```
SELECT ordr.order_id, ARRAY {"item_id": l.item_id, "quantity" : l.quantity} FOR l IN li END
as items
FROM demo ordr NEST demo li ON KEYS ordr.lineitems
```

The query produces this trim result:

```
[
 {
   "items": [
     { "item_id": 576, "quantity": 3 },
     { "item_id": 234, "quantity": 1 },
     { "item_id": 122, "quantity": 2 }
   ],
   "order_id": "1"
 },
 {
   "items": [
     { "item_id": 712, "quantity": 1 },
     { "item_id": 770, "quantity": 2 }
   ],
   "order_id": "5"
 }
]
```

Using Indexes for Nest Operations

Let’s return to the original NEST query and check Couchbase will execute it.

```
EXPLAIN SELECT * FROM demo ordr NEST demo li ON KEYS ordr.lineitems
```

That gets us this plan:

```
[
 {
   "plan": {
     "#operator": "Sequence",
     "~children": [
       {
         "#operator": "PrimaryScan",
         "index": "#primary",
         "keyspace": "demo",
         "namespace": "default",
         "using": "gsi"
       },
       {
         "#operator": "Fetch",
         "as": "ordr",
         "keyspace": "demo",
         "namespace": "default"
       },
       {
         "#operator": "Nest",
         "as": "li",
         "keyspace": "demo",
         "namespace": "default",
         "on_keys": "(`ordr`.`lineitems`)"
       },
       {
         "#operator": "Parallel",
         "~child": {
           "#operator": "Sequence",
           "~children": [
             {
               "#operator": "InitialProject",
               "result_terms": [
                 {
                   "expr": "self",
                   "star": true
                 }
               ]
             },
             {
               "#operator": "FinalProject"
             }
           ]
         }
       }
     ]
   },
   "text": "SELECT * FROM demo ordr NEST demo li ON KEYS ordr.lineitems"
 }
]
```

No great mystery here. Couchbase is going to scan the primary index on the demo bucket, and probe the primary index to get each lineitem.

If there is a condition on the primary object that can be served by an index, Couchbase will use it. You can see the difference by adding a predicate and an index like this:

```
CREATE INDEX demo_cust on demo(customer_id)
EXPLAIN SELECT * FROM demo ordr
NEST demo li ON KEYS ordr.lineitems WHERE ord.customer_id = 334
```

Now the plan contains an IndexScan operator, which shows that Couchbase will use the new index:

```
{
  "#operator": "IndexScan",
  "index": "demo_cust",
  "index_id": "74769ea5090a37b7",
  "keyspace": "demo",
  "namespace": "default",
  "spans": [
    {
      "Range": {
        "High": [
          "334"
        ],
        "Inclusion": 3,
        "Low": [
          "334"
        ]
      }
    }
  ],
  "using": "gsi"
}
```

What about conditions on the line items? Suppose we have a predicate item_id=555?

Here, it turns out indexes don't help. Predicates on the right-hand side of NEST operators are applied after the NEST operation, period. And this may have consequences for the data design. Any fields that are needed for selectivity when using NEST operations should be placed in the principal object, not pushed down to the secondary objects.

Left Outer Nest

In the earlier section, we used nest to group external objects referenced by keys into the top-level objects they belonged to. But what about objects that don't have subobjects?

We can test this by adding a third object to the data set, like this:

```
INSERT INTO demo VALUES ("6", {
 "customer_id": "77077",
 "order_id": "6",
 "total_price": 0,
 "type": "order"
})
```

Then we rerun the query, and get exactly the same result as before. Order "6" isn't present. Why? The problem is that order "6" doesn't have a lineitems array, so it isn't included in the join.

How could we add it to the result? First of all we need to switch to a LEFT OUTER NEST operator in the query. But that isn't quite enough. That change alone will also include documents in the demo bucket that are lineitems. We need want only documents that are orders, even if they don't have lineitems. That gets us this final query...

```
SELECT * FROM demo ordr LEFT OUTER NEST demo li ON KEYS ordr.lineitems
WHERE ordr.type = "order"
```

...producing this result, including order "6":

```
[
 {
   "li": [
     { "base_price": 14.97, "final_price": 15.72, "item_id": 576,
"item_price": 4.99,
       "lineitem_id": "11", "quantity": 3, "tax": 0.75, "type": "lineitem"
},
     { "base_price": 12.95, "final_price": 13.6, "item_id": 234,
"item_price": 12.95,
       "lineitem_id": "12", "quantity": 1, "tax": 0.65, "type": "lineitem"
},
     { "base_price": 0.98, "final_price": 0.98, "item_id": 122,
"item_price": 0.49,
       "lineitem_id": "13", "quantity": 2, "type": "lineitem" }
   ],
   "ordr": {
     "customer_id": "24601",
     "lineitems": [
       "11",
       "12",
       "13"
     ],
     "order_id": "1",
     "total_price": 30.3,
     "type": "order"
   }
 },
 {
   "li": [
     { "base_price": 287.91, "final_price": 302.31, "item_id": 770,
"item_price": 95.97,
       "lineitem_id": "51", "quantity": 2, "tax": 14.4, "type": "lineitem"
},
     { "base_price": 125.73, "final_price": 125.73, "item_id": 712,
"item_price": 125.73,
       "lineitem_id": "52", "quantity": 1, "type": "lineitem” }
   ],
   "ordr": {
     "customer_id": "98732",
     "lineitems": [
       "51",
       "52"
     ],
     "order_id": "5",
     "total_price": 428.04,
     "type": "order"
   }
 },
 {
   "ordr": {
     "customer_id": "77077",
     "order_id": "6",
     "total_price": 0,
     "type": "order"
   }
 }
```

```
]
```

JOIN With NEST

Sometimes objects grow too big to keep in a single document, and it makes sense to split the original single object into sub-objects. For example, here we have split out the original line item and price fields from the original objects. Notice the “1A” and “5A” documents.

```
"1" : { "order_id": "1", "type": "order", "customer_id": "24601", "sub" :
"1A" }
"1A" : { "total_price": 30.3, "lineitems": [ "11", "12", "13" ], "type" :
"order_sub",
       "order_sub_id" : "1A" }
"11" : { "lineitem_id": "11", "type": "lineitem", "item_id": 576,
"quantity": 3, "item_price": 4.99,
       "base_price": 14.97, "tax": 0.75, "final_price": 15.72 }
"12" : { "lineitem_id": "12", "type": "lineitem", "item_id": 234,
"quantity": 1, "item_price": 12.95,
       "base_price": 12.95, "tax": 0.65, "final_price": 13.6 }
"13" : { "lineitem_id": "13", "type": "lineitem", "item_id": 122,
"quantity": 2, "item_price": 0.49,
       "base_price": 0.98, "final_price": 0.98 }
"5" : { "order_id": "5", "type": "order", "customer_id": "98732", "sub" :
"5A" }
"5A" : { "total_price": 428.04,
       "lineitems" : [ "51", "52" ], "type" : "order_sub", "order_sub_id" :
"5A" }
"51" : { "lineitem_id": "51", "type": "lineitem", "item_id": 770,
"quantity": 2, "item_price": 95.97,
       "base_price": 287.91, "tax": 14.4, "final_price": 302.31 }
```

```
"52" : { "lineitem_id": "52", "type": "lineitem", "item_id": 712,
"quantity": 1, "item_price": 125.73,
        "base_price": 125.73, "final_price": 125.73 }
```

How then can we reform the original object?

We begin by joining on the “sub” field to recreate the original object...

```
SELECT * FROM demo ordr JOIN demo subdata ON KEYS ords.sub
```

...yielding this:

```
[
 {
   "ordr": {
     "customer_id": "24601",
     "order_id": "1",
     "sub": "1A",
     "type": "order"
   },
   "subdata": {
     "lineitems": [
       "11",
       "12",
       "13"
     ],
     "order_sub_id": "1A",
     "total_price": 30.3,
     "type": "order_sub"
   }
 },
 {
   "ordr": {
     "customer_id": "98732",
     "order_id": "5",
     "sub": "5A",
     "type": "order"
   },
   "subdata": {
     "lineitems": [
       "51",
       "52"
     ],
     "order_sub_id": "5A",
     "total_price": 428.04,
     "type": "order_sub"
   }
 }
]
```

We can then add a NEST clause to join in the line items, too.

```
SELECT * FROM demo ordr JOIN demo subdata ON KEYS ordr.sub
NEST demo li ON KEYS subdata.lineitems
```

That yields this result:

```
[
 {
   "li": [
     { "base_price": 14.97, "final_price": 15.72, "item_id": 576,
"item_price": 4.99,
       "lineitem_id": "11", "quantity": 3, "tax": 0.75, "type": "lineitem"
},
     { "base_price": 12.95, "final_price": 13.6, "item_id": 234,
"item_price": 12.95,
       "lineitem_id": "12", "quantity": 1, "tax": 0.65, "type": "lineitem"
},
     { "base_price": 0.98, "final_price": 0.98, "item_id": 122,
"item_price": 0.49,
       "lineitem_id": "13", "quantity": 2, "type": "lineitem"
   }
  ],
  "ordr": {
   "customer_id": "24601",
   "order_id": "1",
   "sub": "1A",
   "type": "order"
  },
  "subdata": {
   "lineitems": [
    "11",
    "12",
    "13"
   ],
   "order_sub_id": "1A",
   "total_price": 30.3,
   "type": "order_sub"
  }
 },
 {
  "li": [
   { "base_price": 287.91, "final_price": 302.31, "item_id": 770,
"item_price": 95.97,
    "lineitem_id": "51", "quantity": 2, "tax": 14.4, "type": "lineitem"
},
   { "base_price": 125.73, "final_price": 125.73, "item_id": 712,
"item_price": 125.73,
    "lineitem_id": "52", "quantity": 1, "type": "lineitem" }
  ],
  "ordr": {
   "customer_id": "98732",
   "order_id": "5",
      "sub": "5A",
      "type": "order"
    },
    "subdata": {
      "lineitems": [
        "51",
        "52"
      ],
      "order_sub_id": "5A",
      "total_price": 428.04,
      "type": "order_sub"
    }
  }
]
```

Summary

When working with a document database, we may need to pull nested objects from their top-level documents. In Couchbase N1QL, this is done with the UNNEST operator.

Conversely, we may need to group individual objects under their top-level documents. In Couchbase N1QL, this is done with the NEST operator.

Try it Yourself

You can create the dataset used in the UNNEST section by going to the Data Buckets tab of your Couchbase admin console and creating a “demo” bucket. Then go to the Query tab, and execute these two statements:

```
CREATE PRIMARY INDEX ON demo
INSERT INTO demo VALUES ("1",{
 "order_id": "1",
 "type": "order",
 "customer_id": "24601",
 "total_price": 30.3,
 "lineitems": [
   { "item_id": 576, "quantity": 3, "item_price": 4.99, "base_price": 14.97, "tax": 0.75,
       "final_price": 15.72 },
   { "item_id": 234, "quantity": 1, "item_price": 12.95, "base_price": 12.95, "tax": 0.65,
       "final_price": 13.6 },
   { "item_id": 122, "quantity": 2, "item_price": 0.49, "base_price": 0.98, "final_price":
0.98 }
 ]
}),
("5",{
 "order_id": "5",
 "type": "order",
 "customer_id": "98732",
 "total_price": 428.04,
 "lineitems": [
   { "item_id": 770, "quantity": 3, "item_price": 95.97, "base_price": 287.91, "tax": 14.4,
       "final_price": 302.31 },
   { "item_id": 712, "quantity": 1, "item_price": 125.73, "base_price": 125.73,
       "final_price": 125.73 }
 ]
})
```

You can then run the query:

```
SELECT demo.order_id, SUM(lineitems.tax) as total_tax FROM demo UNNEST
lineitems
GROUP BY demo.order_id
```

Then empty the demo bucket:

```
DELETE FROM demo
```

Set up the data for the NEST section:

```
INSERT INTO demo VALUES
("1",{ "order_id": "1", "type": "order", "customer_id": "24601",
       "total_price": 30.3, "lineitems": [ "11", "12", "13" ]}),
("11",{ "lineitem_id": "11", "type": "lineitem", "item_id": 576, "quantity":
3, "item_price": 4.99,
       "base_price": 14.97, "tax": 0.75, "final_price": 15.72 }),
("12",{ "item_id": 234, "type": "lineitem", "lineitem_id": "12", "quantity":
1, "item_price": 12.95,
       "base_price": 12.95, "tax": 0.65, "final_price": 13.6 }),
("13",{ "lineitem_id": "13", "type": "lineitem", "item_id": 122, "quantity":
2, "item_price": 0.49,
       "base_price": 0.98, "final_price": 0.98 }),
("5",{ "order_id": "5", "type": "order", "customer_id": "98732",
"total_price": 428.04,
       "lineitems" : [ "51", "52" ] }),
("51",{ "lineitem_id": "51", "type": "lineitem", "item_id": 770, "quantity":
2, "item_price": 95.97,
       "base_price": 287.91, "tax": 14.4, "final_price": 302.31 }),
("52",{ "lineitem_id": "52", "type": "lineitem", "item_id": 712, "quantity":
1, "item_price": 125.73,
       "base_price": 125.73, "final_price": 125.73 })
```

And run the query:

```
SELECT ordr.order_id, ARRAY {"item_id": l.item_id, "quantity" : l.quantity}
FOR l IN li END as items
FROM demo ordr NEST demo li ON KEYS ordr.lineitems
```

Working With Schemas in a Schema-Flexible Datastore

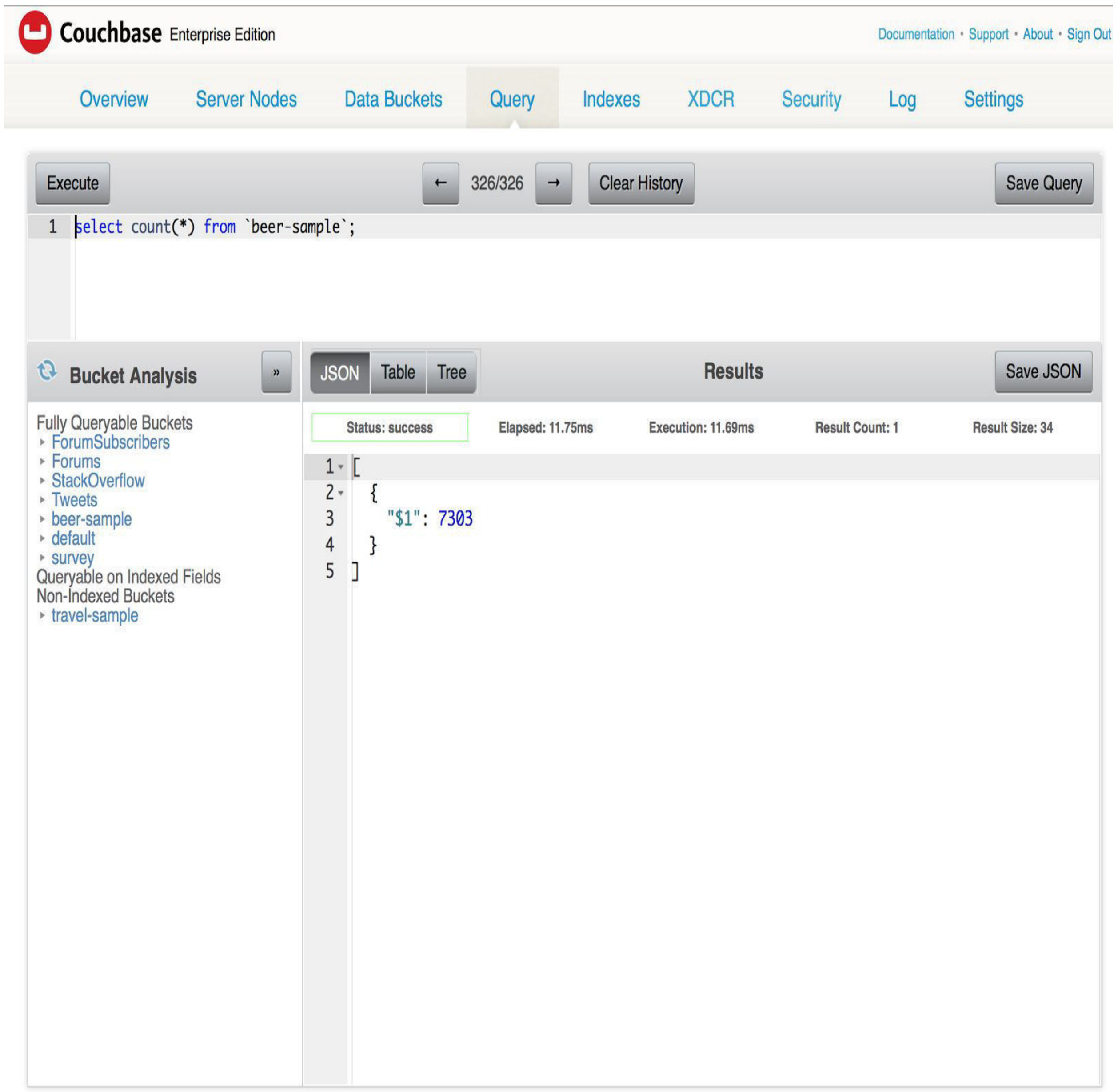
Author: Eben Haber

A schema-flexible JSON datastore, such as Couchbase, provides a big advantage in permitting the application developer to manage schema evolution and referential integrity in a way best suited to the requirements of the application. There's no need to bring down the database or get permission from the DBAs to change the schema, the application can just start writing data in a new schema, with the application logic handling new and old data schemas as appropriate.

This flexibility comes at a cost, however, beyond the need to write more sophisticated application code. Without a formal schema definition, it can be much harder for people to know the data structure details that are needed to write application code, queries, or define indexes:

- How many distinct document schemas exist in a data store?
- How frequently are fields present?
- What is the data like in each field? Type? Format? Range?

Couchbase 4.5 introduced a new web-based query interface.

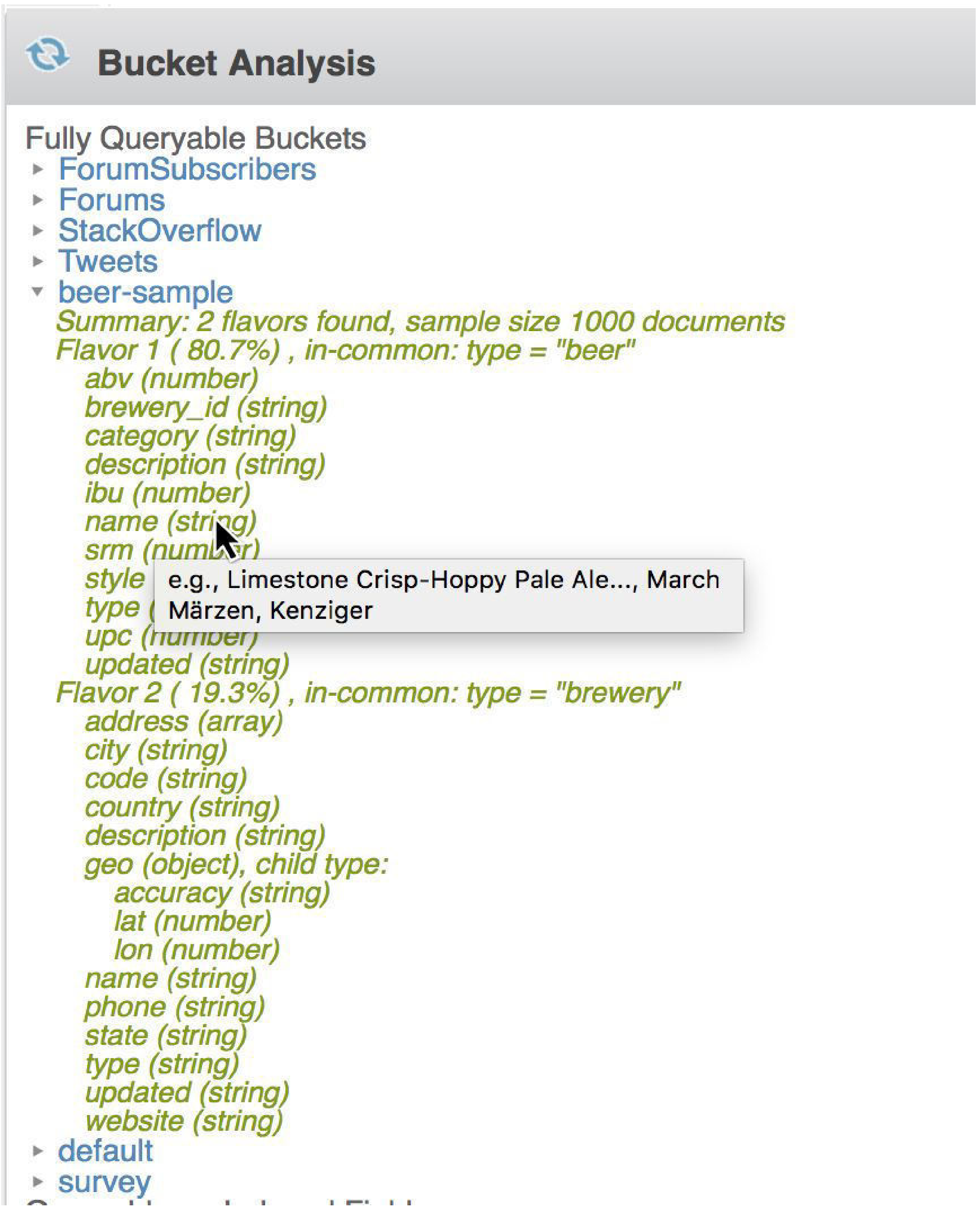


```
select count(*) from `beer-sample`;
```

Status: success   Elapsed: 11.75ms   Execution: 11.69ms   Result Count: 1   Result Size: 34

```
[
  {
    "$1": 7303
  }
]
```

This interface includes the “Bucket Analysis” panel, where if you click the triangle next to a bucket name, something very much like a schema appears:



Bucket Analysis

Fully Queryable Buckets

- ForumSubscribers
- Forums
- StackOverflow
- Tweets
- beer-sample

```
Summary: 2 flavors found, sample size 1000 documents
Flavor 1 ( 80.7%) , in-common: type = "beer"
   abv (number)
   brewery_id (string)
   category (string)
   description (string)
   ibu (number)
   name (string)
   srm (number)
   style
   type
   upc (number)
   updated (string)
Flavor 2 ( 19.3%) , in-common: type = "brewery"
   address (array)
   city (string)
   code (string)
   country (string)
   description (string)
   geo (object), child type:
      accuracy (string)
      lat (number)
      lon (number)
   name (string)
   phone (string)
   state (string)
   type (string)
   updated (string)
   website (string)
```

e.g., Limestone Crisp-Hoppy Pale Ale..., March Märzen, Kenziger

- default
- survey

How is this possible, when Couchbase doesn't have or enforce schemas? Behind the scenes, this panel is using N1QL's INFER command, which takes a random sample of documents from a bucket, and infers what schema(s) exist for those documents. The example above looks at the beer-sample bucket (which is available with every instance of Couchbase). Note that:

- INFER found not one but two schemas with different sets of fields, these schemas are called "flavors". There are some fields in common between the flavors (description, name, and type), but more fields are different between the two flavors.
- Hovering the mouse over a field name shows some sample values for that field.
- Since INFER keeps track of some of the sample values for each field, it knows which fields only have a single distinct value for the documents in flavor. In this case, it sees only a single value for the field named, "type", so it lists that field as something "in-common" for all documents in each flavor. Using a "type" field is a common pattern when data modeling with Couchbase, yet there are no semantics to the name, it could be called "type" or "kind" or "category" or anything. The INFER command helps users learn what approach was taken by the person doing data modeling.

INFER can be run as a regular N1QL command to derive details about a bucket's documents. How does INFER work?

- INFER starts with a random sample of documents from a bucket. By default, it asks for 1000 documents, but a different sample size may be specified.
- INFER extracts a schema from each document (field names, types), maintaining a hash table of all distinct schemas seen thus far. Two schemas are identical if they have the exact same fields and types. Maintaining all distinct schemas could be expensive if there were many of them - in theory every document in a bucket could have a different schema - but in practice most business data follows regular patterns and there aren't many distinct schemas. Sample values for each field are maintained to help users understand a field's format, though long values are truncated to 32 bytes to

save space. When two identical schemas are combined, the sample values for each are combined, up to a specified maximum (by default at most 5 sample values are stored). INFER also keeps track of the number of distinct documents matching each schema.

- Once all distinct schemas are found, similar schemas are merged into schema “flavors”. Two schemas are considered similar if the fraction of shared top-level fields is greater than the “similarity_metric” (default 0.6). If a field is a simple type, it is considered shared if both schemas have a field with the same name and type. If a field has an object type, then the similarity is considered recursively on the subtypes. If you set the similarity_metric to 0, then all schemas are merged into a single, universal-relation-like schema. At the other extreme, a similarity metric of 1.0 would mean that each distinct schema would become its own flavor. Flavors are different from schemas in one important way: with a schema, a field is either present or not. With a flavor, a field can be present some of the time, thus flavors keep track of the frequency that each field appears.
- Each flavor is described using a JSON document whose format is an extension of the draft IETF standard for JSON schemas (http://json-schema.org). The extensions add fields for describing sample values and how often each field is found.

Let’s look at an example: the output of INFER for the ‘beer-sample’ bucket. Note how this is an array of flavors, each flavor following the json-schema format. At the top level each flavor has a “#docs” field which indicates how many documents from the sample matched this flavor. In the schema description, each field has a name and type, and also “#docs” and “%docs” indicating how often the field occurred. Sample values are also included for each field.

```
[
 [
   {
    "#docs": 807,
    "$schema": "http://json-schema.org/schema#",
    "Flavor": "type = \"beer\"",
    "properties": {
```

```
"abv": {
  "#docs": 807,
  "%docs": 100,
  "samples": [
    5,
    5.4,
    0,
    4.9,
    8
  ],
  "type": "number"
},
"brewery_id": {
  "#docs": 807,
  "%docs": 100,
  "samples": [
    "limburg_beer_company",
    "caldera_brewing",
    "western_reserve_brewing",
    "anchor_brewing",
    "traquair_house_brewery"
  ],
  "type": "string"
},
"category": {
  "#docs": 593,
  "%docs": 73.48,
  "samples": [
    "North American Lager",
    "North American Ale",
    "British Ale",
    "Irish Ale",
    "Other Style"
  ],
  "type": "string"
},
"description": {
  "#docs": 807,
  "%docs": 100,
  "samples": [
    "Westmalle Dubbel is a dark, re...",
    "A conundrum in the world of be...",
    "Black, smooth and easy to drin...",
    "A true German-style Kölsch. O...",
    ""
  ],
  "type": "string"
},
"ibu": {
  "#docs": 807,
  "%docs": 100,
  "samples": [
    0,
    41,
    30,
    38,
    35
  ],
  "type": "number"
},
"name": {
  "#docs": 807,
  "%docs": 100,
  "samples": [
    "Witbier",
    "Dry Hop Orange",
    "Cloud Nine Witbier",
    "Anchor Steam",
    "Jacobite Ale"
  ],
  "type": "string"
},
"srm": {
  "#docs": 807,
  "%docs": 100,
  "samples": [
    0,
    12.5,
    47,
    35,
    45
  ],
  "type": "number"
},
"style": {
  "#docs": 593,
  "%docs": 73.48,
  "samples": [
    "American-Style Lager",
    "American-Style Amber/Red Ale",
    "American-Style Pale Ale",
    "American-Style Stout",
    "Classic English-Style Pale Ale"
  ],
  "type": "string"
},
"type": {
  "#docs": 807,
  "%docs": 100,
  "samples": [
    "beer"
  ],
  "type": "string"
},
"upc": {
  "#docs": 807,
  "%docs": 100,
  "samples": [
    0,
    7
  ],
  "type": "number"
},
"updated": {
  "#docs": 807,
  "%docs": 100,
  "samples": [
        "2010-07-22 20:00:20",
        "2011-07-23 20:37:34",
        "2011-08-15 11:46:53",
        "2011-05-17 03:19:48",
        "2011-02-16 12:38:43"
      ],
      "type": "string"
    }
  }
},
{
  "#docs": 193,
  "$schema": "http://json-schema.org/schema#",
  "Flavor": "type = \"brewery\"",
  "properties": {
    "address": {
      "#docs": 193,
      "%docs": 100,
      "items": {
        "#schema": "FieldType",
        "subtype": null,
        "type": "string"
      },
      "maxItems": 1,
      "minItems": 0,
      "samples": [
        [
          "7556 Pine Road"
        ],
        [
          "Val Dieu 225"
        ],
        [
          "6 Chapel Close"
        ],
        [
          "210 Swanson Avenue"
        ],
        [
      "808 West Main Street"
    ]
  ],
  "type": "array"
},
"city": {
  "#docs": 193,
  "%docs": 100,
  "samples": [
    "Arena",
    "Aubel",
    "South Stoke",
    "Lake Havasu City",
    "Ashland"
  ],
  "type": "string"
},
"code": {
  "#docs": 193,
  "%docs": 100,
  "samples": [
    "53503",
    "",
    "86403",
    "54806",
    "94019"
  ],
  "type": "string"
},
"country": {
  "#docs": 193,
  "%docs": 100,
  "samples": [
    "United States",
    "Belgium",
    "United Kingdom",
    "Canada",
    "Denmark"
  ],
  "type": "string"
},
"description": {
  "#docs": 193,
  "%docs": 100,
  "samples": [
    "Since firing up its brew kettl...",
    "(512) Brewing Company is a mic...",
    "An artisanal (non-conglomerate...",
    "Tröegs Brewing Company was es...",
    ""
  ],
  "type": "string"
},
"geo": {
  "#docs": 175,
  "%docs": 90.67,
  "properties": {
    "accuracy": {
      "#docs": 175,
      "%docs": 100,
      "samples": [
        "RANGE_INTERPOLATED",
        "GEOMETRIC_CENTER",
        "ROOFTOP",
        "APPROXIMATE"
      ],
      "type": "string"
    },
    "lat": {
      "#docs": 175,
      "%docs": 100,
      "samples": [
        43.1706,
        50.7046,
        51.5462,
        34.4686,
        46.5872
      ],
      "type": "number"
    },
    "lon": {
      "#docs": 175,
      "%docs": 100,
      "samples": [
        -89.9324,
        5.822,
        -1.1355,
        -114.341,
        -90.8921
      ],
      "type": "number"
    }
  },
  "samples": [
    {
      "accuracy": "RANGE_INTERPOLATED",
      "lat": 43.1706,
      "lon": -89.9324
    },
    {
      "accuracy": "GEOMETRIC_CENTER",
      "lat": 50.7046,
      "lon": 5.822
    },
    {
      "accuracy": "RANGE_INTERPOLATED",
      "lat": 51.5462,
      "lon": -1.1355
    },
    {
      "accuracy": "ROOFTOP",
      "lat": 34.4686,
      "lon": -114.341
    },
    {
      "accuracy": "ROOFTOP",
      "lat": 46.5872,
      "lon": -90.8921
    }
  ],
  "type": "object"
},
"name": {
  "#docs": 193,
  "%docs": 100,
  "samples": [
    "Lake Louie Brewing",
    "Brasserie de l'Abbaye Val-Dieu",
    "Ridgeway Brewing",
    "Mudshark Brewing",
    "South Shore Brewery"
  ],
  "type": "string"
},
"phone": {
  "#docs": 193,
  "%docs": 100,
  "samples": [
    "1-608-753-2675",
    "32-087-68-75-87",
    "44-(01491)-873474",
    "1-928-453-2981",
    "1-715-682-9199"
  ],
  "type": "string"
},
"state": {
  "#docs": 193,
  "%docs": 100,
  "samples": [
    "Wisconsin",
    "Lige",
    "Oxford",
    "Arizona",
    "Brabant Wallon"
  ],
        "type": "string"
      },
      "type": {
        "#docs": 193,
        "%docs": 100,
        "samples": [
          "brewery"
        ],
        "type": "string"
      },
      "updated": {
        "#docs": 193,
        "%docs": 100,
        "samples": [
          "2010-07-22 20:00:20",
          "2010-12-13 19:30:01"
        ],
        "type": "string"
      },
      "website": {
        "#docs": 193,
        "%docs": 100,
        "samples": [
          "http://www.jenningsbrewery.co....",
          "http://www.mudsharkbrewingco.c...",
          "http://512brewing.com/",
          "http://www.darkhorsebrewery.co...",
          ""
        ],
        "type": "string"
      }
    }
  }
 ]
]
```

The format for running the INFER command is as follows:

```
INFER [ WITH { “parameter” : value, ... } ]
```

The most common parameters are:

- sample_size - how many documents to use when inferring schemas (default 1000).
- num_sample_values - how many sample values to retain for each field (default 5).
- similarity_metric - how similar two schemas must be to be merged into the same flavor (default 0.6).

For example, to INFER schemas for beer-sample using a random sample of 2000 documents, and maintaining 10 sample values for each field, you would use:

```
INFER `beer-sample` with {“sample_size”: 2000, “num_sample_values”: 10}
```

The output from INFER is quite detailed, and can be very useful for:

- Learning about what fields exist and their data formats.
- Identifying different schema versions, and finding out how many documents exist for each version.
- Data cleaning, identifying fields that are used rarely or that should always be present but aren’t.

It should be noted that INFER, being based on random sampling, is not deterministic: You will likely get slightly different results each time you run it. You also might not pick up certain document schemas if they occur too rarely with respect to the specified sample size. On the whole, however, it is a powerful tool to help people understand what schemas are present in the documents of their schema-flexible data store.

Indexing

Indexing JSON in Couchbase

Author: Keshav Murthy

Introduction

There are three things important in database systems: performance, performance, performance. Creating the right index, with the right keys, right order, and right expression is critical to query performance in any database system. That's true for Couchbase as well.

We've discussed data modeling for JSON and querying on JSON earlier. In this article, we'll discuss indexing options for JSON in Couchbase.

Couchbase 4.5 can create two types of indices:

1. Standard global secondary index.
2. Memory-Optimized global secondary index.

The standard secondary index stores uses the ForestDB storage engine to store the B-Tree index and keeps the optimal working set of data in the buffer. That means, the total size of the index can be much bigger than the amount of memory available in each index node.

A memory-optimized index uses a novel lock-free skiplist to maintain the index and keeps 100% of the index data in memory. A memory-optimized index has better latency for index scans and can also process the mutations of the data much faster. Both standard and memory-optimized indexes implement multi-version concurrency control (MVCC) to provide consistent index scan results and high throughput. During cluster installation, choose the type of index. The goal is to give you an overview of various indices you create in each of these services so that your queries can execute efficiently. The goal of this article is not to describe or compare and contrast these two types of index services. It does not cover the Full Text Index (FTS), in developer preview now. Another topic not covered by this article is how the index selection is made for scans and joins.

Let's take travel-sample dataset shipped with Couchbase 4.5 to walk through this. To try out these indices, install Couchbase 4.5. On your web console, go to Settings->Sample Buckets to install travel-sample.
Here are the various indices you can create.

```
Primary Index
Named primary index
Secondary index
Composite Secondary Index
Functional index
Array Index
ALL array
ALL DISTINCT array
Partial Index
Duplicate Indices
Covering Index
```

Background

Couchbase is a distributed database. It supports flexible data model using JSON. Each document in a bucket will have a user-generated unique document key. This uniqueness is enforced during insertion of the data.

Here's an example document.

```
select meta().id, travel
from `travel-sample` travel
where type = 'airline' limit 1;
[
   {
       "id": "airline_10",
       "travel": {
           "callsign": "MILE-AIR",
           "country": "United States",
           "iata": "Q5",
           "icao": "MLA",
           "id": 10,
           "name": "40-Mile Air",
           "type": "airline"
           }
       }
   ]
```

1. Primary Index

```
create the primary index on 'travel-sample';
```

The primary index is simply the index on the document key on the whole bucket. The Couchbase data layer enforces the uniqueness constraint on the document key. The primary index, like every other index, is maintained asynchronously. The primary index is used for full bucket scans (primary scans) when the query does not have any filters (predicates) or no other index or access path can be used.

Here is the metadata for this index:

```
    select * from system:indexes where name = ‘#primary’;
    "indexes": {
       "datastore_id": "http://127.0.0.1:8091",
       "id": "f6e3c75d6f396e7d",
       "index_key": [],
       "is_primary": true,
       "keyspace_id": "travel-sample",
       "name": "#primary",
       "namespace_id": "default",
       "state": "online",
       "using": "gsi"
       }
```

The metadata gives you additional information on the index: Where the index resides (datastore_id), its state (state) and the indexing method (using).

2. Named Primary Index

CREATE PRIMARY INDEX `def_primary` ON `travel-sample`

You can also name the primary index. The rest of the features of the primary index are the same, except the index is named. A good side effect of this is that you can have multiple primary indices in the system. Duplicate indices help with high availability as well as query load distribution throughout them. This is true for both primary indices and secondary indices.

```
select meta().id as documentkey, `travel-sample` airline
from `travel-sample`
where type = 'airline' limit 1;
 {
   "airline": {
     "callsign": "MILE-AIR",
     "country": "United States",
     "iata": "Q5",
     "icao": "MLA",
     "id": 10,
     "name": "40-Mile Air",
     "type": "airline"
   },
   "documentkey": "airline_10"
 }
```

3. Secondary Index

The secondary index is an index on any key-value or document-key. This index can be any key within the document. The key can be of any time: scalar, object, or array. The query has to use the same type of object for the query engine to exploit the index.

```
CREATE INDEX travel_name ON `travel-sample`(name);
name is a simple scalar value.
{    "name": "Air France"   }
CREATE INDEX travel_geo on `travel-sample`(geo);
```

geo is an object embedded within the document. Example:

```
   "geo": {
       "alt": 12,
       "lat": 50.962097,
       "lon": 1.954764
```

```
        }
```

Creating indexes on keys from nested objects is straightforward.

```
CREATE INDEX travel_geo on `travel-sample`(geo.alt);
CREATE INDEX travel_geo on `travel-sample`(geo.lat);
```

Schedule is an array of objects with flight details. This indexes on the complete array. Not exactly useful unless you're looking for the whole array.

```
CREATE INDEX travel_schedule ON `travel-sample`(schedule);
```

Example:

```
"schedule": [
        {
            "day": 0,
            "flight": "AF198",
            "utc": "10:13:00"
            },
        {
            "day": 0,
            "flight": "AF547",
            "utc": "19:14:00"
            },
        {
            "day": 0,
            "flight": "AF943",
            "utc": "01:31:00"
            },
        {
            "day": 1,
            "flight": "AF356",
            "utc": "12:40:00"
            },
        {
            "day": 1,
            "flight": "AF480",
            "utc": "08:58:00"
            },
        {
            "day": 1,
            "flight": "AF250",
            "utc": "12:59:00"
            }
]
```

4. Composite Secondary Index

It's common to have queries with multiple filters (predicates). So, you want the indices with multiple keys so the indices can return only the qualified document keys. Additionally, if a query is referencing only the keys in the index, the query engine will simply answer the query from the index scan result without going to the data nodes. This is a commonly exploited performance optimization.

```
CREATE INDEX travel_info ON `travel-sample`(name, type, id, icoo, iata);
```

Each of the keys can be a simple scalar field, object, or an array. For the index filtering to be exploited, the filters have to use respective object type in the query filter. The keys to the secondary indices can include document keys (meta().id) explicitly if you need to filter on it in the index.

5. Functional Index

It's common to have names in the database with a mix of upper and lower cases. When you need to search, "John," you want it to search for any combination of "John," "john," etc. Here's how you do it.

```
CREATE INDEX travel_cxname ON `travel-sample`(LOWER(name));
```

Provide the search string in lowercase and the index will efficiently search for already lowercased values in the index.

```
    EXPLAIN SELECT * FROM `travel-sample` WHERE LOWER(name) = “john”;
    {
       "#operator": "IndexScan",
       "index": "travel_cxname",
       "index_id": "2f39d3b7aac6bbfe",
       "keyspace": "travel-sample",
       "namespace": "default",
       "spans": [
       {
           "Range": {
               "High": [
               "\"john\""
               ],
               "Inclusion": 3,
               "Low": [
               "\"john\""
               ]
           }
       }
       ],
```

You can use complex expressions in this functional index.

```
   CREATE INDEX travel_cx1 ON `travel-sample`(LOWER(name),
      length*width, round(salary));
```

6. Array Index

JSON is hierarchical. At the top level, it can have scalar fields, objects, or arrays. Each object can nest other objects and arrays. Each array can have other objects and arrays. And so on. The nesting continues.
When you have this rich structure, here's how you index a particular array, or a field within the sub-object.
Consider the array, schedule:

```
   schedule:
   [
      {
          "day" : 0,
          "special_flights" :
          [
          {
              "flight" : "AI111", "utc" : ”1:11:11"
              },
          {
              "flight" : "AI222", "utc" : ”2:22:22"
              }
              ]
          },
      {
          "day": 1,
          "flight": "AF552",
          "utc": "14:41:00"
          }
      ]
   CREATE INDEX travel_sched ON `travel-sample`
   (ALL DISTINCT ARRAY v.day FOR v IN schedule END)
```

This index key is an expression on the array to clearly reference only the elements needed to be indexed. schedule the array we're dereferencing into. v is the variable we've implicitly declared to reference each element/object within the array: schedule v.day refers to the element within each object of the array schedule.

The query below will exploit the array index.

```
EXPLAIN SELECT * FROM `travel-sample`
WHERE ANY v IN SCHEDULE SATISFIES v.day = 2 END;
{
   "#operator": "DistinctScan",
   "scan": {
       "#operator": "IndexScan",
       "index": "travel_sched",
       "index_id": "db7018bff5f10f17",
       "keyspace": "travel-sample",
       "namespace": "default",
       "spans": [
       {
           "Range": {
               "High": [
               "2"
               ],
               "Inclusion": 3,
               "Low": [
               "2"
               ]
               }
           }
       ],
       "using": "gsi"
```

```
    }
```

Because the key is a generalized expression, you get the flexibility to apply additional logic and processing on the data before indexing. For example, you can create functional indexing on elements of each array. Because you're referencing individual fields of the object or element within the array, the index creation, size, and search are efficient. The index above stores only the distinct values within an array. To store all elements of an array in an index, use the DISTINCT modifier to the expression.

```
CREATE INDEX travel_sched ON `travel-sample`
   (ALL ARRAY v.day FOR v IN schedule END)
```

7. Partial Index

So far, the indices we've created will create indices on the whole bucket. Because the Couchbase data model is JSON and JSON schema are flexible, an index may not contain entries to documents with absent index keys. That's expected. Unlike relational systems, where each type of row is in a distinct table, Couchbase buckets can have documents of various types. Typically, customers include a type field to differentiate distinct types.

```
{
   "airline": {
       "callsign": "MILE-AIR",
       "country": "United States",
       "iata": "Q5",
       "icao": "MLA",
       "id": 10,
       "name": "40-Mile Air",
       "type": "airline"
       },
   "documentkey": "airline_10"
   }
```

When you want to create an index of airline documents, you can simply add the type field for the WHERE clause of the index.

```
CREATE INDEX travel_info ON `travel-sample`(name, id, icoo, iata)
WHERE type = 'airline';
```

This will create an index only on the documents that have (type = ‘airline'). In your queries, you'd need to include the filter (type = ‘airline') in addition to other filters so this index qualifies.

You can use complex predicates in the WHERE clause of the index. Various use cases to exploit partial indexes are:

1. Partitioning a large index into multiple indices using the mod function.
2. Partitioning a large index into multiple indices and placing each index into distinct indexer nodes.
3. Partitioning the index based on a list of values. For example, you can have an index for each state.
4. Simulating index range partitioning via a range filter in the WHERE clause. One thing to remember is Couchbase N1QL queries will use one partitioned index per query block. Use UNION ALL to have a query exploit multiple partitioned indices in a single query.

8. Duplicate Index

This isn't really a special type of index, but a feature of Couchbase indexing. You can create duplicate indexes with distinct names.

```
CREATE INDEX i1 ON `travel-sample`(LOWER(name),id, icoo)
WHERE type = ‘airline’;
CREATE INDEX i2 ON `travel-sample`(LOWER(name),id, icoo)
WHERE type = ‘airline’;
CREATE INDEX i3 ON `travel-sample`(LOWER(name),id, icoo)
WHERE type = ‘airline’;
```

All three indices have identical keys, identical WHERE clause; Only difference is the name of the indices. You can choose their physical location using the WITH clause of the CREATE INDEX. During query optimization, query will choose one of the names. You see that in your plan. During query runtime, these indices are used in round-robin fashion to distribute the load. This gives you scale-out, multi-dimensional scaling, performance, and high availability. Not bad!

9. Covering Index

Index selection for a query solely depends on the filters in the WHERE clause of your query. After the index selection is made, the engine analyzes the query to see if it can be answered using only the data in the index. If it does, query engine skips retrieving the whole document. This is a performance optimization to consider while designing the indices.

Summary

Let's put together a partitioned composite functional array index now!

```
CREATE INDEX travel_all ON `travel-sample`(
iata,
LOWER(name),
UPPER(callsign),
           ALL DISTINCT ARRAY p.model FOR p IN jets END),
TO_NUMBER(rating),
           meta().id
) WHERE LOWER(country) = "united states" AND type = "airline";
```

References

Nitro: A Fast, Scalable In-Memory Storage Engine for NoSQL Global Secondary Index : http://vldb2016.persistent.com/industrial_track_papers.php
Couchbase: http://www.couchbase.com
Couchbase Documentation: http://docs.couchbase.com

Designing Index for Query in Couchbase N1QL

Author: Sitaram Vemulapalli

Put simply: Every flight has a flight plan. Every query has a query plan. The performance and efficiency of a query depend on its plan. The N1QL query engine prepares the query plan for each query. N1QL uses a rule-based optimizer. The creation and selection of the right index have a major influence on both performance and efficiency.

This article will walk through the step-by-step process of how to create an index and modify the query for optimal performance.

We'll use travel-sample dataset shipped with Couchbase. Install travel-sample shipped with Couchbase 4.5.1, then drop all of the indexes to start with a clean slate.

```
DROP INDEX `travel-sample`.def_city;
DROP INDEX `travel-sample`.def_faa;
DROP INDEX `travel-sample`.def_icao;
DROP INDEX `travel-sample`.def_name_type;
DROP INDEX `travel-sample`.def_airportname;
DROP INDEX `travel-sample`.def_schedule_utc;
DROP INDEX `travel-sample`.def_type;
DROP INDEX `travel-sample`.def_sourceairport;
DROP INDEX `travel-sample`.def_route_src_dst_day;
DROP INDEX `travel-sample`.def_primary;
```

We will walk through the step-by-step process and create the index for the following query. In this query, we're looking to get airlines in the United States with an id between zero and 1000, ordered by id. We're interested in the 10 airlines, ordered by id and skipping the first five.

```
SELECT country, id, name
FROM `travel-sample`
WHERE type = "airline"
AND country = "United States"
AND id BETWEEN 0 AND 1000
ORDER BY id
LIMIT 10
OFFSET 5;
```

Step 1: Using Primary Index

The primary index is the simplest index, indexing all of the documents in `travel-sample`. Any query can exploit the primary index to execute the query.

```
CREATE PRIMARY INDEX ON  `travel-sample`;
SELECT country, id, name
FROM `travel-sample`
WHERE type = "airline"
AND country = "United States"
   AND id BETWEEN 0 AND 1000
ORDER BY id
LIMIT 10
OFFSET 5;
{
    "results": [
       {
           "country": "United States",
           "id": 149,
           "name": "Air Cargo Carriers"
       },
       {
           "country": "United States",
           "id": 210,
        "name": "Airlift International"
    },
    {
        "country": "United States",
        "id": 281,
        "name": "America West Airlines"
    },
    {
        "country": "United States",
        "id": 282,
        "name": "Air Wisconsin"
    },
    {
        "country": "United States",
        "id": 287,
        "name": "Allegheny Commuter Airlines"
    },
    {
        "country": "United States",
        "id": 295,
        "name": "Air Sunshine"
    },
    {
        "country": "United States",
        "id": 315,
        "name": "ATA Airlines"
    },
    {
        "country": "United States",
        "id": 397,
        "name": "Arrow Air"
    },
        {
            "country": "United States",
            "id": 452,
            "name": "Atlantic Southeast Airlines"
        },
        {
            "country": "United States",
            "id": 659,
            "name": "American Eagle Airlines"
        }
    ]
}
EXPLAIN
SELECT country, id, name
FROM `travel-sample`
WHERE type = "airline"
AND country = "United States"
   AND id BETWEEN 0 AND 1000
ORDER BY id
LIMIT 10
OFFSET 5;
   "~children": [
            {
               "#operator": "PrimaryScan",
               "index": "def_primary",
               "keyspace": "travel-sample",
               "namespace": "default",
              "using": "gsi"
            },
            {
           "#operator": "Fetch",
               "keyspace": "travel-sample",
                    "namespace": "default"
                }
            ]
```

While the query plan tells you indexes selected and predicates pushed down, it won't tell you the amount of work (index items scanned, documents fetched) to execute the query.

```
SELECT * FROM system:completed_requests;
{
   "completed_requests": {
       "ElapsedTime": "1.219329819s",
       "ErrorCount": 0,
       "PhaseCounts": {
           "Fetch": 31591,
           "PrimaryScan": 31591,
           "Sort": 33
       },
       "PhaseOperators": {
           "Fetch": 1,
           "PrimaryScan": 1,
           "Sort": 1
       },
       "RequestId": "cad6fcc0-88cc-4329-ba56-e05e8ef6a53e",
       "ResultCount": 10,
       "ResultSize": 1153,
       "ServiceTime": "1.219299801s",
       "State": "completed",
       "Statement": "SELECT country, id, name FROM `travel-sample` WHERE type = \"airline\" AND
country = \"United States\" AND id BETWEEN 0 AND 1000 ORDER BY id LIMIT 10 OFFSET 5",
       "Time": "2016-12-08 12:00:02.093969361 -0800 PST"
   }
}
```

The travel-sample bucket has the following type of documents.

Type of document	Count

airline	187
airport	1968
route	24024
landmark	4495
hotel	917
total	3591

Primary Index Scan gets all the item keys from the index and the query engine fetches the items, applies the predicate, sorts the results, and then paginates to produces 10 items.

Query engine processed 31591 items to produce 10 items.

Step 2: Using Secondary Index

Let's create a secondary index on type.

```
CREATE INDEX ts_ix1 ON `travel-sample`(type);
EXPLAIN
SELECT country, id, name
FROM `travel-sample`
WHERE type = "airline"
       AND country = "United States"
       AND id BETWEEN 0 AND 1000
ORDER BY id
LIMIT 10
OFFSET 5;
{
.......
  "~children": [
       {
           "#operator": "IndexScan",
           "index": "ts_ix1",
           "index_id": "dd95586d9994d427",
           "keyspace": "travel-sample",
           "namespace": "default",
           "spans": [
               {
                   "Range": {
                       "High": [
                           "\"airline\""
                       ],
                       "Inclusion": 3,
                       "Low": [
                           "\"airline\""
                       ]
                   }
               }
           ],
           "using": "gsi"
       }
   ]
........
}
```

The query uses a secondary index and pushes type predicate down to indexer. This results in 187 items.

id and country predicates are not pushed down to the indexer; they are applied after fetching the items.

Step 3: Using Composite Index on All Attributes in Query Predicates

Let’s drop the index created in the previous step and create a composite index based on type, id, and country.

```
DROP INDEX `travel-sample`.ts_ix1;
CREATE INDEX ts_ix1 ON `travel-sample`(type, id, country);
EXPLAIN
SELECT country, id, name
```

```
FROM `travel-sample`
WHERE type = "airline"
       AND country = "United States"
       AND id BETWEEN 0 AND 1000
ORDER BY id
LIMIT 10
OFFSET 5;
{
........
   "~children": [
       {
           "#operator": "IndexScan",
           "index": "ts_ix1",
           "index_id": "f725e59b0adf5875",
           "keyspace": "travel-sample",
           "namespace": "default",
           "spans": [
               {
                   "Range": {
                       "High": [
                           "\"airline\"",
                           "1000",
                           "\"United States\""
                       ],
                       "Inclusion": 3,
                       "Low": [
                           "\"airline\"",
                           "0",
                           "\"United States\""
                       ]
                   }
               }
           ],
           "using": "gsi"
       }
........
}
```

The query uses a composite index and pushes all the predicates down to indexer. This results in 18+ items.

This index has airport, route, landmark, and hotel items as well. We can restrict the index only to airline items.

Step 4: Using Partial Composite Index

Let's drop the index created in the previous step and create a partial composite index.

```
DROP INDEX `travel-sample`.ts_ix1;
CREATE INDEX ts_ix1 ON `travel-sample`(id, country) WHERE type = "airline";
EXPLAIN
SELECT country, id, name
FROM `travel-sample`
WHERE type = "airline"
       AND country = "United States"
       AND id BETWEEN 0 AND 1000
ORDER BY id
LIMIT 10
OFFSET 5;
{
........
   "~children": [
       {
           "#operator": "IndexScan",
           "index": "ts_ix1",
           "index_id": "276fabb0b7375a64",
           "keyspace": "travel-sample",
           "namespace": "default",
           "spans": [
               {
                   "Range": {
                       "High": [
                           "1000",
                           "\"United States\""
                       ],
                       "Inclusion": 3,
                       "Low": [
                           "0",
                           "\"United States\""
                       ]
                   }
               }
           ],
           "using": "gsi"
       }
........
}
```

This index has only items that have type = "airline."

Index condition (type = "airline") is equality predicate, so it is not required to include it in the index keys.

This makes the index lean and makes it perform better by reducing I/O, memory, CPU, and network.

It pushes all the predicates down to the indexer. This results in 18+ items.

Step 5: Using Covering Partial Composite Index

The query uses type, id, country, and name.

The type is part of the index condition as equality predicate and N1QL can derive the type value from the index condition and answer the query.

The id and country are index keys.

The only missing attribute is a name. Let's add it as trailing index key.

Let’s drop the index created in the previous step and create a covering partial composite index.

```
DROP INDEX `travel-sample`.ts_ix1;
CREATE INDEX ts_ix1 ON `travel-sample`(id, country,name) WHERE type = "airline";
EXPLAIN
SELECT country, id, name
FROM `travel-sample`
```

```
WHERE type = "airline"
      AND country = "United States"
      AND id BETWEEN 0 AND 1000
ORDER BY id
LIMIT 10
OFFSET 5;
{
  "requestID": "8763f612-ab85-4d53-ad83-4c0d5248787b",
  "signature": "json",
  "results": [
    {
      "plan": {
        "#operator": "Sequence",
        "~children": [
          {
            "#operator": "Sequence",
            "~children": [
              {
                "#operator": "IndexScan",
                "covers": [
                  "cover ((`travel-sample`.`id`))",
                  "cover ((`travel-sample`.`country`))",
                  "cover ((`travel-sample`.`name`))",
                  "cover ((meta(`travel-sample`).`id`))"
                ],
                "filter_covers": {
                  "cover ((`travel-sample`.`type`))": "airline"
                },
                "index": "ts_ix1",
                "index_id": "64ecc9c1396eb225",
                "keyspace": "travel-sample",
                "namespace": "default",
                "spans": [
                  {
                    "Range": {
                      "High": [
                        "1000",
                        "successor(\"United States\")"
                      ],
                                        "Inclusion": 1,
                                        "Low": [
                                            "0",
                                            "\"United States\""
                                        ]
                                    }
                                }
                            ],
                            "using": "gsi"
                        },
                        {
                            "#operator": "Parallel",
                            "maxParallelism": 1,
                            "~child": {
                                "#operator": "Sequence",
                                "~children": [
                                    {
                                        "#operator": "Filter",
                                        "condition": "(((cover ((`travel-sample`.`type`)) = \"airline\") and (cover ((`travel-sample`.`country`)) = \"United States\")) and (cover ((`travel-sample`.`id`)) between 0 and 1000))"
                                    },
                                    {
                                        "#operator": "InitialProject",
                                        "result_terms": [
                                            {
                                                "expr": "cover ((`travel-sample`.`country`))"
                                            },
                                            {
                                                "expr": "cover ((`travel-sample`.`id`))"
                                            },
                                            {
                                                "expr": "cover ((`travel-sample`.`name`))"
                                            }
                                        ]
                                    }
                                ]
                            }
                        }
```

```
                ]
              },
              {
                "#operator": "Offset",
                "expr": "5"
              },
              {
                "#operator": "Limit",
                "expr": "10"
              },
              {
                "#operator": "FinalProject"
              }
            ]
          },
          "text": "SELECT country, id, name FROM `travel-sample` WHERE type = \"airline\" AND
country = \"United States\" AND id BETWEEN 0 AND 1000 ORDER BY id LIMIT 10 OFFSET"
        }
      ]
    }
```

The query uses covering partial index and pushes all the predicates down to the indexer. This results in 18+ items.

The index has all the information required by the query.

Query avoids fetch and answers from the index scan.

This query is called a Covered Query and the index is called Covering Index.

Step 6: Query Exploit Index Order

Index stores data pre-sorted by the index keys. When the ORDER BY list matches the INDEX keys list order left to right, the query can exploit index order.

Query ORDER BY expressions match index keys from left to right.

IndexScan has a single span and the query doesn't have any JOINs, GROUP BY, or other clauses that can change the order or granularity of items produced by indexer.

The query can exploit index order and avoid expensive sort and fetch.

The EXPLAIN (step 5) will not have an Order section.

Step 7: OFFSET Pushdown to Indexer

Providing pagination hints to the indexer tells the indexer how many items it can produce and reduce unnecessary work. This allows scaling the workload in the cluster.

Currently, the offset is not pushed down to the indexer separately.

If the LIMIT is present, the Query Engine pushes down limit as limit + offset and applies limit and offset separately.

Step 8: LIMIT Pushdown to Indexer

Providing pagination hints(LIMIT) to indexer. This tells indexer maximum items it should produce. This reduces unnecessary work by indexer and Query Engine, helping to scale.

With the technique used in Step 5, LIMIT is not pushed to indexer. Let's see how we can push the LIMIT to indexer.

LIMIT can be pushed down to the indexer:

- When the query does not have ORDER BY clause OR query uses index order.
- When exact predicates are pushed down to the indexer and the Query Engine will not eliminate any documents (i.e., no false positives from indexer).

In the case of a composite key, if leading keys predicates are non-equality, the indexer will produce false positives.

The id is the range predicate; LIMIT will not be pushed to the indexer. The EXPLAIN (step 5) will not have a “limit” in the IndexScan section.

The query has an equality predicate on "country"; if "country" is the leading index key followed by the id, the indexer will not produce any false positives and allows LIMIT push down to the indexer. However, the query ORDER BY expressions and index keys order are different; this disallows LIMIT push-down and requires sorting.

By closely looking at the query, we have "country" as an equality predicate. If we change the query to include the ORDER BY country, the id query outcome will not change and we can use index order and push the LIMIT down to indexer. In 4.6.0, this is detected automatically and no query changes are required.

Let's drop the index created in the previous step and create a covering partial composite index.

```
DROP INDEX `travel-sample`.ts_ix1;
CREATE INDEX ts_ix1 ON `travel-sample`(country, id, name) WHERE type =
"airline";
EXPLAIN
SELECT country, id, name
FROM `travel-sample`
WHERE type = "airline"
       AND country = "United States"
       AND id BETWEEN 0 AND 1000
ORDER BY country, id
LIMIT 10
OFFSET 5;
{
   "requestID": "922e2b13-4152-4327-be02-49463e150ced",
   "signature": "json",
   "results": [
       {
           "plan": {
               "#operator": "Sequence",
               "~children": [
                   {
                       "#operator": "Sequence",
                       "~children": [
                           {
                               "#operator": "IndexScan",
                               "covers": [
                                   "cover ((`travel-sample`.`country`))",
                                   "cover ((`travel-sample`.`id`))",
                                   "cover ((`travel-sample`.`name`))",
                                   "cover ((meta(`travel-sample`).`id`))"
                               ],
                               "filter_covers": {
                                   "cover ((`travel-sample`.`type`))":
"airline"
                               },
                               "index": "ts_ix1",
                               "index_id": "a51d9f2255cfb89e",
                               "keyspace": "travel-sample",
                               "limit": "(5 + 10)",
                               "namespace": "default",
                               "spans": [
                                   {
                                       "Range": {
                                           "High": [
                                               "\"United States\"",
                                               "successor(1000)"
                                           ],
                                           "Inclusion": 1,
                                           "Low": [
                                               "\"United States\"",
                                               "0"
                                           ]
```

```
                            }
                        }
                    ],
                    "using": "gsi"
                },
                {
                    "#operator": "Parallel",
                    "maxParallelism": 1,
                    "~child": {
                        "#operator": "Sequence",
                        "~children": [
                            {
                                "#operator": "Filter",
                                "condition": "(((cover
((`travel-sample`.`type`)) = \"airline\") and (cover
((`travel-sample`.`country`)) = \"United States\")) and (cover
((`travel-sample`.`id`)) between 0 and 1000))"
                            },
                            {
                                "#operator": "InitialProject",
                                "result_terms": [
                                    {
                                        "expr": "cover
((`travel-sample`.`country`))"
                                    },
                                    {
                                        "expr": "cover
((`travel-sample`.`id`))"
                                    },
                                    {
                                        "expr": "cover
((`travel-sample`.`name`))"
```

```
                                                }
                                            ]
                                        }
                                    ]
                                }
                            }
                        ]
                    },
                    {
                        "#operator": "Offset",
                        "expr": "5"
                    },
                    {
                        "#operator": "Limit",
                        "expr": "10"
                    },
                    {
                        "#operator": "FinalProject"
                    }
                ]
            },
            "text": "SELECT country, id, name FROM `travel-sample` WHERE type = \"airline\" AND country = \"United States\" AND id BETWEEN 0 AND 1000 ORDER BY country, id LIMIT 10 OFFSET 5"
        }
    ]
}
```

The EXPLAIN will have “limit” in the index section(line 38) and will not have order section, The value of the “limit” is the value of limit + offset.

Final Query and Index

```
CREATE INDEX ts_ix1 ON `travel-sample`(country, id, name) WHERE type = "airline";
```

```
SELECT country, id, name
FROM `travel-sample`
WHERE type = "airline"
AND country = "United States"
   AND id BETWEEN 0 AND 1000
ORDER BY country, id
LIMIT 10
OFFSET 5;
```

Summary

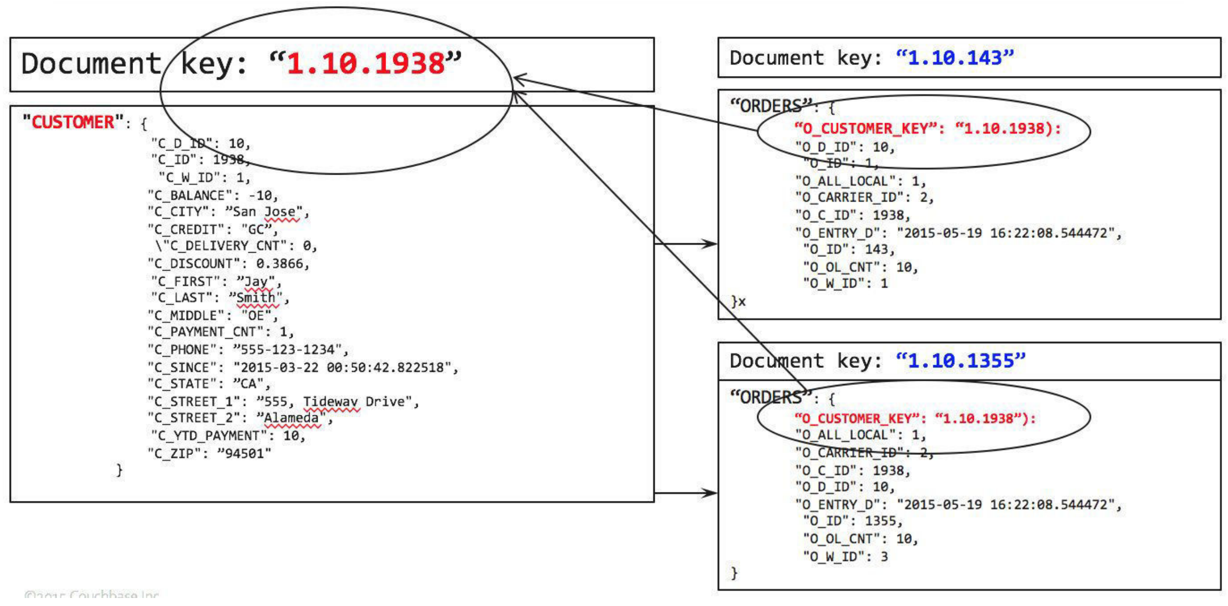
While designing the index, explore all index options. Include as many query predicates as possible in the leading index keys (equality, IN, less than, (less than or equal), greater than, (greater than or equal)), followed by other attributes used in your query. Use partial indexes to focus your queries on the relevant data.

JOIN Faster With Couchbase Index JOINs

Author: Keshav Murthy

Good features in a query language help you to optimize the data model, save space, and increase performance.

Normally, you'd have a child table pointing to its parent. For example, orders have the document key of the customer. So, starting with orders, you join customers to have the fully joined document, which can be processed further.



```
Document key: “1.10.1938”

"CUSTOMER": {
                  "C_D_ID": 10,
                  "C_ID": 1938,
                   "C_W_ID": 1,
                 "C_BALANCE": -10,
                 "C_CITY": ”San Jose",
                 "C_CREDIT": "GC”,
                  \"C_DELIVERY_CNT": 0,
                 "C_DISCOUNT": 0.3866,
                  "C_FIRST": ”Jay",
                  "C_LAST": ”Smith",
                 "C_MIDDLE": "OE",
                 "C_PAYMENT_CNT": 1,
                 "C_PHONE": ”555-123-1234",
                 "C_SINCE": "2015-03-22 00:50:42.822518",
                 "C_STATE": ”CA",
                 "C_STREET_1": ”555, Tidewav Drive",
                 "C_STREET_2": ”Alameda",
                  "C_YTD_PAYMENT": 10,
                 "C_ZIP": ”94501"
          }
```

```
Document key: “1.10.143”

“ORDERS”: {
       “O_CUSTOMER_KEY”: “1.10.1938):
       "O_D_ID": 10,
        "O_ID": 1,
       "O_ALL_LOCAL": 1,
       "O_CARRIER_ID": 2,
       "O_C_ID": 1938,
       "O_ENTRY_D": "2015-05-19 16:22:08.544472",
        "O_ID": 143,
        "O_OL_CNT": 10,
        "O_W_ID": 1
}x
```

```
Document key: “1.10.1355”

“ORDERS”: {
       “O_CUSTOMER_KEY”: “1.10.1938”):
       "O_ALL_LOCAL": 1,
       "O_CARRIER_ID": 2,
       "O_C_ID": 1938,
       "O_D_ID": 10,
       "O_ENTRY_D": "2015-05-19 16:22:08.544472",
        "O_ID": 1355,
        "O_OL_CNT": 10,
        "O_W_ID": 3
}
```

To get the list of orders by zip code, you write the following query:

```
SELECT c.C_ZIP, COUNT(o.O_ID)
FROM ORDERS AS o LEFT OUTER JOIN CUSTOMER AS c
   ON KEYS o.O_CUSTOMER_KEY
```

```
GROUP BY c.C_ZIP
ORDER BY COUNT(1) desc;
```

This works like a charm. Let's look at the query plan.

We use the primary index on ORDERS to do the full scan. For each document there, try to find the matching CUSTOMER document by using the ORDERS.O_CUSTOMER_KEY as the document key. After the JOIN, grouping, aggregation, and sorting follows.

```
[
 {
   "plan": {
     "#operator": "Sequence",
     "~children": [
       {
         "#operator": "Sequence",
         "~children": [
           {
             "#operator": "PrimaryScan",
             "index": "#primary",
             "keyspace": "ORDERS",
             "namespace": "default",
             "using": "gsi"
           },
           {
             "#operator": "Parallel",
             "~child": {
               "#operator": "Sequence",
               "~children": [
                 {
```

```
        "#operator": "Fetch",
        "as": "o",
        "keyspace": "ORDERS",
        "namespace": "default"
      },
      {
        "#operator": "Join",
        "as": "c",
        "keyspace": "CUSTOMER",
        "namespace": "default",
        "on_keys": "(`o`.`O_CUSTOMER_KEY`)",
        "outer": true
      },
      {
        "#operator": "InitialGroup",
        "aggregates": [
          "count((`o`.`O_ID`))",
          "count(1)"
        ],
        "group_keys": [
          "(`c`.`C_ZIP`)"
        ]
      }
    ]
  }
},
{
  "#operator": "IntermediateGroup",
  "aggregates": [
    "count((`o`.`O_ID`))",
    "count(1)"
  ],
  "group_keys": [
    "(`c`.`C_ZIP`)"
  ]
},
{
  "#operator": "FinalGroup",
  "aggregates": [
    "count((`o`.`O_ID`))",
    "count(1)"
  ],
  "group_keys": [
    "(`c`.`C_ZIP`)"
  ]
},
{
  "#operator": "Parallel",
  "~child": {
    "#operator": "Sequence",
    "~children": [
      {
        "#operator": "InitialProject",
        "result_terms": [
          {
            "expr": "(`c`.`C_ZIP`)"
          },
          {
            "expr": "count((`o`.`O_ID`))"
          }
        ]
      }
    ]
  }
```

```
            }
          ]
        },
        {
          "#operator": "Order",
          "sort_terms": [
            {
              "desc": true,
              "expr": "count(1)"
            }
          ]
        },
        {
          "#operator": "FinalProject"
        }
      ]
    },
    "text": "SELECT c.C_ZIP, COUNT(o.O_ID)\nFROM ORDERS AS o LEFT OUTER JOIN
CUSTOMER AS c\n         ON KEYS o.O_CUSTOMER_KEY\nGROUP BY c.C_ZIP\nORDER BY
COUNT(1) desc;"
  }
]
```

But what if you're interested California (CA) residents only? Simply add a predicate on the C_STATE field.

```
SELECT c.C_ZIP, COUNT(o.O_ID)
FROM ORDERS AS o LEFT OUTER JOIN CUSTOMER AS c
   ON KEYS o.O_CUSTOMER_KEY
WHERE c.C_STATE = "CA"
GROUP BY c.C_ZIP
```

```
ORDER BY COUNT(1) desc;
```

This works, except we end up scanning all of the orders, whether the orders belong to California or not. Only after the JOIN operation do we apply the C_STATE = "CA" filter. In a large data set, this has negative performance impact. What if we could improve the performance by limiting the amount of data accessed on the ORDERS bucket.

This is exactly what the index JOINs feature will help you do. The alternate query is below.

```
SELECT c.C_ZIP, COUNT(o.O_ID)
FROM CUSTOMER AS c LEFT OUTER JOIN ORDERS AS o
        ON KEY o.O_CUSTOMER_KEY FOR c
WHERE c.C_STATE = "CA"
GROUP BY c.C_ZIP
ORDER BY COUNT(1) desc;
```

You do need an index on ORDERS.O_CUSTOMER_KEY.

To further improve the performance, you can create the index on CUSTOMER.C_STATE.

```
CREATE INDEX idx_okey ON ORDERS(O_CUSTOMER_KEY);
```

With these indexes, you get a plan like the following:

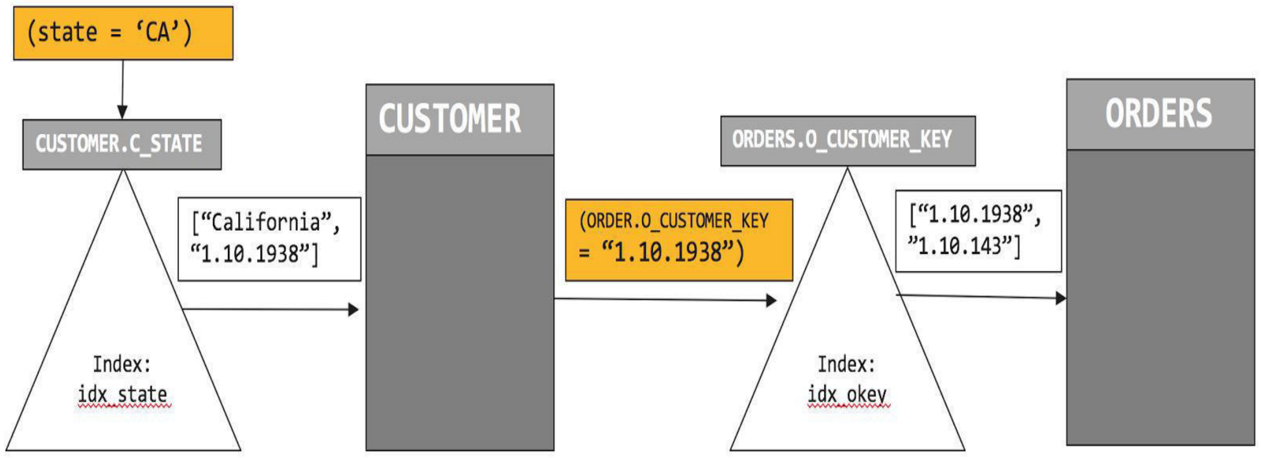
```
CREATE INDEX idx_cstate ON CUSTOMER(C_STATE);
```

Let's examine the explanation. We use two indexes idx_cstate, which scans the CUSTOMER with the predicate (C_STATE = "CA"), and then idx_okey, which helps to find the matching document in ORDERS.

```
[
 {
   "plan": {
     "#operator": "Sequence",
     "~children": [
       {
         "#operator": "Sequence",
         "~children": [
           {
             "#operator": "IndexScan",
             "index": "idx_cstate",
             "index_id": "a3a663ec9928d888",
             "keyspace": "CUSTOMER",
             "namespace": "default",
             "spans": [
               {
                 "Range": {
                   "High": [
                     "\"CA\""
                   ],
                   "Inclusion": 3,
                   "Low": [
                     "\"CA\""
                   ]
                 }
               }
             ],
             "using": "gsi"
           },
           {
             "#operator": "Parallel",
             "~child": {
```

```
"#operator": "Sequence",
"~children": [
  {
    "#operator": "Fetch",
    "as": "c",
    "keyspace": "CUSTOMER",
    "namespace": "default"
  },
  {
    "#operator": "IndexJoin",
    "as": "o",
    "for": "c",
    "keyspace": "ORDERS",
    "namespace": "default",
    "on_key": "(`o`.`O_CUSTOMER_KEY`)",
    "outer": true,
    "scan": {
      "index": "idx_okey",
      "index_id": "271ea96d9390e10d",
      "using": "gsi"
    }
  },
  {
    "#operator": "Filter",
    "condition": "((`c`.`C_STATE`) = \"CA\")"
  },
  {
    "#operator": "InitialGroup",
    "aggregates": [
      "count((`o`.`O_ID`))",
      "count(1)"
    ],
        "group_keys": [
          "(`c`.`C_ZIP`)"
        ]
      }
    ]
  }
},
{
  "#operator": "IntermediateGroup",
  "aggregates": [
    "count((`o`.`O_ID`))",
    "count(1)"
  ],
  "group_keys": [
    "(`c`.`C_ZIP`)"
  ]
},
{
  "#operator": "FinalGroup",
  "aggregates": [
    "count((`o`.`O_ID`))",
    "count(1)"
  ],
  "group_keys": [
    "(`c`.`C_ZIP`)"
  ]
},
{
  "#operator": "Parallel",
  "~child": {
    "#operator": "Sequence",
    "~children": [
              {
                "#operator": "InitialProject",
                "result_terms": [
                  {
                    "expr": "(`c`.`C_ZIP`)"
                  },
                  {
                    "expr": "count((`o`.`O_ID`))"
                  }
                ]
              }
            ]
          }
        }
      ]
    },
    {
      "#operator": "Order",
      "sort_terms": [
        {
          "desc": true,
          "expr": "count(1)"
        }
      ]
    },
    {
      "#operator": "FinalProject"
    }
  ]
},
        "text": "SELECT c.C_ZIP, COUNT(o.O_ID)\nFROM CUSTOMER AS c LEFT OUTER JOIN
    ORDERS AS o\n         ON KEY o.O_CUSTOMER_KEY FOR c\nWHERE c.C_STATE =
    \"CA\"\nGROUP BY c.C_ZIP\nORDER BY COUNT(1) desc;"
     }
    ]
```

So, how does this plan execute? Let's look at the visual version of this.



We first initiate the index scan on CUSTOMER.idx_state and pushdown the filter (c.C_STATE = "CA"). Index scan returns a list of qualified customers. In this case, the CUSTOMER document key is "1.10.1938." We retrieve the CUSTOMER document, then initiate the index scan on ORDERS.idx_okey with the predicate on the CUSTOMER document key (ORDERS.O_CUSTOMER_KEY = "1.10.1938"). That scan returns the document key of the ORDERS, "1.10.143."

Comparing plan 1 with plan 2, the plan to uses two indices to minimize the amount of data to retrieve and process. It, therefore, performs faster.

The index JOIN feature is composable. You can use index JOIN as part of any of JOIN statements to help you navigate through your data model. For example:

```
    SELECT c.C_ZIP, COUNT(o.O_ID), COUNT(ol.OL_ORDER_ITEMS)
```

```
FROM CUSTOMER AS c LEFT OUTER JOIN ORDERS AS o
        ON KEY o.O_CUSTOMER_KEY FOR c
    INNER JOIN ORDER_LINE ol
     ON KEYS o.O_OL_ORDER_KEY
WHERE c.C_STATE = "CA"
GROUP BY c.C_ZIP
ORDER BY COUNT(1) desc;
```

Try it yourself. I've given examples you can try out yourself on Couchbase 4.5 using the beer-sample dataset shipped with it. Check out the slides here.

Summary

Index joins help you to join tables from parent-to-child even when the parent document does not have a reference to its children documents. You can use this feature with INNER JOINS and LEFT OUTER JOINS. This feature is composable. You can have a multi-join statement, with only some of them exploiting index joins.

A Couchbase Index Technique for LIKE Predicates With Wildcard

Author: Tai Tran

I have a query below where I want to find all documents in the travel-sample bucket having the airport name containing the string "Washington."

```
SELECT airportname from `travel-sample`
where
   airportname LIKE "%Washington%"
AND type = "airport"
```

This is a classic problem with most of the RDBMS systems where these systems can't use the index when a LIKE predicate has a leading wildcard i.e. leading %. Elsewhere in this article, I will refer to the query above as the "slow query." I am pleased to tell you that help is on the way with Couchbase release 4.5.1, where this problem can be solved with a combination of the new SUFFIXES() function and Array Index.

Note: To follow along with the examples in this article, please refer to the setup instructions as given here to install the travel-sample Couchbase bucket.

SUFFIXES() is a N1QL function that generates all suffixes of an input string. For example, the N1QL statement below:

```
select SUFFIXES("Baltimore Washington Intl")
```

That returns an array of suffixes:

```
[
    "Baltimore Washington Intl",
    "altimore Washington Intl",
    "ltimore Washington Intl",
    "timore Washington Intl",
    "imore Washington Intl",
    "more Washington Intl",
    "ore Washington Intl",
    "re Washington Intl",
    "e Washington Intl",
    " Washington Intl",
    "Washington Intl",
    "ashington Intl",
    "shington Intl",
    "hington Intl",
    "ington Intl",
    "ngton Intl",
    "gton Intl",
    "ton Intl",
    "on Intl",
    "n Intl",
    " Intl",
    "Intl",
    "ntl",
    "tl",
    "l"
  ]
```

By applying the SUFFIXES() function on the airportname attribute of the documents of type 'airport,' Couchbase generates an array of suffixes for each document. You can then create an array index on the suffixes arrays and rewrite the query to leverage such index. Here is an example:

- Create an array index on the suffixes of the airportname attribute :

```
create index suffixes_airport_name on `travel-sample`(
      DISTINCT ARRAY array_element FOR array_element IN
SUFFIXES(LOWER(airportname)) END)
      WHERE type = "airport";
```

- The "slow query" can be rewritten to use the suffixes_airport_name index and returns all the documents containing the airportname suffix, starting with 'washington%.' It is worth noting that in this re-written query the LIKE predicate no longer needs the leading % wildcard because the matching criterion is on the suffixes.

```
select airportname from `travel-sample`
where
   ANY array_element IN SUFFIXES(LOWER(airportname)) SATISFIES
array_element LIKE 'washington%' END
and  type="airport";
the query returns
[
 {
   "airportname": "Ronald Reagan Washington Natl"
 },
 {
   "airportname": "Washington Dulles Intl"
 },
 {
   "airportname": "Baltimore Washington Intl"
 },
 {
   "airportname": "Washington Union Station"
 }
]
```

The elapsed time for this rewritten query is roughly 25ms when executed on my small VM cluster, and it is 10 times faster compared to the elapsed time of the original "slow query." I am including a portion of the explain plan to confirm that the rewritten query uses the array index.

```
[
  {
        ...
            {
              "#operator": "DistinctScan",
              "scan": {
                "#operator": "IndexScan",
                "index": "suffixes_airport_name",
                "index_id": "6fc9785d59a93892",
                "keyspace": "travel-sample",
                "namespace": "default",
                "spans": [
                  {
                    "Range": {
                      "High": [
                        "\"washingtoo\""
                      ],
                      "Inclusion": 1,
                      "Low": [
                        "\"washington\""
                      ]
                    }
                  }
                ],
                "using": "gsi"
              }
            }
        ......
 ]
```

A common application of such a query is when an end user enters a partial searching string in a UI search field, e.g. searching for an airport name. After the end user has entered a few characters, the logic behind the search field issues the query and displays the matches returned as a drop-down picklist (much like what you see when you enter search terms in Google).

Readers should be aware that the longer the input string, the longer the list of suffixes and, therefore, the index on suffixes can be large. I recommend using this technique only on short attributes, which belong to small Reference Data collections, such as Airport, Hotel, Product Catalog, etc.

If your documents have multi-value attribute(s) such as SKILLS, TAGS as shown in the sample document below, I recommend using the SPLIT() function instead of SUFFIXES().

```
candidate: {
        name: "John Doe",
        skills: "JAVA,C,NodeJS,SQL",
}
```

Below is a simple example of SPLIT(). When it is invoked with an input string of comma-separated words, it will return an array of words.

```
SELECT SPLIT("JAVA,C,NodeJS,SQL", ",")
```

The select returns:

```
[
 {
  "$1": [
   "JAVA",
   "C",
   "NodeJS",
   "SQL"
  ]
 }
]
```

By creating an array index on these arrays of words, your application should be able to support word search. You should try this out yourself to see how it works.

If you have documents with long text attributes such as "Description", TOKENS() function is more suitable for parsing text paragraphs into individual words. The TOKENS() function is not available in our current GA release but will be soon.

Optimization

A Deep Dive Into Couchbase N1QL Query Optimization

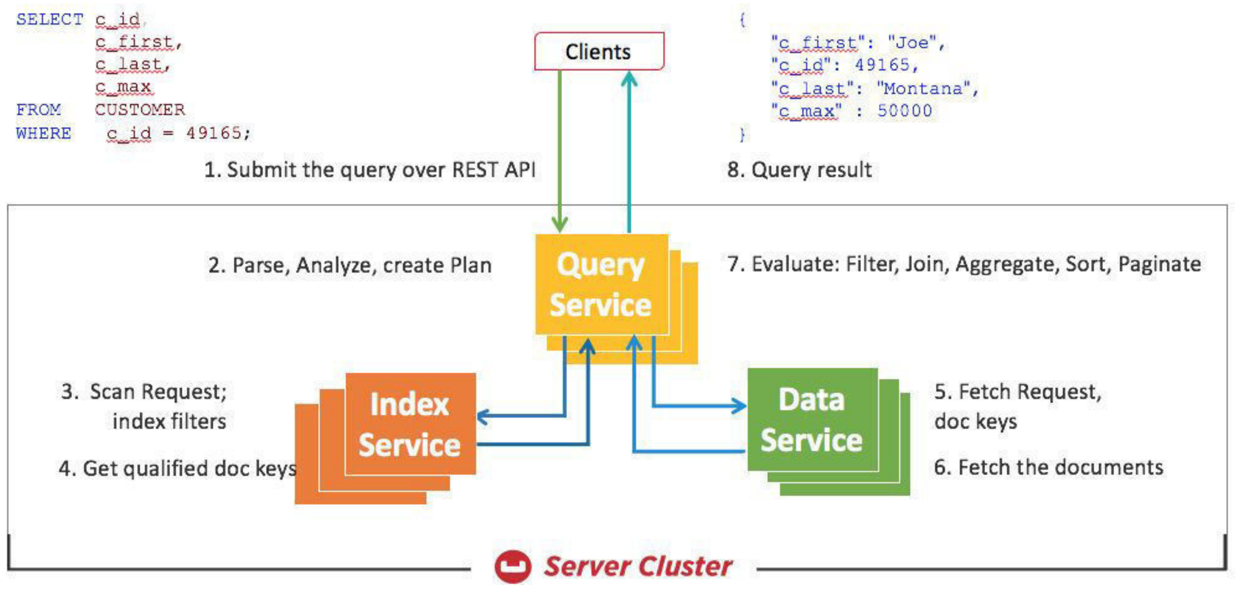
Authors: Keshav Murthy, Sitaram Vemulapalli

SQL is the declarative language for manipulating data in a relational database system. N1QL is the declarative language for JSON data and metadata. Similar to SQL, N1QL has DML statements to manipulate JSON data: SELECT, INSERT, UPDATE, DELETE, MERGE, EXPLAIN. It also introduces a new statement, INFER, which samples the data to describe the schema and show data samples.

Execution of a N1QL query by the engine involves multiple steps. Understanding these will help you to write queries, design for performance, tune query engine efficiently. The N1QL query engine includes parser, semantic analyzer, optimizer and executor. This article will describe how the N1QL query optimizer creates the query plan.

Life of a Query: Overview

Applications and their drivers submit the N1QL query to one of the available query nodes in the Couchbase cluster. The query node analyzes the query, uses metadata on underlying objects to to create the execution plan, executes it. During execution, the query node orchestrates with index and data nodes to perform the select-join-nest-unnest-project operations.


Clients
1. Submit the query over REST API
8. Query result
2. Parse, Analyze, create Plan
Query Service
7. Evaluate: Filter, Join, Aggregate, Sort, Paginate
3. Scan Request; index filters
4. Get qualified doc keys
Index Service
Data Service
5. Fetch Request, doc keys
6. Fetch the documents
Server Cluster

```
SELECT c_id,
       c_first,
       c_last,
       c_max
FROM   CUSTOMER
WHERE   c_id = 49165;
```

```
{
    "c_first": "Joe",
    "c_id": 49165,
    "c_last": "Montana",
    "c_max" : 50000
}
```

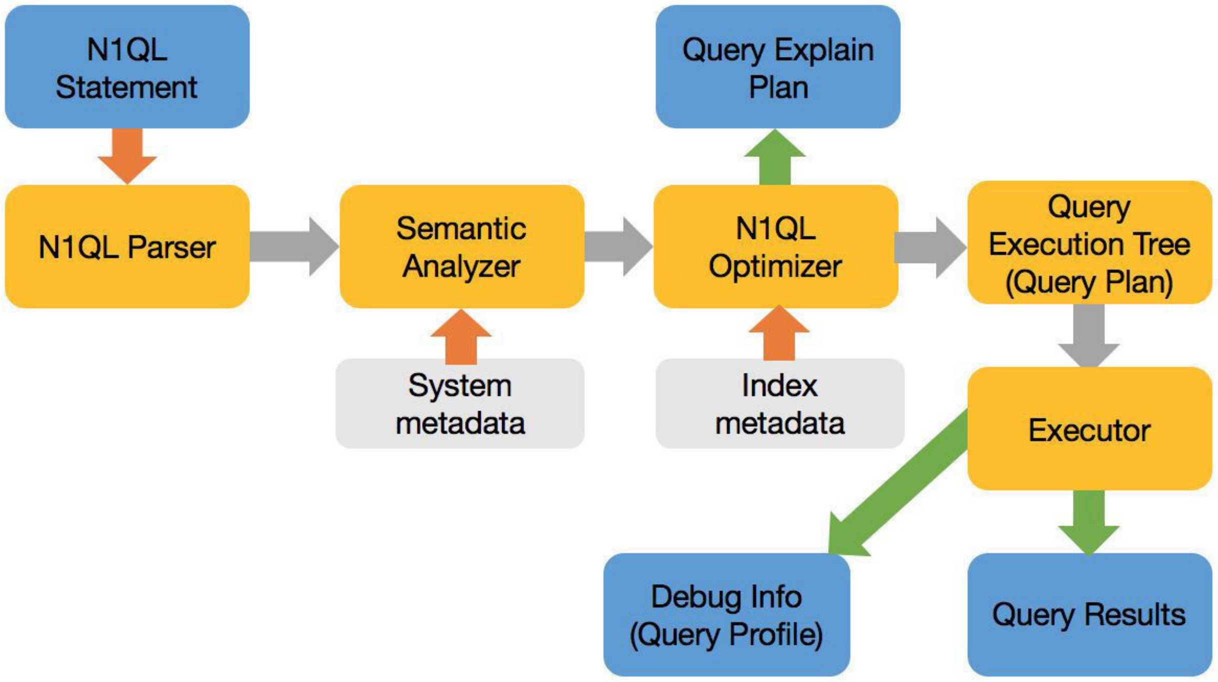
Inside a Query Node

Within each query service, the query flows through the different stages shown below. The N1QL parser checks for syntactic correctness, and the semantic analyzer ensures that the query is meaningful. The N1QL optimizer takes the parsed tree as input, selects suitable indexes, selects access paths for each object and then creates the query plan. It's from this plan that an execution tree with all the necessary operators is created. The execution tree is then executed to manipulate the data, project query results.

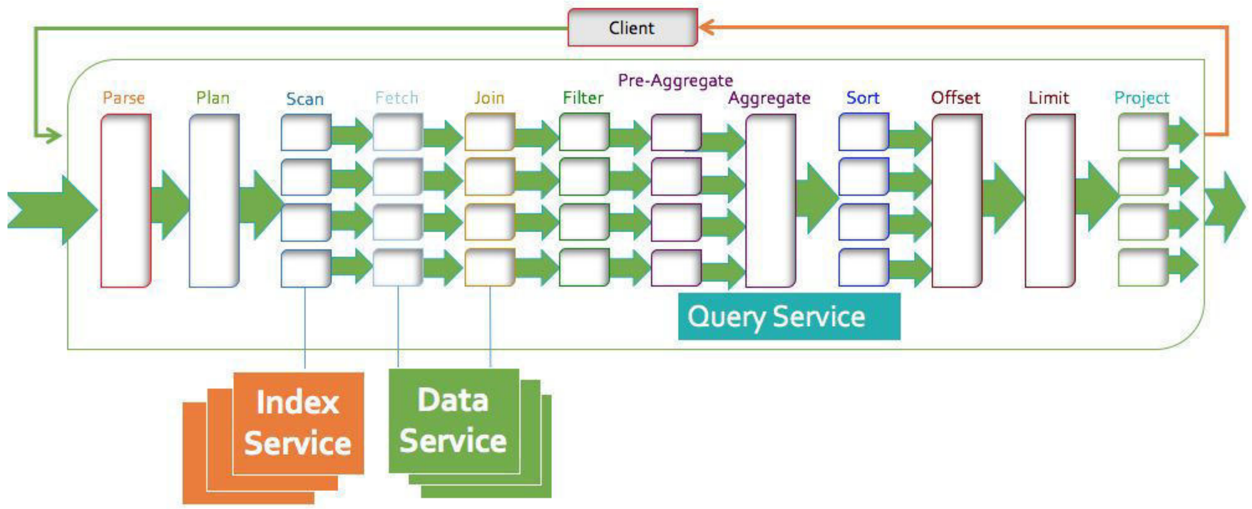
N1QL Query Optimizer

Every flight has a flight plan before it takes off. Each flight has many options for its route. But, depending on guidelines, weather, best practices, and regulations, they decide on a specific flight plan.

Every N1QL query has a query plan. Just like a flight plan, each query can be executed in many ways — different indexes to choose, different predicates to push down, different orders of joins, etc. Eventually, the N1QL query engine will choose an access path in each case and come up with the plan. This plan is converted to query execution tree. This stage of query execution is called optimization.



The figure below shows all the possible phases a query goes through during execution to return the results. Not all queries need to go through every phase. Some go through many of these phases multiple times. Optimizer decides phases each queries executes. For example, the Sort phase will be skipped when there is no ORDER BY clause in the query; the scan-fetch-join phase executes multiple times to perform multiple joins.



Some operations, like query parsing and planning, are done serially — denoted by a single block in the figure. Operations like fetch, join, or sort are done in parallel — denoted by multiple blocks in the figure.

N1QL optimizer analyzes the query and comes up with available access path options for each keyspace in the query. The planner needs to first select the access path for each keyspace, determine the join order and then determine the type of the join. This is done for each query block. Once the plans are done for all the query blocks, the execution is created.

The query below has two keyspaces, airline and airport:

```
SELECT airline, airport
FROM `travel-sample` airline
        INNER JOIN `travel-sample airport
               ON KEYS airline.airport_id
WHERE airline.type = 'airline'
AND airport.type = 'airports'
```

Access Path Selection

Keyspace (Bucket) access path options:

1. **Keyscan access**. When specific document IDs (keys) are available, the Keyscan access method retrieves those documents. Any filters on that keyspace are applied on those documents. The Keyscan can be used when a keyspace is queried by specifying the document keys (USE KEYS modifier) or during join processing. The Keyscan is commonly used to retrieve qualifying documents on for the inner keyspace during join processing.
2. **PrimaryScan access:** This is equivalent of a full table scan in relational database systems. This method is chosen when documents IDs are not given and no qualifying secondary indexes are available for this keyspace. N1QL will retrieve all of document IDs from the primary index, fetch the document and then do the predicate-join-project processing. This access method is quite expensive and the average time to return results increases linearly with number of documents in the bucket. Many customers use primary scan while developing the application, but drop the primary index altogether during production deployment. Couchbase or N1QL does not require the primary index as long as the query has a viable index to access the data.
3. **IndexScan access:** A qualifying secondary index scan is used to first filter the keyspace and to determine the qualifying documents IDs. It then retrieves the qualified documents from the data store, if necessary. If the selected index has all the data to answer the query, N1QL avoids fetching the document altogether — this method is called the **covering index scan**. This is highly performant. Your secondary indexes should help queries choose covering index scans as much as possible.

In Couchbase, the secondary index can be a standard global secondary index using ForestDB or Memory Optimized Index(MOI).

Index Selection

Here is the approach to select the secondary indexes for the query. The N1QL planner determines qualified indexes on the keyspace based on query predicates. The following is the algorithm to select the indexes for a given query.

- **Online indexes**: Only online indexes are selected. That means when the indexes are being built (pending) or only defined, but not built, they aren't chosen.
- **Preferred indexes**: When a statement has the USE INDEX clause, it provides the list of indices to consider. In this case, only those indices are evaluated.
- **Satisfying index condition:** Partial indexes (when the index creation includes the WHERE clause) with a condition that is a super set of the query predicate are selected
- **Satisfying index keys:** Indexes whose leading keys satisfy query predicate are selected. This is the common way to select indexes with B-TREE indexes.
- **Longest satisfying index keys:** Redundancy is eliminated by keeping longest satisfying index keys in the index key order. For example, Index with satisfying keys (a, b, c) is retained over index with satisfying (a, b).

Once the index selection is done the following scan methods are considered in the order.

1. **IndexCountScan**
    - Queries with a single projection of COUNT aggregate, NO JOINs, or GROUP BY are considered. The chosen index needs to be covered with a single exact range for the given predicate, and the argument to COUNT needs to be constant or leading key.
2. **Covering secondary scan**
    - Each satisfied index with the most number of index keys is examined for query coverage, and the shortest covering index will be used. For an index to cover the query, we should be able to run the complete query just using the data in the index. In other words, the index needs to have both keys in the predicate as well as the keys referenced in other clauses, e.g., projection, subquery, order by, etc.
3. **Regular secondary scan**

    - Indexes with the most number of matching index keys are used. When more than one index is qualified, IntersectScan is used. To avoid IntersectScan, provide a hint with USE INDEX.
4. **UNNEST Scan**
    - Only array indexes with an index key matching the predicates are used for UNNEST scan.
5. **Regular primary scan**
    - If a primary scan is selected, and no primary index available, the query errors out.

JOIN Methods

N1QL supports nested loop access method for all the join supports: INNER JOIN and LEFT OUTER JOIN and the index join method.

Here is the simplest explanation of the join method.

For this join, ORDERS become the outer keyspace and CUSTOMER becomes the inner keyspace. The ORDERS keyspace is scanned first (using one of the keyspace scan options). For each qualifying document on ORDERS, we do a KEYSCAN on CUSTOMER based on the key O_C_D in the ORDERS document.

```
SELECT *
FROM ORDERS o INNER JOIN CUSTOMER c ON KEYS o.O_C_ID;
```

Index Join

Index joins help you to join tables from parent-to-child even when the parent document does not have a reference to its children documents. You can use this feature with INNER JOINS and LEFT OUTER JOINS. This feature is composable. You can have a multi-join statement, with only some of them exploiting index joins.

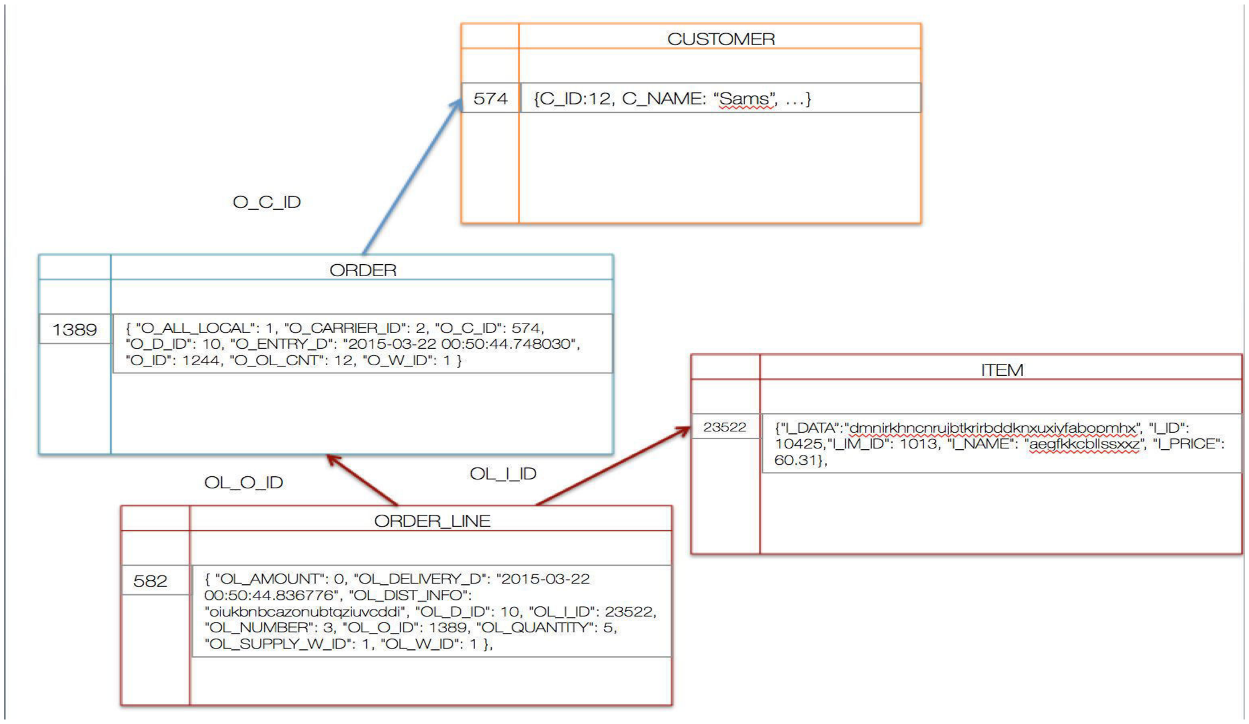
```
SELECT c.C_ZIP, COUNT(o.O_ID)
FROM CUSTOMER AS c LEFT OUTER JOIN ORDERS AS o
        ON KEY o.O_CUSTOMER_KEY FOR c
WHERE c.C_STATE = "CA"
GROUP BY c.C_ZIP
ORDER BY COUNT(1) desc;
```

Read more information on index joins here.

JOIN Order

The keyspaces specified in the FROM clause are joined in the exact order given in the query. N1QL does not change the join order

We use the following keyspaces in our examples. CUSTOMER, ORDER, ITEM, and ORDERLINE. Example documents for these are given at the end of this blog. Those familiar with TPCC will recognize these tables. The figure below illustrates the relationship between these documents.



```
574    {C_ID:12, C_NAME: "Sams", …}

1389   { "O_ALL_LOCAL": 1, "O_CARRIER_ID": 2, "O_C_ID": 574,
       "O_D_ID": 10, "O_ENTRY_D": "2015-03-22 00:50:44.748030",
       "O_ID": 1244, "O_OL_CNT": 12, "O_W_ID": 1 }

23522  {"I_DATA":"dmnirkhncnrujbtkrirbddknxuxiyfabopmhx", "I_ID":
       10425,"I_IM_ID": 1013, "I_NAME": "aegfkkcbllssxxz", "I_PRICE":
       60.31},

582    { "OL_AMOUNT": 0, "OL_DELIVERY_D": "2015-03-22
       00:50:44.836776", "OL_DIST_INFO":
       "oiukbnbcazonubtqziuvcddi", "OL_D_ID": 10, "OL_I_ID": 23522,
       "OL_NUMBER": 3, "OL_O_ID": 1389, "OL_QUANTITY": 5,
       "OL_SUPPLY_W_ID": 1, "OL_W_ID": 1 },
```

Each of these keyspaces has a primary key index and following secondary indices.

```
create index CU_ID_D_ID_W_ID on CUSTOMER(C_ID, C_D_ID, C_W_ID) using gsi;
create index ST_W_ID,I_ID on STOCK(S_I_ID, S_W_ID) using gsi;
create index OR_O_ID_D_ID_W_ID on ORDERS(O_ID, O_D_ID, O_W_ID, O_C_ID) using gsi;
create index OL_O_ID_D_ID_W_ID on ORDER_LINE(OL_O_ID, OL_D_ID, OL_W_ID) using gsi;
create index IT_ID on ITEM(I_ID) using gsi;
```

Example 1

If you know the document keys, specify with the USE KEYS clause for each keyspace. When a USE KEYS clause is specified, the KEYSCAN access path is chosen. Given the keys, KEYSCAN will retrieve the documents from the respective nodes more efficiently. After retrieving the specific documents, the query node applies the filter c.C_STATE = "CA".

```
cbq> EXPLAIN select * from CUSTOMER c
             USE KEYS ["110192", "120143", "827482"]
     WHERE c.C_STATE = "CA";
{
   "requestID": "991e69d2-b6f9-42a1-9bd1-26a5468b0b5f",
   "signature": "json",
   "results": [
       {
           "#operator": "Sequence",
           "~children": [
               {
                   "#operator": "KeyScan",
                   "keys": "[\"110192\", \"120143\", \"827482\"]"
               },
               {
                   "#operator": "Parallel",
                   "~child": {
                       "#operator": "Sequence",
                       "~children": [
                           {
                               "#operator": "Fetch",
                               "as": "c",
                               "keyspace": "CUSTOMER",
                               "namespace": "default"
                           },
                           {
                               "#operator": "Filter",
                               "condition": "((`c`.`C_STATE`) = \"CA\")"
                           },
                           {
                               "#operator": "InitialProject",
                               "result_terms": [
                                   {
                                       "star": true
                                   }
                               ]
                           },
...
```

Example 2

In this case, the query is looking to count all of all customers with (c.C_YTD_PAYMENT < 100). Since we don't have an index on key-value, c.C_YTD_PAYMENT, a primary scan of the keyspace (bucket) is chosen. Filter (c.C_YTD_PAYMENT < 100) is applied after the document is retrieved. Obviously, for larger buckets primary scan takes time. As part of planning for application performance, create relevant secondary indices on frequently used key-values within the filters.

N1QL parallelizes many of the phases within the query execution plan. For this query, fetch and filter applications are parallelized within the query execution.

```
cbq> EXPLAIN SELECT c.C_STATE AS state, COUNT(*) AS st_count
            FROM CUSTOMER c
            WHERE c.C_YTD_PAYMENT < 100
            GROUP BY state
            ORDER BY st_count desc;
   "results": [
       {
           "#operator": "Sequence",
           "~children": [
               {
                   "#operator": "Sequence",
               "~children": [
                       {
                           "#operator": "PrimaryScan",
                           "index": "#primary",
                           "keyspace": "CUSTOMER",
                "namespace": "default",
                "using": "gsi"
            },
                {
                "#operator": "Parallel",
                "~child": {
                    "#operator": "Sequence",
                    "~children": [
                        {
                            "#operator": "Fetch",
                            "as": "c",
                            "keyspace": "CUSTOMER",
                            "namespace": "default"
                        },
                        {
                            "#operator": "Filter",
                            "condition": "((`c`.`C_YTD_PAYMENT`)
\u003c 100)"
                        },
                        {
                            "#operator": "InitialGroup",
                            "aggregates": [
                                "count(*)"
                            ],
                            "group_keys": [
                                "(`c`.`state`)"
                            ]
                        },
                        {
                            "#operator": "IntermediateGroup",
                            "aggregates": [
                                "count(*)"
                            ],
                "group_keys": [
                    "(`c`.`state`)"
                ]
            }
        ]
    }
},
{
    "#operator": "IntermediateGroup",
    "aggregates": [
        "count(*)"
    ],
    "group_keys": [
        "(`c`.`state`)"
    ]
},
{
    "#operator": "FinalGroup",
    "aggregates": [
        "count(*)"
    ],
    "group_keys": [
        "(`c`.`state`)"
    ]
},
{
    "#operator": "Parallel",
    "~child": {
        "#operator": "Sequence",
        "~children": [
            {
                "#operator": "InitialProject",
                "result_terms": [
                                        {
                                            "as": "state",
                                            "expr": "(`c`.`C_STATE`)"
                                        },
                                        {
                                            "as": "st_count",
                                            "expr": "count(*)"
                                        }
                                    ]
                                }
                            ]
                        }
                    }
                ]
            },
            {
                "#operator": "Order",
                "sort_terms": [
                    {
                        "desc": true,
                        "expr": "`st_count`"
                    }
                ]
            },
            {
                "#operator": "Parallel",
                "~child": {
                    "#operator": "FinalProject"
                }
            }
        ]
    }
],
```

Example 3

In this example, we join keyspace ORDER_LINE with ITEM. For each qualifying document in ORDER_LINE, we want to match with ITEM. The ON clause is interesting. Here, you only specify the keys for the key space ORDER_LINE (TO_STRING(ol.OL_I_ID)) and nothing for ITEM. That's because it's implicitly joined with the document key of the ITEM.

N1QL's FROM clause:

```
SELECT ol.*, i.*
 FROM ORDER_LINE ol INNER JOIN ITEM i
       ON KEYS (TO_STRING(ol.OL_I_ID))
```

Is equivalent to SQL's:

In N1QL, META(ITEM).id is the document key of the particular document in ITEM.

```
SELECT ol.*, i.*
       FROM ORDER_LINE ol INNER JOIN ITEM i
        ON (TO_STRING(ol.OL_I_ID) = meta(ITEM).id)
```

If the field is not a string, it can be converted to a string using TO_STRING() expression. You can also construct the document key using multiple fields with the document.

```
SELECT *
FROM ORDERS o LEFT OUTER JOIN CUSTOMER c
   ON KEYS (TO_STRING(o.O_C_ID) || TO_STRING(o.O_D_ID))
```

To summarize, while writing JOIN queries in N1QL, it's important to understand how the document key is constructed on the keyspace. It's important to think about these during data modeling.

First, to scan the ORDER_LINE keyspace, for the given set of filters, the planner chooses the index scan on the index OL_O_ID_D_ID_W_ID. As we discussed before, the access path on the other keyspace in the join is always keyscan using the primary key index. In this plan, we first do the index scan on the ORDER_LINE keyspace pushing down the possible filters to the index scan. Then, we retrieve the qualifying document and apply additional filters. If the document qualifies, that document is then joined with ITEM.

```
cbq> EXPLAIN SELECT COUNT(DISTINCT(ol.OL_I_ID)) AS CNT_OL_I_ID
    FROM ORDER_LINE ol INNER JOIN ITEM i
                  ON KEYS (TO_STRING(ol.OL_I_ID))
    WHERE ol.OL_W_ID = 1
       AND ol.OL_D_ID =  10
       AND ol.OL_O_ID < 200
       AND ol.OL_O_ID >= 100
       AND ol.S_W_ID = 1
       AND i.I_PRICE < 10.00;
{
   "requestID": "4e0822fb-0317-48a0-904b-74c607f77b2f",
   "signature": "json",
   "results": [
       {
           "#operator": "Sequence",
           "~children": [
               {
                   "#operator": "IndexScan",
                   "index": "OL_O_ID_D_ID_W_ID",
                   "keyspace": "ORDER_LINE",
                   "limit": 9.223372036854776e+18,
                   "namespace": "default",
                   "spans": [
                       {
                           "Range": {
                               "High": [
                                        "200"
                                    ],
                                    "Inclusion": 1,
                                    "Low": [
                                        "100"
                                    ]
                                },
                                "Seek": null
                            }
                        ],
                        "using": "gsi"
                    },
                    {
                        "#operator": "Parallel",
                        "~child": {
                            "#operator": "Sequence",
                            "~children": [
                                {
                                    "#operator": "Fetch",
                                    "as": "ol",
                                    "keyspace": "ORDER_LINE",
                                    "namespace": "default"
                                },
                                {
                                    "#operator": "Join",
                                    "as": "i",
                                    "keyspace": "ITEM",
                                    "namespace": "default",
                                    "on_keys": "to_string((`ol`.`OL_I_ID`))"
                                },
                                {
                                    "#operator": "Filter",
                                    "condition": "(((((((`ol`.`OL_W_ID`) = 1) and
((`ol`.`OL_D_ID`) = 10)) and ((`ol`.`OL_O_ID`) \u003c 200)) and (100 \u003c=
(`ol`.`OL_O_ID`))) and ((`ol`.`S_W_ID`) = 1)) and ((`i`.`I_PRICE`) \u003c 10))"
                                },
                                {
                                    "#operator": "InitialGroup",
                                    "aggregates": [
                                        "count(distinct (`ol`.`OL_I_ID`))"
                                    ],
                                    "group_keys": []
                                },
                                {
                                    "#operator": "IntermediateGroup",
                                    "aggregates": [
                                        "count(distinct (`ol`.`OL_I_ID`))"
                                    ],
                                    "group_keys": []
                            }
                        ]
                    }
                },
                {
                    "#operator": "IntermediateGroup",
                    "aggregates": [
                        "count(distinct (`ol`.`OL_I_ID`))"
                    ],
                    "group_keys": []
                },
                {
                    "#operator": "FinalGroup",
                    "aggregates": [
                        "count(distinct (`ol`.`OL_I_ID`))"
                    ],
                    "group_keys": []
                },
                {
                    "#operator": "Parallel",
                    "~child": {
                        "#operator": "Sequence",
                        "~children": [
                            {
                                "#operator": "InitialProject",
                                "result_terms": [
                                    {
                                        "as": "CNT_OL_I_ID",
                                        "expr": "count(distinct
(`ol`.`OL_I_ID`))"
                                    }
                                ]
                            },
                            {
                                "#operator": "FinalProject"
                            }
                        ]
                    }
                }
            ]
        }
    ],
    "status": "success",
    "metrics": {
        "elapsedTime": "272.823508ms",
        "executionTime": "272.71231ms",
        "resultCount": 1,
        "resultSize": 4047
    }
    }
```

Example Documents

Data is generated from modified scripts from here.

```
CUSTOMER:

select meta(CUSTOMER).id as PKID, * from CUSTOMER limit 1;
   "results": [
       {
           "CUSTOMER": {
               "C_BALANCE": -10,
               "C_CITY": "ttzotwmuivhof",
               "C_CREDIT": "GC",
               "C_CREDIT_LIM": 50000,
               "C_DATA":
"sjlhfnvosawyjedregoctclndqzioadurtnlslwvuyjeowzedlvypsudcuerdzvdpsvjfecouyavny
yemivgrcyxxjsjcmkejvekzetxryhxjlhzkzajiaijammtyioheqfgtbhekdisjypxoymfsaepqkzbi
tdrpsjppivjatcwxxipjnloeqdswmogstqvkxlzjnffikuexjjofvhxdzleymajmifgzzdbdfvpwuhl
ujvycwlsgfdfodhfwiepafifbippyonhtahsbigieznbjrmvnjxphzfjuedxuklntghfckfljijfeyz
nxvwhfvnuhsecqxcmnivfpnawvgjjizdkaewdidhw",
               "C_DELIVERY_CNT": 0,
               "C_DISCOUNT": 0.3866,
               "C_D_ID": 10,
               "C_FIRST": "ujmduarngl",
               "C_ID": 1938,
               "C_LAST": "PRESEINGBAR",
               "C_MIDDLE": "OE",
               "C_PAYMENT_CNT": 1,
               "C_PHONE": "6347232262068241",
               "C_SINCE": "2015-03-22 00:50:42.822518",
               "C_STATE": "ta",
               "C_STREET_1": "deilobyrnukri",
               "C_STREET_2": "goziejuaqbbwe",
               "C_W_ID": 1,
               "C_YTD_PAYMENT": 10,
               "C_ZIP": "316011111"
           },
           "PKID": "1101938"
       }
   ],
```

ITEM:

```
select meta(ITEM).id as PKID, * from ITEM limit 1;
   "results": [
       {
           "ITEM": {
               "I_DATA": "dmnjrkhncnrujbtkrirbddknxuxiyfabopmhx",
               "I_ID": 10425,
               "I_IM_ID": 1013,
               "I_NAME": "aegfkkcbllssxxz",
               "I_PRICE": 60.31
           },
           "PKID": "10425"
       }
   ],
```

ORDERS:

```
select meta(ORDERS).id as PKID, * from ORDERS limit 1;
   "results": [
       {
           "ORDERS": {
               "O_ALL_LOCAL": 1,
               "O_CARRIER_ID": 2,
               "O_C_ID": 574,
               "O_D_ID": 10,
               "O_ENTRY_D": "2015-03-22 00:50:44.748030",
               "O_ID": 1244,
               "O_OL_CNT": 12,
               "O_W_ID": 1
           },
           "PKID": "1101244"
       }
   ],
```

ORDER_LINE:

```
cbq> select meta(ORDER_LINE).id as PKID, * from ORDER_LINE limit 1;
"results": [
       {
           "ORDER_LINE": {
               "OL_AMOUNT": 0,
               "OL_DELIVERY_D": "2015-03-22 00:50:44.836776",
               "OL_DIST_INFO": "oiukbnbcazonubtqziuvcddi",
               "OL_D_ID": 10,
               "OL_I_ID": 23522,
               "OL_NUMBER": 3,
                "OL_O_ID": 1389,
                "OL_QUANTITY": 5,
                "OL_SUPPLY_W_ID": 1,
                "OL_W_ID": 1
            },
            "PKID": "11013893"
        }
    ],
```

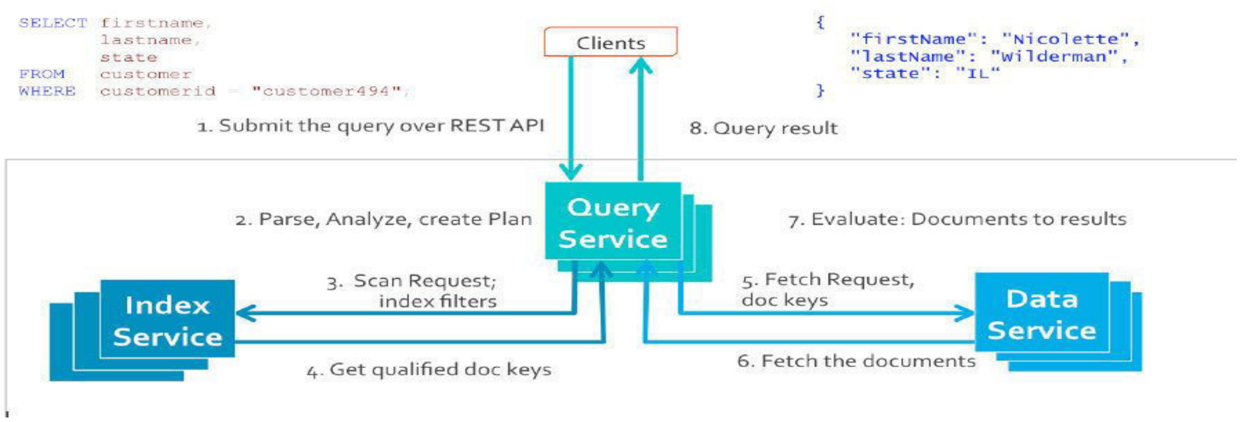
Understanding Index Scans in Couchbase N1QL Query

Author: Sitaram Vemulapalli

Couchbase N1QL is a modern query processing engine designed to provide SQL for JSON on distributed data with a flexible data model. Modern databases are deployed on massive clusters. Using JSON provides a flexible data mode. N1QL supports enhanced SQL for JSON to make query processing easier.

Applications and database drivers submit the N1QL query to one of the available Query nodes on a cluster. The Query node analyzes the query, uses metadata on underlying objects to figure out the optimal execution plan, which it then executes. During execution, depending on the query, using applicable indexes, query node works with index and data nodes to retrieve and perform the planned operations. Because Couchbase is a modular clustered database, you scale out data, index, and query services to fit your performance and availability goals.

Query Execution: Inside View

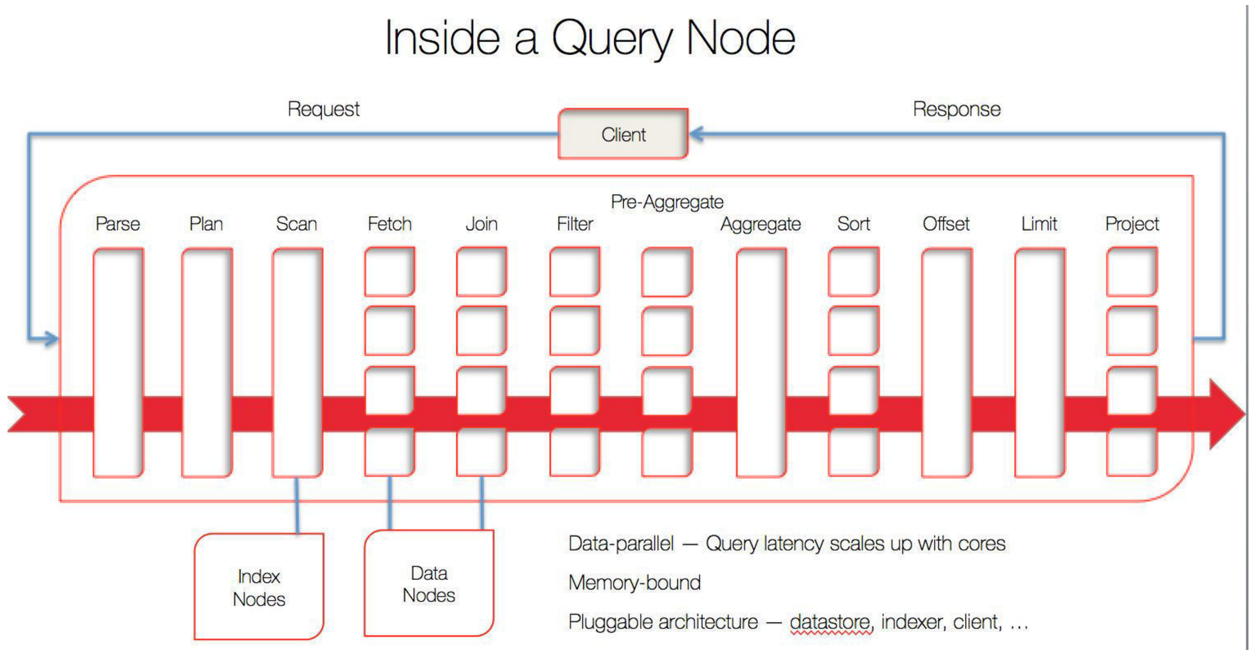


```
SELECT firstname,
       lastname,
       state
FROM   customer
WHERE  customerid = "customer494";
```

```
{
    "firstName": "Nicolette",
    "lastName": "Wilderman",
    "state": "IL"
}
```

Clients
1. Submit the query over REST API
8. Query result
2. Parse, Analyze, create Plan
Query Service
7. Evaluate: Documents to results
3. Scan Request; index filters
5. Fetch Request, doc keys
Index Service
Data Service
4. Get qualified doc keys
6. Fetch the documents

This figure shows all the possible phases a SELECT query goes through to return the results. Not all queries need to go through every phase, some go through many of these phases multiple times. For example, sort phase can be skipped when there is no ORDER BY clause in the query; scan-fetch-join phase will execute multiple times for correlated subqueries.



This brief introduction to query planning has details of query planner. When the Index path is chosen, query engine requests the scan by providing the range of values to return. This range is represented as a SPAN in the query plan. The index spans will play major roles in optimal plan generation and execution. In this article, we discuss how the Index spans are generated from the query predicates (filters).

Spans Overview

FILTER, JOIN, and PROJECT are fundamental operations of database query processing. The filtering process takes the initial keyspace and produces an

optimal subset of the documents the query is interested in. To produce the smallest possible subset, indexes are used to apply as many predicates as possible.
Query predicate indicates the subset of the data interested. During the query planning phase, we select the indexes to be used. Then, for each index, we decide the predicates to be applied by each index. The query predicates are translated into spans in the query plan and passed to Indexer. Spans simply express the predicates in terms of data ranges.

Example Translations

Predicate	Span Low	Span High	Span Inclusion
id = 10	10	10	3 (BOTH)
id > 10	10	no upper bound	0 (NEITHER)
id <= 10	null	10	2 (HIGH)

Consider the plan for the query below:

```
EXPLAIN select meta().id from `travel-sample` where id = 10;
```

The spans can be seen as part of IndexScan section of the explain for the query:

```
{
    "requestID": "4f64d56a-0db6-4d1e-8868-36bfe11146cf",
    "signature": "json",
    "results": [
        {
            "plan": {
                "#operator": "Sequence",
                "~children": [
                    {
                        "#operator": "IndexScan",
                        "covers": [
                            "cover ((`travel-sample`.`id`))",
                            "cover ((meta(`travel-sample`).`id`))"
                        ],
                        "index": "idx_id",
                        "index_id": "8ad897f8afb165ef",
                        "keyspace": "travel-sample",
                        "namespace": "default",
                        "spans": [
                            {
                                "Range": {
                                    "High": [
                                        "10"
                                    ],
                                    "Inclusion": 3,
                                    "Low": [
                                        "10"
                                    ]
                                }
                            }
                        ],
                        "using": "gsi"
                    }
                ]
                ……...
            }
        }
    ]
 }
```

In N1QL, index scan requests are based on range. So, the index spans consists of one or more ranges. Each range will have start value, end value and whether to include the start or end value.

- "High" field of Range indicates the end value. If High is missing, then there is no upper bound.
- "Low" field of Range indicates start value. If Low is missing, the scan starts with null (i.e. null is lowest value, because MISSING values are not indexed).
- Inclusion indicates values of Low or High included

Inclusion		Meaning
0	NEITHER	Neither Low nor High included
1	LOW	Low included
2	HIGH	High included

3	BOTH	Both Low and High included

To run the examples Install travel-sample sample bucket and create the following indexes.

```
CREATE INDEX `idx_id` ON `travel-sample`(`id`);
CREATE INDEX `idx_name` ON `travel-sample`(`name`);
CREATE INDEX `idx_saiport_dairport_stops`  ON `travel-sample` (`sourceairport`,
`destinationairport`, `stops`);
CREATE INDEX idx_sched ON `travel-sample`(DISTINCT ARRAY v.day FOR v IN
schedule END) WHERE type = "route";
```

Example 1: EQUALITY Predicate

```
SELECT meta().id FROM `travel-sample` WHERE id = 10;
```

In this example, the predicate id = 10 pushed to index scan.

Span for	Low	High	Inclusion
id = 10	10	10	3 (BOTH)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE id = 10;
```

```
    "spans": [
        {
            "Range": {
                "High": [
                     "10"
                 ],
                 "Inclusion": 3,
                 "Low": [
                     "10"
                 ]
            }
        }
    ]
```

Example 2: Inclusive One-Sided Range Predicate

```
SELECT meta().id FROM `travel-sample` WHERE id >= 10;
```

In this example, the predicate id >= 10 pushed to index scan.

Span for	Low	High	Inclusion
id >= 10	10	unbounded	1 (LOW)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE id >= 10;
```

```
    "spans": [
        {
            "Range": {
                "Inclusion": 1,
                "Low": [
                    "10"
                ]
            }
        }
    ]
```

Example 3: Exclusive One-Sided Range Predicate

```
SELECT meta().id FROM `travel-sample` WHERE id > 10;
```

In this example, the predicate id > 10 pushed to index scan.

Span for	Low	High	Inclusion
id > 10	10	unbounded	0 (NEITHER)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE id > 10;
```

```
    "spans": [
        {
            "Range": {
                "Inclusion": 0,
                "Low": [
                    "10"
                ]
            }
        }
    ]
```

Example 4: Inclusive One-sided Range Predicate

```
SELECT meta().id FROM `travel-sample` WHERE id <= 10;
```

In this example, the predicate id <= 10 pushed to index scan. The query predicate doesn't contain an explicit start value, when the field is present in predicate it is implicit non-null values. So it is implicit the start value will be non-inclusive null value.

Span for	Low	High	Inclusion
id <= 10	null	10	2 (HIGH)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE id <= 10;

    "spans": [
        {
            "Range": {
                "High": [
                    "10"
                ],
                "Inclusion": 2,
                "Low": [
                    "null"
                ]
            }
        }
    ]
```

Example 5: Exclusive One-Sided Range Predicate

```
SELECT meta().id FROM `travel-sample` WHERE id < 10;
```

In this example, the predicate id < 10 pushed to index scan. The query predicate doesn't contain explicit start value, when the field is present in predicate it is implicit non-null values. So it is implicit the start value will be non-inclusive null value.

Span for	Low	High	Inclusion
id < 10	null	10	0 (NEITHER)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE id < 10;

    "spans": [
        {
            "Range": {
                "High": [
                    "10"
                ],
                "Inclusion": 0,
                "Low": [
                    "null"
                ]
            }
        }
    ]
```

Example 6: AND Predicate

```
SELECT meta().id FROM `travel-sample` WHERE id >= 10 AND id < 25;
```

In this example, the predicate id >= 10 AND id < 25 pushed to index scan.

Span for	Low	High	Inclusion
id >= 10 AND id < 25	10	25	1 (LOW)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE id >=10 AND id < 25;

    "spans": [
        {
            "Range": {
                "High": [
                    "25"
                ],
                "Inclusion": 1,
                "Low": [
                    "10"
                ]
            }
        }
    ]
```

Example 7: Multiple AND Predicates

```
SELECT meta().id FROM `travel-sample` WHERE id >= 10 AND id < 25 AND id <= 20;
```

In this example, the predicate id >= 10 AND id < 25 AND id <= 20 pushed to index scan.

Span for	Low	High	Inclusion
id >= 10 AND id < 25 AND id <= 20	10	20	3 (BOTH)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE id >=10 AND id < 25 AND id
<= 20;

    "spans": [
        {
            "Range": {
                "High": [
                    "20"
                ],
                "Inclusion": 3,
                "Low": [
                    "10"
                ]
            }
        }
    ]
```

Observe that optimizer created the span without the (id < 25) because there is an AND predicate (id <= 20) makes the other predicate redundant. Internally, optimizer breaks down each predicate and then combines it in a logically consistent manner. FYI: If this is too detailed for now, you can skip over to Example 8.

Span for	Low	High	Inclusion
id >= 10	10	unbounded	1 (LOW)
id < 25	null	25	0 (NEITHER)
id >= 10 AND id < 25	10	25	1 (LOW)

`id <= 20`	`null`	`20`	`2 (HIGH)`
`id >= 10 AND id < 25 AND id <= 20`	`10`	`20`	`3 (BOTH)`

1. Combined Low becomes highest of both Low's (null is lowest).
2. Combined High becomes lowest of both High's (unbounded is highest).
3. Combined Inclusion becomes OR of corresponding inclusions of step 1 and 2.
4. Repeat the steps 1-3 for each AND clause.

Example 8: AND Predicate Makes Empty

```
SELECT meta().id FROM `travel-sample` WHERE id > 10 AND id < 5;
```

In this example, the predicate id > 10 AND id < 5 pushed to index scan.

Span for	Low	High	Inclusion
`id > 10 AND id < 5`	`null`	`null`	`0 (NEITHER)`

This is a special case. Span will be Low: 10, High:5, Inclusion:0. In this case, the start value is higher than the end value and will not produce results, so the span is converted to EMPTY SPAN.

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE id >10 AND id < 5;

   "spans": [
         {
             "Range": {
                 "High": [
                      "null"
                  ],
                  "Inclusion": 0
             }
         }
     ]
```

Example 9: BETWEEN Predicate

```
SELECT meta().id FROM `travel-sample` WHERE id BETWEEN 10 AND 25;
```

In this example, the predicate id BETWEEN 10 AND 25 (i.e. id >=10 AND id <=25) pushed to index scan.

Span for	Low	High	Inclusion
id BETWEEN 10 AND 25	10	25	3 (BOTH)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE id BETWEEN 10 AND 25;

        "spans": [
            {
                "Range": {
                    "High": [
                         "25"
                     ],
                     "Inclusion": 3,
                     "Low": [
                         "10"
                     ]
                }
            }
        ]
```

Example 10: Simple OR Predicate

```
SELECT meta().id FROM `travel-sample` WHERE id = 10 OR id = 20;
```

In this example, the predicate id = 10 OR id = 20 produce two independent ranges and both of them are pushed to index scan. Duplicate ranges are eliminated, but overlaps are not eliminated.

Span for	Low	High	Inclusion
id = 10	10	10	3 (BOTH)
Id = 20	20	20	3 (BOTH)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE id = 10 OR id = 20;

        "spans": [
            {
                "Range": {
                    "High": [
                         "10"
                     ],
                     "Inclusion": 3,
                     "Low": [
                         "10"
                     ]
                }
            },
            {
                "Range": {
                    "High": [
                         "20"
                     ],
                     "Inclusion": 3,
                     "Low": [
                         "20"
                     ]
                }
            }
        ]
```

Example 11: Simple IN Predicate

```
SELECT meta().id FROM `travel-sample` WHERE id IN [10, 20];
```

In this example, the predicate id IN [10, 20] (i.e. id = 10 OR id = 20) after eliminating duplicates each element as pushed as a separate range to index scan.
In Couchbase Server 4.5 up to 8192 IN elements are pushed as a separate range to the indexer. If the number of elements more than 8192 it does full scan on that key.

Span for	Low	High	Inclusion
id = 10	10	10	3 (BOTH)
Id = 20	20	20	3 (BOTH)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE id IN [10, 20];

        "spans": [
            {
                "Range": {
                    "High": [
                        "10"
                    ],
                    "Inclusion": 3,
                    "Low": [
                        "10"
                    ]
                }
            },
            {
                "Range": {
                    "High": [
                        "20"
                    ],
                    "Inclusion": 3,
                    "Low": [
                        "20"
                    ]
                }
            }
        ]
```

Example12: OR, BETWEEN, AND Predicate

```
SELECT meta().id FROM `travel-sample` WHERE (id BETWEEN 10 AND 25) OR (id > 50
AND id <= 60);
```

In this example, the predicate (id BETWEEN 10 AND 25) OR (id > 50 AND id <= 60) pushed to index scan.

Span for	Low	High	Inclusion
id between 10 and 25	10	25	3 (BOTH)
Id > 50 AND id <= 60	50	60	2 (HIGH)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE (id BETWEEN 10 AND 25) OR
(id > 50 AND id <= 60);

        "spans": [
            {
                "Range": {
                    "High": [
                        "25"
                    ],
                    "Inclusion": 3,
                    "Low": [
                        "10"
                    ]
                }
            },
            {
                "Range": {
                    "High": [
                        "60"
                    ],
                    "Inclusion": 2,
                    "Low": [
                        "50"
                    ]
                }
            }
        ]
```

Example 13: NOT Predicate

```
SELECT meta().id FROM `travel-sample` WHERE id <> 10;
```

In this example, the predicate id <> 10 is transformed to id < 10 OR id > 10 and pushed to index scan.

Span for	Low	High	Inclusion
id < 10	null	10	0 (NEITHER)
Id > 10	10	unbounded	0 (NEITHER)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE id <> 10;
        "spans": [
            {
                "Range": {
                    "High": [
                        "10"
                    ],
                    "Inclusion": 0,
                    "Low": [
                        "null"
                    ]
                }
            },
            {
                "Range": {
                     "Inclusion": 0,
                     "Low": [
                         "10"
                     ]
                }
            }
        ]
```

Example 14: NOT, AND Predicate

```
SELECT meta().id FROM `travel-sample` WHERE NOT (id >= 10 AND id < 25);
```

In this example, the predicate NOT (id >= 10 AND id < 25) is transformed to id < 10 OR id >=25 and pushed to index scan.

Span for	Low	High	Inclusion
id < 10	null	10	0 (NEITHER)
Id >= 25	25	unbounded	1 (LOW)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE NOT (id >= 10 AND id < 25);

   "spans": [
         {
             "Range": {
                 "High": [
                     "10"
                 ],
                 "Inclusion": 0,
                 "Low": [
                     "null"
                 ]
             }
         },
         {
             "Range": {
                 "Inclusion": 1,
                 "Low": [
                     "25"
```

```
                    ]
                }
            }
        ]
```

Example 15: EQUALITY Predicate on String Type

```
SELECT meta().id FROM `travel-sample` WHERE name = "American Airlines";
```

In this example, the predicate name = "American Airlines" pushed to index scan.

Span for	Low	High	Inclusion
name = "American Airlines"	"American Airlines"	"American Airlines"	3 (BOTH)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE name = "American Airlines";

        "spans": [
            {
                "Range": {
                    "High": [
                        "\"American Airlines\""
                    ],
                    "Inclusion": 3,
                    "Low": [
                        "\"American Airlines\""
                    ]
                }
            }
        ]
```

Example 16: Range Predicate on String Type

```
SELECT meta().id FROM `travel-sample` WHERE name >= "American Airlines" AND
name <= "United Airlines";
```

In this example, the predicate name >= "American Airlines" AND name <= "United Airlines"pushed to index scan.

Span for	Low	High	Inclusion
name >= "American Airlines" AND name <= "United Airlines"	"American Airlines"	"United Airlines"	3 (BOTH)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE name >= "American Airlines"
AND name <= "United Airlines";

"spans": [
         {
             "Range": {
                 "High": [
                      "\"United Airlines\""
                  ],
                  "Inclusion": 3,
                  "Low": [
                      "\"American Airlines\""
                  ]
             }
         }
     ]
```

Example 17: LIKE Predicate

```
SELECT meta().id FROM `travel-sample` WHERE name LIKE "American%";
```

In this example, the predicate name LIKE “American%” is transformed to name >= “American” AND name < “Americao” (i.e. "Americao" is next N1QL collation order of "American") and pushed to index scan. In LIKE predicate % means match with any number of any character.

Span for	Low	High	Inclusion
name LIKE "American%"	"American"	"Americao"	1 (LOW)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE name LIKE "American%";

     "spans": [
         {
             "Range": {
                 "High": [
                      "\"Americao\""
                  ],
                  "Inclusion": 1,
                  "Low": [
                      "\"American\""
                  ]
             }
         }
     ]
```

Example 18: LIKE Predicate

```
SELECT meta().id FROM `travel-sample` WHERE name LIKE "%American%";
```

In this example, the predicate name LIKE “%American%” is transformed and pushed to index scan. In LIKE predicate ‘%’ is leading portion of the string so we can’t push any portion of the string to indexer. “” is the lowest string. [] is empty array and is greater than every string value in N1QL collation order.

Span for	Low	High	Inclusion
name LIKE "%American%"	""	"[]"	1 (LOW)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE name LIKE "%American%";

     "spans": [
         {
             "Range": {
                 "High": [
                      "[]"
                  ],
                  "Inclusion": 1,
                  "Low": [
                      "\"\""
                  ]
             }
```

```
        }
    ]
```

Example 19: AND Predicate With Composite Index

```
SELECT meta().id FROM `travel-sample` WHERE sourceairport = "SFO" AND
destinationairport = "JFK" AND stops BETWEEN 0 AND 2;
```

In this example, the predicate sourceairport = "SFO" AND destinationairport = "JFK" and stops BETWEEN 0 AND 2 pushed to index scan.

Span for	Low	High	Inclusion
sourceairport = "SFO"	"SFO"	"SFO"	3 (BOTH)
destinationairport = "JFK"	"JFK"	"JFK"	3 (BOTH)
stops	0	2	3 (BOTH)

The index selected has keys (sourceairport,destinationairport,stops) the spans are stitched in that order create composite span as follows.

Span for	Low	High	Inclusion
sourceairport = "SFO" AND destinationairport = "JFK" and stops BETWEEN 0 AND 2	["SFO", "JFK",0]	["SFO", "JFK",2]	3 (BOTH)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE sourceairport = "SFO" AND
destinationairport = "JFK" AND stops BETWEEN 0 AND 2;

     "spans": [
         {
             "Range": {
                 "High": [
                    "\"SFO\"",
                    "\"JFK\"",
                    "2"
                 ],
                 "Inclusion": 3,
                 "Low": [
                    "\"SFO\"",
                                    "\"JFK\"",
                                    "0"
                                ]
                            }
                        }
                    ]
```

Example 20: AND Predicate With Composite Index

```
SELECT meta().id from `travel-sample`
WHERE sourceairport IN ["SFO", "SJC"]
AND destinationairport = "JFK"
AND stops = 0;
```

In this example, the predicate sourceairport IN ["SFO", "SJC"] AND destinationairport = "JFK" AND stops = 0pushed to index scan.

Span for	Low	High	Inclusion
sourceairport IN ["SFO", “SJC”]	"SFO" “SJC”	"SFO" “SJC”	3 (BOTH) 3 (BOTH)
destinationairport = "JFK"	"JFK"	"JFK"	3 (BOTH)
stops	0	0	3 (BOTH)

The index selected has keys (sourceairport, destinationairport, stops) the spans are stitched in that order create composite span as follows.

Span for	Low	High	Inclusion
sourceairport IN ["SFO", ”SJC”] AND destinationairport = "JFK" and stops = 0	["SFO", "JFK",0] [“SJC”, ”JFK”,0]	["SFO", "JFK",2] [“SJC”, ”JFK”,0]	3 (BOTH) 3 (BOTH)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE sourceairport IN ["SFO",
"SJC"] AND destinationairport = "JFK" AND stops = 0;

     "spans": [
         {
             "Range": {
                  "High": [
                     "\"SFO\"",
                     "\"JFK\"",
                     "0"
                  ],
                  "Inclusion": 3,
                  "Low": [
                     "\"SFO\"",
                     "\"JFK\"",
                     "0"
                  ]
             }
         },
             "Range": {
                  "High": [
                     "\"SJC\"",
                     "\"JFK\"",
                     "0"
                  ],
                  "Inclusion": 3,
                  "Low": [
                     "\"SJC\"",
                     "\"JFK\"",
                     "0"
                  ]
             }
         }
     ]
```

Example 21: Composite AND Predicate With Trailing Keys Are Missing in Predicate

```
SELECT meta().id FROM `travel-sample` WHERE sourceairport = "SFO" AND
destinationairport = "JFK";
```

In this example, the predicate sourceairport = "SFO" AND destinationairport = "JFK" pushed to index scan.

Span for	Low	High	Inclusion
sourceairport = "SFO"	"SFO"	"SFO"	3 (BOTH)
destinationairport = "JFK"	"JFK"	"JFK"	3 (BOTH)

The index selected has keys (sourceairport, destinationairport, stops). As the stop key predicate is missing in the query the one before span high value is converted to successor and removed inclusive bit. The transformed spans are

Span for	Low	High	Inclusion
sourceairport = "SFO"	"SFO"	"SFO"	3 (BOTH)
destinationairport = "JFK"	"JFK"	successor("JFK")	1 (LOW)

Then spans are stitched in that order create composite span as follows.

Span for	Low	High	Inclusion
sourceairport = "SFO" AND destinationairport = "JFK"	["SFO","JFK"]	["SFO",successor("JFK")]	1 (LOW)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE sourceairport = "SFO" AND
destinationairport = "JFK";

    "spans": [
        {
            "Range": {
                "High": [
                    "\"SFO\"",
                    "successor(\"JFK\")"
                ],
                "Inclusion": 1,
                "Low": [
                    "\"SFO\"",
                    "\"JFK\""
                ]
            }
        }
    ]
```

Example 22: Composite AND Predicate With Unbounded High of Trailing Key

```
SELECT meta().id FROM `travel-sample` WHERE sourceairport = "SFO" AND
destinationairport = "JFK" AND stops >= 0;
```

In this example, the predicate sourceairport = "SFO" AND destinationairport = "JFK" AND stops >= 0 pushed to index scan.

Span for	Low	High	Inclusion
sourceairport = "SFO"	"SFO"	"SFO"	3 (BOTH)
destinationairport = "JFK"	"JFK"	"JFK"	3 (BOTH)
stops >= 0	0	unbounded	1 (LOW)

The index selected has keys (sourceairport, destinationairport, stops). As the stop key high is unbounded before span high value is converted to successor and removed inclusive bit. The transformed spans are

Span for	Low	High	Inclusion
sourceairport = "SFO"	"SFO"	"SFO"	3 (BOTH)
destinationairport = "JFK"	"JFK"	successor("JFK")	1 (LOW)
stops >= 0	0	unbounded	1 (LOW)

Then spans are stitched in that order create composite span as follows.

Span for	Low	High	Inclusion
sourceairport = "SFO" AND destinationairport = "JFK" AND stops >= 0	["SFO", "JFK",0]	["SFO",successor("JFK")]	1 (LOW)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE sourceairport = "SFO" AND
destinationairport = "JFK" AND stops >= 0;
```

```
    "spans": [
         {
             "Range": {
                  "High": [
                     "\"SFO\"",
                     "successor(\"JFK\")"
                  ],
                  "Inclusion": 1,
                  "Low": [
                     "\"SFO\"",
                     "\"JFK\"",
                     "0"
                  ]
             }
         }
    ]
```

Example 23: Equality Predicate With Query Parameters

```
SELECT meta().id FROM `travel-sample` WHERE id = $1;
```

In this example, the predicate id = $1 pushed to index scan.

Span for	Low	High	Inclusion
id = $1	$1	$1	3 (BOTH)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE id = $1;

   "spans": [
         {
             "Range": {
                 "High": [
                      "$1"
                  ],
                  "Inclusion": 3,
                  "Low": [
                      "$1"
                  ]
             }
         }
     ]
```

Example 24: AND Predicate With Query Parameters

```
SELECT meta().id FROM `travel-sample` WHERE id >= $1 AND id < $2;
```

In this example, the predicate id >= $1 AND id < $2 pushed to index scan.

Span for	Low	High	Inclusion
id >= $1 AND id < $2	$1	$2	1 (LOW)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE id >= $1 AND id < $2;

    "spans": [
         {
             "Range": {
                 "High": [
                      "$2"
                  ],
                  "Inclusion": 1,
                  "Low": [
                      "$1"
                  ]
             }
         }
     ]
```

Example 25: OR Predicate With Query Parameters

```
SELECT meta().id FROM `travel-sample` WHERE id = $1 OR id < $2;
```

In this example, the predicate id = $1 OR id < $2 pushed to index scan.

Span for	Low	High	Inclusion
id = $1	$1	$1	3 (BOTH)
Id < $2	null	$2	0 (NEITHER)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE id = $1 OR id < $2;
```

```
        "spans": [
            {
                "Range": {
                    "High": [
                         "$1"
                     ],
                     "Inclusion": 3,
                     "Low": [
                         "$1"
                     ]
                },
                "Range": {
                    "High": [
                         "$2"
                     ],
                     "Inclusion": 0,
                     "Low": [
                         "null"
                     ]
            }
        }
    ]
```

Example 26: IN Predicate With Query Parameters

```
SELECT meta().id FROM `travel-sample` WHERE id IN [ $1, 10, $2] ;
```

In this example, the predicate id IN [$1, 10, $2] pushed to index scan. This is improved plan for Couchbase Server 4.5.1

Span for	Low	High	Inclusion
id IN [$1, 10, $2]	$1 10 $2	$1 10 $2	3 (BOTH) 3 (BOTH) 3 (BOTH)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE id IN [$1, 10, $2];
```

```
    "spans": [
        {
            "Range": {
                "High": [
                    "$1"
                 ],
                 "Inclusion": 3,
                 "Low": [
                    "$1"
                 ]
            },
            "Range": {
                "High": [
                    "10"
                    ],
                    "Inclusion": 3,
                    "Low": [
                        "10"
                    ]
                },
                "Range": {
                    "High": [
                        "$2"
                    ],
                    "Inclusion": 3,
                    "Low": [
                        "$2"
                    ]
                }
            }
        ]
```

Example 27: ANY Predicate

```
SELECT meta().id
FROM `travel-sample`
WHERE type = "route" AND
ANY v IN schedule SATISFIES v.day = 0 END ;
```

In this example, the predicate v.day = 0 pushed to ARRAY index scan.

Span for	Low	High	Inclusion
v.day = 0	0	0	3 (BOTH)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE type = "route" AND ANY v IN
schedule SATISFIES v.day = 0 END ;
```

```
        "spans": [
            {
                "Range": {
                    "High": [
                         "0"
                     ],
                     "Inclusion": 3,
                     "Low": [
                         "0"
                     ]
                }
            }
        ]
```

Example 28: ANY Predicate

```
SELECT meta().id
FROM `travel-sample`
WHERE type = "route" AND
      ANY v IN schedule SATISFIES v.day IN [1,2,3] END ;
```

In this example, the predicate v.day IN [1,2,3] pushed to ARRAY index scan.

Span for	Low	High	Inclusion
v.day IN [1,2,3]	1 2 3	1 2 3	3 (BOTH) 3 (BOTH) 3 (BOTH)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE type = "route" AND ANY v IN
schedule SATISFIES v.day IN [1,2,3] END ;
```

```
        "spans": [
            {
                "Range": {
```

```
            "High": [
                 "1"
             ],
             "Inclusion": 3,
             "Low": [
                 "1"
             ]
        },
        "Range": {
            "High": [
                 "2"
             ],
             "Inclusion": 3,
             "Low": [
                 "2"
             ]
        },
        "Range": {
            "High": [
                 "3"
             ],
             "Inclusion": 3,
             "Low": [
                 "3"
             ]
        }
    }
]
```

Note: The following examples don't have the right indexes, or the queries need to be modified to produce an optimal plan.

Example 29: Equality Predicate on Expression

```
SELECT meta().id FROM `travel-sample` WHERE abs(id) = 10;
```

In this example, NO predicate is pushed to index scan.

Span for	Low	High	Inclusion
id	null	unbounded	0 (NEITHER)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE abs(id) = 10;
```

```
"spans": [
     {
              "Inclusion": 0,
              "Low": [
                  "null"
              ]
          }
     }
 ]
```

The span indicates we are doing a complete index scan, if it is not covered index we fetch the document from Data node and apply predicate. For better performance create new index as follows:

```
CREATE INDEX `idx_absid` ON `travel-sample`(abs(`id`));
```

When idx_absid is used predicate abs(id) = 10 pushed to index scan.

Span for	Low	High	Inclusion
abs(id) = 10	10	10	3 (BOTH)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE abs(id) = 10;

     "spans": [
         {
             "Range": {
                 "High": [
                      "10"
                  ],
                  "Inclusion": 3,
                  "Low": [
                      "10"
                  ]
             }
         }
     ]
```

Example 30: Overlapping Predicates

```
SELECT meta().id
FROM `travel-sample`
WHERE id <= 100 OR (id BETWEEN 50 AND 150);
```

In this example, id <= 100 OR (id between 50 and 150) predicates are pushed to index scan as two ranges.

Span for	Low	High	Inclusion
id <= 100	null	100	2 (HIGH)
Id between 50 and 150	50	150	3 (BOTH)

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE id <= 100 OR (id BETWEEN 50
AND 150);
    "spans": [
         {
             "Range": {
                 "High": [
                      "100"
                  ],
                  "Inclusion": 2,
                  "Low": [
                      "null"
                  ]
             }
         },
         {
             "Range": {
                 "High": [
                      "150"
                  ],
                  "Inclusion": 3,
                  "Low": [
                      "50"
                  ]
             }
         }
     ]
```

The spans indicate we are doing two range scans. When we observe closely, the ranges are overlapped. The values 50-100 are scanned twice (in future Couchbase Server releases, this might be improved). To eliminate duplicates N1QL does distinct operations on meta().id. If possible, rewriting query will eliminate duplicates.

```
EXPLAIN SELECT meta().id FROM `travel-sample` WHERE id <= 150;

     "spans": [
         {
             "Range": {
                 "High": [
                      "150"
                  ],
                  "Inclusion": 2,
                  "Low": [
                      "null"
                  ]
             }
         }
     ]
```

Summary

When you analyze the explain plan, correlate the predicates in the explain to the spans. Ensure the most optimal index is selected and the spans have the expected range for all the index keys. More keys in each span will make the query more efficient.

COUNT and GROUP Faster With N1QL

Author: Keshav Murthy

Humans have counted things for a very long time. In database applications, COUNT() is frequently used in various contexts. The COUNT performance affects both application performance and user experience. Keeping this mind, Couchbase supported generalized COUNT() and has improved its performance in Couchbase 4.5 release.

Two Couchbase 4.5 Features

There are two features in Couchbase 4.5 helping the COUNT performance. When the query is interested only in the COUNT of a range of data that's indexed, the indexer does the counting itself. In other words, the query pushes the counting to the index. This reduces the amount of data exchanged between indexer and query, improving the query speed.

```
CREATE INDEX idxname ON `travel-sample` (name);
SELECT COUNT(name)
FROM `travel-sample`
WHERE name = 'Air Alaska';
```

First, when all of the predicates can be pushed down to index scan and we know index scan won't result in any false positives, the index can simply do the COUNT() of qualifying values without returning the qualifying data to the query engine. This improves the COUNT performance. When the COUNT() is pushed to the index scan, the optimizer will denote this by choosing the **IndexCountScan** method in the query plan. The qualifying queries benefit directly, without any effort from the application developer.
The predicate (name = 'Air Alaska') is pushed down to the index scan in addition to the COUNT() aggregate as well.

```
{
   "#operator": "IndexCountScan",
   "covers": [
       "cover ((`travel-sample`.`name`))",
       "cover ((meta(`travel-sample`).`id`))"
   ],
   "index": "ixname",
   "index_id": "ea81dd71a0a98351",
   "keyspace": "travel-sample",
   "namespace": "default",
   "spans": [
       {
           "Range": {
               "High": [
                   "\"Air Alaska\""
               ],
               "Inclusion": 3,
               "Low": [
                   "\"Air Alaska\""
               ]
           }
```

- Second, the optimal execution of the MIN() aggregate when the query is covered and can be pushed down to index.

```
SELECT MIN(name)
FROM `travel-sample
WHERE prodname > 'Air Alaska';
```

For this query, the first name that's greater than "Air Alaska" is determined by the index scan. Because we're interested in the just one minimum value and therefore, in the query plan, we push the {limit:1} parameter to the index scan. This works very quickly compared to getting all the qualifying values from the index scan to determine the minimum value.

```
{
   "#operator": "IndexScan",
   "covers": [
       "cover ((`travel-sample`.`name`))",
       "cover ((meta(`travel-sample`).`id`))"
   ],
   "index": "ixname",
   "index_id": "ea81dd71a0a98351",
   "keyspace": "travel-sample",
   "limit": "1",
   "namespace": "default",
   "spans": [
   {
       "Range": {
       "Inclusion": 0,
       "Low": [
           "\"Air Alaska\""
       ]
   }
```

Use Cases

Let’s walk through common use cases and how these features work together to speed up many queries. The examples use travel-sample data set shipped along with Couchbase server.

There are some interesting uses with a little bit of help from the application developer. Since multiple customers are using this approach, it’s worth checking out if this is going to help you as well.

Let’s look at common questions and how to improve the performance with Couchbase 4.5.

Use Case 1

What’s the total number of documents in the bucket?

```
cbq> select count(*) from `travel-sample`;
{
   "requestID": "15549343-54ec-44d8-b62a-a4d98445b484",
   "signature": {
       "$1": "number"
   },
   "results": [
       {
           "$1": 31591
       }
   ],
   "status": "success",
   "metrics": {
       "elapsedTime": "7.724136ms",
       "executionTime": "7.693342ms",
       "resultCount": 1,
       "resultSize": 35
   }
}
```

Counting of the whole bucket runs in a few milliseconds because we can directly get the total number of documents directly from the bucket metadata.

```
{
    "#operator": "CountScan",
    "keyspace": "travel-sample",
    "namespace": "default"
}
```

Use Case 2

Let’s see how many document types we have in the travel-sample bucket:

```
> select distinct type from `travel-sample`;
{
   "requestID": "584a7785-95fe-4d4a-b72b-7e161858bf2a",
   "signature": {
       "type": "json"
   },
   "results": [
       {
           "type": "airline"
       },
       {
           "type": "route"
       },
       {
           "type": "hotel"
       },
       {
           "type": "landmark"
       },
       {
           "type": "airport"
       }
   ],
   "status": "success",
   "metrics": {
       "elapsedTime": "1.031288771s",
       "executionTime": "1.031253954s",
       "resultCount": 5,
        "resultSize": 202
      }
    }
```

One full second for this seems quite expensive. Let’s see the query plan.

```
    {
      "#operator": "PrimaryScan",
      "index": "def_primary",
      "keyspace": "travel-sample",
      "namespace": "default",
      "using": "gsi"
    }
```

To determine the number of distinct items, this plan will scan the whole bucket, which determines the distinct types. Let’s see how we can improve this.

The MIN() optimization in Couchbase 4.5 can be exploited to optimize this. We can evaluate if the field is the leading key of any index, as the index has pre-sorted data. Using this, we can write a small script to calculate the distinct values much faster.

Step 1

Get the first entry in the index.

```
        SELECT MIN(type)
        FROM `travel-sample`
        WHERE type IS NOT MISSING;
    {
    "results": [
          {
              "$1": "airline"
          }
      ],
      "status": "success",
      "metrics": {
          "elapsedTime": "15.050713ms",
          "executionTime": "15.023588ms",
          "resultCount": 1,
          "resultSize": 39
      }
   }
```

The Query Plan shows that this is done by index scan only. The scan parameter "limit": "1" shows why we get the results so fast. Because the index stores data in a sorted order, the first item will be the lowest value.

```
        {
            "#operator": "IndexScan",
            "covers": [
                "cover ((`travel-sample`.`type`))",
                "cover ((meta(`travel-sample`).`id`))"
            ],
            "index": "def_type",
            "index_id": "7cea57503ecfe0d3",
            "keyspace": "travel-sample",
            "limit": "1",
            "namespace": "default",
            "spans": [
                {
                    "Range": {
                        "Inclusion": 0,
                        "Low": [
                            "null"
                        ]
                    }
                }
            ],
            "using": "gsi"
        }
```

Step 2

Now we use the index to find the next value for the type.

Use the predicate (type > "airline"). Now you get airport:

```
SELECT MIN(type)
FROM `travel-sample`
WHERE type > "airline";
{
"results": [
       {
           "$1": "airport"
       }
   ],
   "status": "success",
   "metrics": {
       "elapsedTime": "10.370383ms",
       "executionTime": "10.346036ms",
       "resultCount": 1,
       "resultSize": 39
   }
}
```

Repeat this step 2 using the previous value in the predicate until you get NULL as the result.

A simple Python script will get the job done.

```
import requests
import json
url="http://localhost:8093/query"
s = requests.Session()
s.keep_alive = True
s.auth = ('Administrator','password')
query = {'statement':'SELECT MIN(type) minval FROM `travel-sample` WHERE type IS NOT MISSING ;'}
r = s.post(url, data=query, stream=False, headers={'Connection':'close'})
result = r.json()['results'][0]
lastval = result['minval']
while lastval != None:
   print lastval
   stmt = 'SELECT MIN(type) minval FROM `travel-sample` WHERE type > "' + lastval + '";';
   query = {'statement':stmt}
   r = s.post(url, data=query, stream=False, headers={'Connection':'close'})
   result = r.json()['results'][0]
   lastval = result['minval']
```

Pro Tip

- Use a prepared statement for the SELECT statement in the while loop to improve the performance even further.

Case 3

Grouping works closely with counting. It’s common to group the data by type, name, and date and count them. If you have a dataset with millions of documents, the index scan itself may take minutes to scan and retrieve all the data.

Here is a simple example on travel-sample.

```
SELECT type, count(type)
FROM `travel-sample`
GROUP BY type;
   "metrics": {
       "elapsedTime": "904.607551ms",
       "executionTime": "904.585043ms",
       "resultCount": 5,
       "resultSize": 321
   }
```

This scans the entire bucket to calculate the groups and counts. You can improve the performance by using the index scan.

```
SELECT type, count(type)
FROM `travel-sample`
WHERE type IS NOT MISSING
GROUP BY type;
The query plan uses  the index now.
       {
         "#operator": "IndexScan",
         "covers": [
           "cover ((`travel-sample`.`type`))",
           "cover ((meta(`travel-sample`).`id`))"
         ],
         "index": "def_type",
         "index_id": "a1ae13a30ad4408",
         "keyspace": "travel-sample",
         "namespace": "default",
...
```

This query runs faster as well.

```
   "metrics": {
       "elapsedTime": "212.792255ms",
       "executionTime": "212.771933ms",
       "resultCount": 5,
       "resultSize": 321
   }
```

Let’s see if we can make this faster.

Using the combination of MIN() and COUNT() optimization features in 4.5, we can improve this further by writing a small script.

Here is the outline.

Step 1: Get the first entry in the index for the type.

Step 2: Then, COUNT() from the data set where type = first-value.

Step 3: Now we use the index to find the next value for type.

Step 4: Repeat step 2 and 3 for all the values of type.

Here’s the actual Python script:

```
import requests
import json
url="http://localhost:8093/query"
s = requests.Session()
s.keep_alive = True
s.auth = ('Administrator','password')
query = {'statement':'SELECT MIN(type) minval FROM `travel-sample` WHERE type
IS NOT MISSING ;'}
r = s.post(url, data=query, stream=False, headers={'Connection':'close'})
result = r.json()['results'][0]
lastval = result['minval']
while lastval != None:
   stmt = 'SELECT COUNT(type) tcount FROM `travel-sample` WHERE type = "' +
lastval + '";';
   query = {'statement':stmt}
   r = s.post(url, data=query, stream=False, headers={'Connection':'close'})
   result = r.json()['results'][0]
```

```
        tcount = result['tcount']
        print lastval, tcount
        stmt = 'SELECT MIN(type) minval FROM `travel-sample` WHERE type > "' +
    lastval + '";';
        query = {'statement':stmt}
        r = s.post(url, data=query, stream=False, headers={'Connection':'close'})
        result = r.json()['results'][0]
        lastval = result['minval']
```

Let's run this!

```
    $ time python ./group.py
    airline 187
    airport 1968
    hotel 917
    landmark 4495
    route 24024
    real0m0.372s
    user0m0.079s
    sys0m0.036s
    $
```

This is run on travel-sample with about 31,000 documents. The Python script itself runs about 10 SELECT statements to retrieve this result. As the number of documents in the dataset increases, the script approach will run much faster to a single statement. Further, using the prepared statement, you can save more CPU cycles.

Conclusion

The MIN() and COUNT() optimizations in 4.5 improves query performance. Using additional logic, you can use these optimizations to improve the performance of a variety of queries.

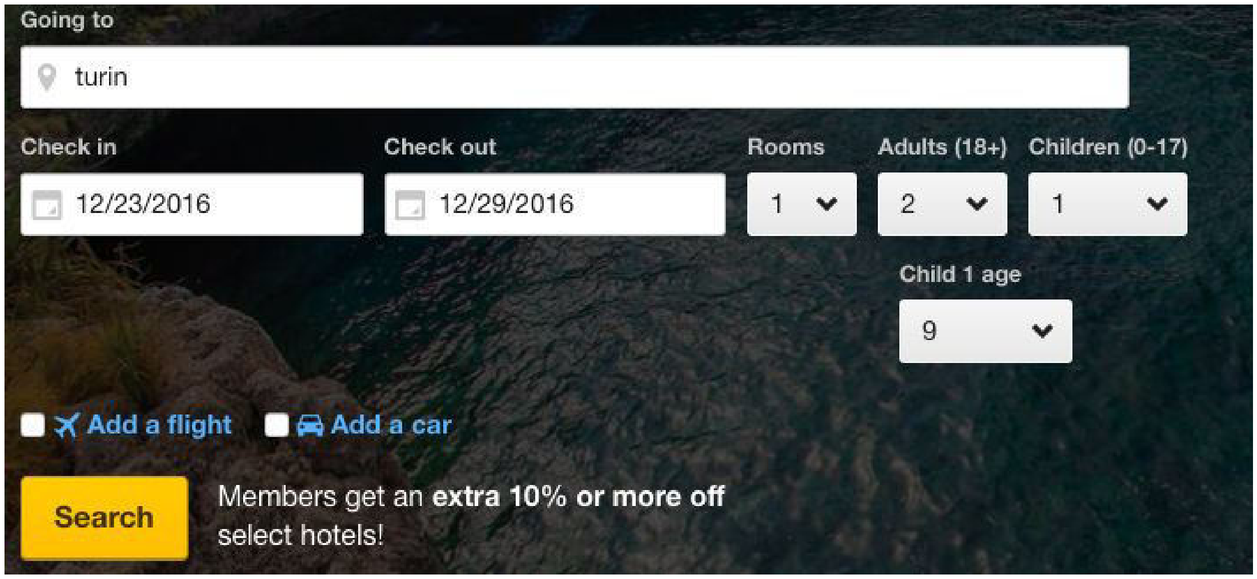
More Than LIKE: Efficient JSON Searching With N1QL

Author: Keshav Murthy

Enterprise applications require both exact search (equality, range predicate) and pattern search (LIKE or text search). The B-Tree based indexes in databases help perform an exact search in milliseconds. Pattern search is a whole different problem. Predicates of patterns (name LIKE "%turin%") will scan the whole index — unsuitable for application use. Recognizing the need for speed, we've introduced a TOKENS() functions in Couchbase 4.6 which retuns an array of tokens from each field or document. We then index the array using the array indexing in Couchbase N1QL. This transforms the pattern search problem to a simple equality look up, improves the performance by orders of magnitude.

Introduction

Whether you're booking a hotel in Turin or Timbuktu, Milan or Malibu, you need to search based on various criteria: location, dates, rooms, number of people, their age and more. The numerical requirements (number of rooms, number of people in the party), date ranges (check-in and checkout dates) are exact predicates. The target place to visit could be a search within a city name, region name, nearby places, or interesting landmarks.



Here is the problem.

- How do you make this query run really, really quickly?
- How do you get thousands of users searching and buying your product and service concurrently?

Search is a broad term. So, we divide searches into distinct types.

1. Exact search with one or more predicates.
2. Pattern search to find a word or a pattern within a larger string.
3. Fuzzy search — text search with a lot of text data.

Of course, all of these search operations have to come back with results in milliseconds and not seconds.

In Couchbase:

- For exact searches, you can use B-tree- or skip-list-based global secondary indexes with a simple or composite index to achieve your goals.
- For full-text search, which requires heavy text processing — stemming, scoring, and more — you can use Couchbase full-text search (FTS).

The hotel search problem doesn't fall nicely into either category completely. Exact predicates require exact index; search predicates require FTS index. Using only index scans, you'll get too much data, and you'd need filter further. On the other hand, if you search using full-text index, you'll need to filter further for dates, rooms, etc. This second level of filtering will slow things down significantly.

Real world problems like you saw above are not always clean cut. You want the exact search for numerical and date range; you want the string search with smaller data like city, neighborhood, region.
Here is the problem. How do you make this query run really, really quickly? How do you get thousands of users searching and buying your product and service concurrently?
Let’s use sample Hotel documents in the Couchbase travel-sample data set to explore the use case.

```
select hotel, meta().id as dockey
from `travel-sample` hotel
where hotel.type = "hotel"
limit 1;
[
 {
    "dockey": "hotel_10025",
    "hotel": {
     "address": "Capstone Road, ME7 3JE",
     "alias": null,
     "checkin": null,
     "checkout": null,
     "city": "Medway",
     "country": "United Kingdom",
     "description": "40 bed summer hostel about 3 miles from Gillingham, housed
in a distinctive converted Oast House in a semi-rural setting.",
     "directions": null,
     "email": null,
     "fax": null,
     "free_breakfast": true,
     "free_internet": false,
     "free_parking": true,
     "geo": {
       "accuracy": "RANGE_INTERPOLATED",
       "lat": 51.35785,
       "lon": 0.55818
     },
     "id": 10025,
     "name": "Medway Youth Hostel",
     "pets_ok": true,
     "phone": "+44 870 770 5964",
      "price": null,
      "public_likes": [
        "Julius Tromp I",
        "Corrine Hilll",
        "Jaeden McKenzie",
        "Vallie Ryan",
        "Brian Kilback",
        "Lilian McLaughlin",
        "Ms. Moses Feeney",
        "Elnora Trantow"
      ],
      "reviews": [
        {
          "author": "Ozella Sipes",
          "content": "This was our 2nd trip here and we enjoyed it as much or
more than last year. Excellent location across from the French Market and just
across the street from the streetcar stop. Very convenient to several small but
good restaurants. Very clean and well maintained. Housekeeping and other staff
are all friendly and helpful. We really enjoyed sitting on the 2nd floor
terrace over the entrance and \"people-watching\" on Esplanade Ave., also
talking with our fellow guests. Some furniture could use a little updating or
replacement, but nothing major.",
          "date": "2013-06-22 18:33:50 +0300",
          "ratings": {
            "Cleanliness": 5,
            "Location": 4,
            "Overall": 4,
            "Rooms": 3,
            "Service": 5,
            "Value": 4
          }
        },
        {
          "author": "Barton Marks",
          "content": "We found the hotel de la Monnaie through Interval and we
thought we'd give it a try while we attended a conference in New Orleans. This
place was a perfect location and it definitely beat staying downtown at the
Hilton with the rest of the attendees. We were right on the edge of the French
Quarter within walking distance of the whole area. The location on Esplanade is
more of a residential area so you are near the fun but far enough away to enjoy
some quiet downtime. We loved the trolly car right across the street and we
took that down to the conference center for the conference days we attended. We
also took it up Canal Street and nearly delivered to the WWII museum. From
there we were able to catch a ride to the Garden District - a must see if you
love old architecture - beautiful old homes(mansions). We at lunch ate Joey K's
there and it was excellent. We ate so many places in the French Quarter I can't
remember all the names. My husband loved all the NOL foods - gumbo, jambalaya
and more. I'm glad we found the Louisiana Pizza Kitchen right on the other side
of the U.S. Mint (across the street from Monnaie). Small little spot but
excellent pizza! The day we arrived was a huge jazz festival going on across
the street. However, once in our rooms, you couldn't hear any outside noise.
Just the train at night blowing it's whistle! We enjoyed being so close to the
French Market and within walking distance of all the sites to see. And you
can't pass up the Cafe du Monde down the street - a busy happening place with
the best French doughnuts!!!Delicious! We will definitely come back and would
stay here again. We were not hounded to purchase anything. My husband only
received one phone call regarding timeshare and the woman was very pleasant.
The staff was laid back and friendly. My only complaint was the very firm bed.
Other than that, we really enjoyed our stay. Thanks Hotel de la Monnaie!",
         "date": "2015-03-02 19:56:13 +0300",
         "ratings": {
           "Business service (e.g., internet access)": 4,
           "Check in / front desk": 4,
           "Cleanliness": 4,
           "Location": 4,
           "Overall": 4,
           "Rooms": 3,
           "Service": 3,
           "Value": 5
         }
       }
     ],
     "state": null,
     "title": "Gillingham (Kent)",
     "tollfree": null,
     "type": "hotel",
     "url": "http://www.yha.org.uk",
     "vacancy": true
   }
 }
]
```

Now, let's see some common questions from your application and how these questions can be translated to queries.
1. Find all the hotel information for "Medway Youth Hostel"
* This is an exact search of the predicate (hotel.name = "Medway Youth Hostel")

```
Query:
SELECT hotel
FROM `travel-sample` hotel
WHERE hotel.type = "hotel"
AND hotel.name = "Medway Youth Hostel";
Index:
CREATE INDEX idx_hotel_typename
ON `travel-sample`(name) WHERE type = "hotel";
```

2. Find all the hotels in "Medway", "United Kingdom"
* This is an exact search with the predicate: (hotel.country = "United Kingdom" AND hotel.city = "Medway")

```
Query:
SELECT hotel
FROM `travel-sample` hotel
WHERE hotel.type = "hotel"
AND hotel.country = "United Kingdom"
AND hotel.city = "Medway"
Index:
CREATE INDEX idx_hotel_typecountryname
ON `travel-sample`(country, name) WHERE type = "hotel"
```

3. Find all the hotels liked (in the field public_field) by the a person with a name (first or last) Vallie
 * public_likes is an array of field. It has names of the peopel who liked the hotel.
* This is a pattern search. We're looking for a first name or last name Vallie.

```
"public_likes": [
  "Julius Tromp I",
  "Corrine Hilll",
  "Jaeden McKenzie",
  "Vallie Ryan",
  "Brian Kilback",
  "Lilian McLaughlin",
  "Ms. Moses Feeney",
  "Elnora Trantow"
```

```
        ]
```

So we use the array predicate to search through array elements. To make this search faster, we create an array index on public_likes. This index scan will still fetch all the elements from the index and apply the LIKE "%vallie%" on top of it.

```
Query:
SELECT hotel
FROM `travel-sample` hotel
WHERE hotel.type = "hotel"
AND ANY p IN hotel.public_likes
       SATISFIES LOWER(p) LIKE "%vallie%"
   END
Index:
CREATE INDEX idx_publiclikes ON `travel-sample`
(DISTINCT ARRAY LOWER(p) FOR p IN public_likes END)
WHERE type = "hotel";
Explain:
       {
         "#operator": "DistinctScan",
         "scan": {
           "#operator": "IndexScan",
           "index": "idx_publiclikes",
           "index_id": "9a10e62d4ad4387a",
           "keyspace": "travel-sample",
           "namespace": "default",
           "spans": [
             {
               "Range": {
                 "High": [
                   "[]"
                 ],
                 "Inclusion": 1,
                 "Low": [
                   "\"\""
                 ]
               }
             }
           ],
           "using": "gsi"
         }
       }
```

4. Find all the hotels that mention architecture anywhere in the document. This anywhere -- could be in keyname (attribute name), any value — in name, description, reviews, etc.

This has a pattern search. Search for architecture anywhere in the document is a pattern search.Index: This query can use a primary index.

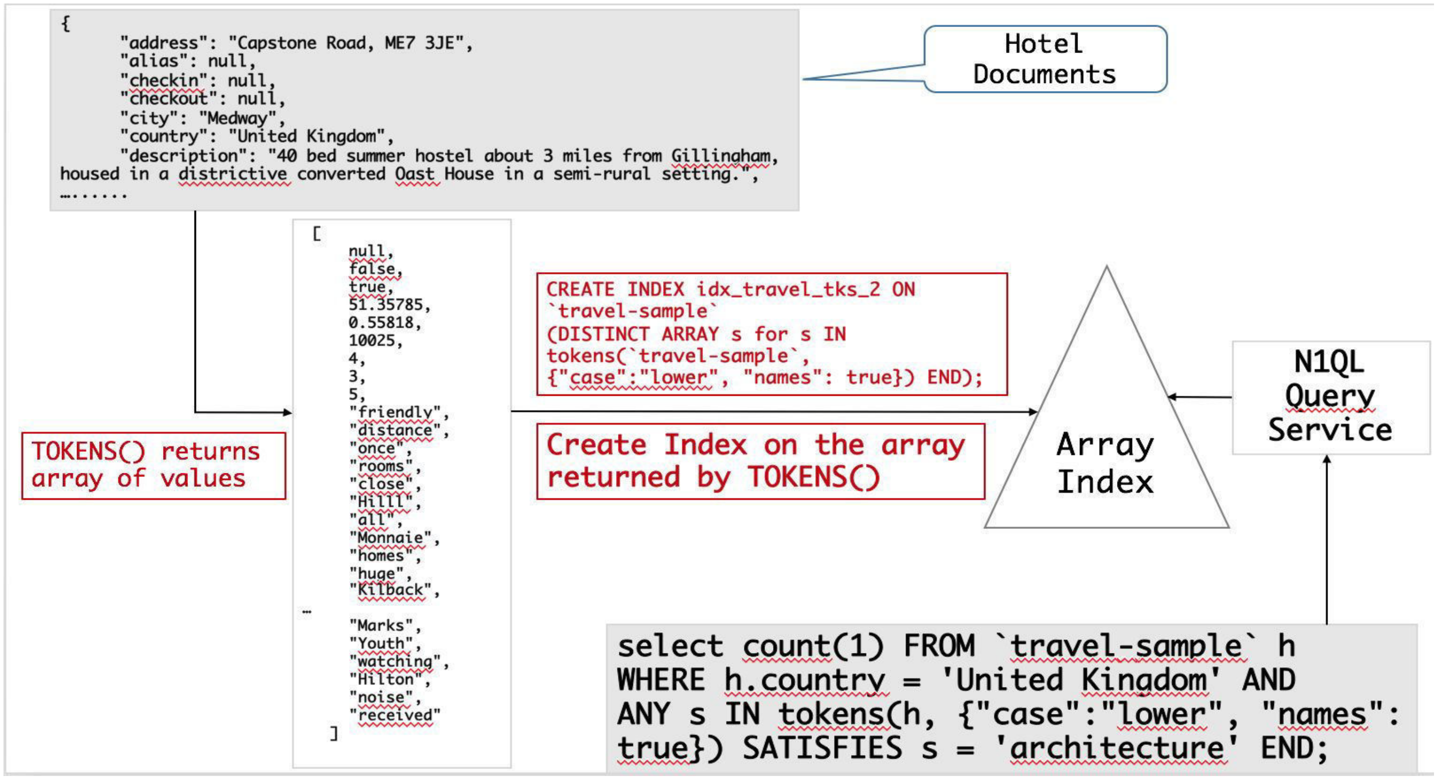
```
SELECT COUNT(1) totalcount FROM `travel-sample` hotel
WHERE ANY v WITHIN OBJECT_PAIRS(hotel)
         SATISFIES LOWER(v)  LIKE "%architecture%"
     END
{
   "requestID": "426b993e-a214-4e5f-824e-aef2ebfd3262",
   "signature": {
       "$1": "number"
   },
   "results": [
       {
           "$1": 94
       }
   ],
   "status": "success",
   "metrics": {
       "elapsedTime": "12.130759027s",
       "executionTime": "12.130687174s",
       "resultCount": 1,
       "resultSize": 32
   }
}
```

While this query works, it takes 12.13 seconds to find the 41 documents which have the word architecture somewhere in the document. Too slow and too expensive.
Here is where the TOKENS come into the picture. TOKENS() function takes a JSON document or an expression, analyzes all of name value pairs, arrays, objects and returns an array of values of basic types: numeric, strings, boolean, null.

```
SELECT TOKENS(hotel)
FROM `travel-sample` hotel
```

```
WHERE hotel.type = 'hotel'
LIMIT 1;
[
 {
   "$1": [
     null,
     false,
     true,
     51.35785,
     0.55818,
     10025,
     4,
     3,
     5,
     "friendly",
     "distance",
     "once",
     "rooms",
     "close",
     "Hilll",
     "all",
     "Monnaie",
     "homes",
     "huge",
     "Kilback",
     "vacancy",
     "Value",
     "Sipes",
     "call",
     "converted",
     "770",
…
     "Marks",
     "Youth",
     "watching",
     "Hilton",
     "noise",
     "received"
   ]
 }
]
```

Once we get this array, we can use the array indexing, to index the result of TOKENS and query them:


Hotel Documents
TOKENS() returns array of values
Create Index on the array returned by TOKENS()
Array Index
N1QL Query Service

```
{
      "address": "Capstone Road, ME7 3JE",
      "alias": null,
      "checkin": null,
      "checkout": null,
      "city": "Medway",
      "country": "United Kingdom",
      "description": "40 bed summer hostel about 3 miles from Gillingham,
housed in a districtive converted Oast House in a semi-rural setting.",
…......
```

```
[
    null,
    false,
    true,
    51.35785,
    0.55818,
    10025,
    4,
    3,
    5,
    "friendly",
    "distance",
    "once",
    "rooms",
    "close",
    "Hilll",
    "all",
    "Monnaie",
    "homes",
    "huge",
    "Kilback",
…
    "Marks",
    "Youth",
    "watching",
    "Hilton",
    "noise",
    "received"
  ]
```

```
CREATE INDEX idx_travel_tks_2 ON
`travel-sample`
(DISTINCT ARRAY s for s IN
tokens(`travel-sample`,
{"case":"lower", "names": true}) END);
```

```
select count(1) FROM `travel-sample` h
WHERE h.country = 'United Kingdom' AND
ANY s IN tokens(h, {"case":"lower", "names":
true}) SATISFIES s = 'architecture' END;
```

```
CREATE INDEX idx_travel_tokens ON `travel-sample`
(DISTINCT ARRAY s for s IN TOKENS(`travel-sample`, {"case":"lower", "names":
true}) END);

select count(1) FROM `travel-sample` h
WHERE ANY s IN tokens(h, {"case":"lower", "names": true}) SATISFIES s =
'architecture' END;


{
   "requestID": "dd6cfb24-ae5f-4667-a332-fc21ec48f808",
   "signature": {
       "$1": "number"
   },
   "results": [
       {
           "$1": 94
       }
   ],
   "status": "success",
   "metrics": {
       "elapsedTime": "33.398256ms",
       "executionTime": "33.359819ms",
       "resultCount": 1,
       "resultSize": 32
   }
}
```

Let’s look at the query plan. It chose the tokens right index and more importantly, the predicates were pushed correctly into index scans (reported here as spans). The appropriate index is selected and the predicate looking for architecture is pushed down to index scan.

```
        "~children": [
          {
            "#operator": "DistinctScan",
            "scan": {
              "#operator": "IndexScan",
              "index": "idx_travel_tokens",
              "index_id": "9f370cd078ebd2eb",
              "keyspace": "travel-sample",
              "namespace": "default",
              "spans": [
                {
                  "Range": {
                    "High": [
                      "\"architecture\""
                    ],
                    "Inclusion": 3,
                    "Low": [
                      "\"architecture\""
                    ]
                  }
                }
              ],
              "using": "gsi"
            }
          }
```

1. Finally, let’s put all of this together to answer this question: Find all the hotels in United Kingdom which have vacancies and with a website.

- We can check for (country = ‘United Kingdom’ and vacancy = true) easily via composite index.
- Having a “website” needs to be defined well.
- When you get information from hotels, unless you go through a ETL process to save the information in an exact field and format, you’d have this information anywhere in the hotel document.

2. Example:

```
"url": "http://www.yha.org.uk",
"Website": "http://www.yha.org.uk”
“Http”: "www.yha.org.uk"
“Http address”: "www.yha.org.uk"
“Web” : "Our website: http://www.yha.org.uk "
“Contact” :
    “Phone” : "+44 482 492 292",
    “Web” : "http://www.yha.org.uk",
    “Mail” : "help@yha.org.uk"
  }
“Info”: " +44 482 492 292 http://www.yha.org.uk"
“Info”: ["+44 482 492 292”,  “http://www.yha.org.uk"
```

Problem: You’d need to search in every field name, every object, every array.

```
Query:
EXPLAIN
SELECT *
FROM `travel-sample` hotel
WHERE country = "United Kingdom"
AND vacancy = true
AND ANY v IN TOKENS(hotel, {"case":"lower", "name":true})
       SATISFIES v = "http"
   END
AND type = "hotel"
Index
CREATE INDEX idx_countryvacancytoken ON `travel-sample`
 (country,
  vacancy,
  DISTINCT
    ARRAY v for v IN
    tokens(`travel-sample`, {"case":"lower", "name":true})
    END
 ) WHERE type = "hotel";
Plan
```

```
====
       {
         "#operator": "DistinctScan",
         "scan": {
           "#operator": "IndexScan",
           "index": "idx_countryvacancytoken",
           "index_id": "c44e9b1d7102a736",
           "keyspace": "travel-sample",
           "namespace": "default",
           "spans": [
             {
               "Range": {
                 "High": [
                   "\"United Kingdom\"",
                   "true",
                   "\"http\""
                 ],
                 "Inclusion": 3,
                 "Low": [
                   "\"United Kingdom\"",
                   "true",
                   "\"http\""
                 ]
               }
             }
           ],
```

In this case, we push down all of the predicates to the index scan.
The query runs in about **233 milliseconds!**

```
SELECT *
FROM `travel-sample` hotel
WHERE country = "United Kingdom"
AND vacancy = true
AND ANY v IN TOKENS(hotel, {"case":"lower", "name":true}) SATISFIES v = "http"
   END
AND type = "hotel";
Results (partially shows for brevity):
{
   "requestID": "c4a30037-d808-419c-a691-ab783f46866c",
   "signature": {
       "*": "*"
   },
   "results": [
        {
            "hotel": {
                "address": "Capstone Road, ME7 3JE",
                "alias": null,
                "checkin": null,
                "checkout": null,
                "city": "Medway",
                "country": "United Kingdom",
                "description": "40 bed summer hostel about 3 miles from
Gillingham, housed in a districtive converted Oast House in a semi-rural
setting.",
                "directions": null,
                "email": null,
                "fax": null,
                "free_breakfast": true,
                "free_internet": false,
                "free_parking": true,
                "geo": {
                    "accuracy": "RANGE_INTERPOLATED",
                    "lat": 51.35785,
                    "lon": 0.55818
                },
                "id": 10025,
                "name": "Medway Youth Hostel",
                "pets_ok": true,
                "phone": "+44 870 770 5964",
                "price": null,
                "public_likes": [
                    "Julius Tromp I",
                    "Corrine Hilll",
                    "Jaeden McKenzie",
                    "Vallie Ryan",
                    "Brian Kilback",
                    "Lilian McLaughlin",
                    "Ms. Moses Feeney",
                    "Elnora Trantow"
                ],
                "reviews": [
                    {
                        "author": "Ozella Sipes",
                        "content": "This was our 2nd trip here and we enjoyed it
as much or more than last year. Excellent location across from the French
```

```
Market and just across the street from the streetcar stop. Very convenient to
several small but good restaurants. Very clean and well maintained.
Housekeeping and other staff are all friendly and helpful. We really enjoyed
sitting on the 2nd floor terrace over the entrance and \"people-watching\" on
Esplanade Ave., also talking with our fellow guests. Some furniture could use a
little updating or replacement, but nothing major.",
                        "date": "2013-06-22 18:33:50 +0300",
                        "ratings": {
                            "Cleanliness": 5,
                            "Location": 4,
                            "Overall": 4,
                            "Rooms": 3,
                            "Service": 5,
                            "Value": 4
                        }
                    },
                    {
                        "author": "Barton Marks",
                        "content": "We found the hotel de la Monnaie through
Interval and we thought we'd give it a try while we attended a conference in
New Orleans. This place was a perfect location and it definitely beat staying
downtown at the Hilton with the rest of the attendees. We were right on the
edge of the French Quarter withing walking distance of the whole area. The
location on Esplanade is more of a residential area so you are near the fun but
far enough away to enjoy some quiet downtime. We loved the trolly car right
across the street and we took that down to the conference center for the
conference days we attended. We also took it up Canal Street and nearly
delivered to the WWII museum. From there we were able to catch a ride to the
Garden District - a must see if you love old architecture - beautiful old
homes(mansions). We at lunch ate Joey K's there and it was excellent. We ate so
many places in the French Quarter I can't remember all the names. My husband
loved all the NOL foods - gumbo, jambalya and more. I'm glad we found the
Louisiana Pizza Kitchen right on the other side of the U.S. Mint (across the
street from Monnaie). Small little spot but excellent pizza! The day we arrived
was a huge jazz festival going on across the street. However, once in our
rooms, you couldn't hear any outside noise. Just the train at night blowin it's
whistle! We enjoyed being so close to the French Market and within walking
distance of all the sites to see. And you can't pass up the Cafe du Monde down
the street - a busy happenning place with the best French dougnuts!!!Delicious!
We will defintely come back and would stay here again. We were not hounded to
purchase anything. My husband only received one phone call regarding timeshare
and the woman was very pleasant. The staff was laid back and friendly. My only
```

```
complaint was the very firm bed. Other than that, we really enjoyed our stay.
Thanks Hotel de la Monnaie!",
                       "date": "2015-03-02 19:56:13 +0300",
                       "ratings": {
                           "Business service (e.g., internet access)": 4,
                           "Check in / front desk": 4,
                           "Cleanliness": 4,
                           "Location": 4,
                           "Overall": 4,
                           "Rooms": 3,
                           "Service": 3,
                           "Value": 5
                       }
                   }
               ],
               "state": null,
               "title": "Gillingham (Kent)",
               "tollfree": null,
               "type": "hotel",
               "url": "http://www.yha.org.uk",
               "vacancy": true
           }
.....
   "status": "success",
   "metrics": {
       "elapsedTime": "233.475262ms",
       "executionTime": "233.397438ms",
       "resultCount": 185,
       "resultSize": 1512687
   }
}
```

TOKENS() Function

This function is explained in detail in Couchbase documentation. The important thing to note is that first expression passed into token can be anything: constant literal, simple JSON value, JSON key name or the whole document itself.

TOKENS (Expression, Options)

You can invoke TOKENS() simply with the expressions or with the option. The function always returns an array of tokens it extracted for the given (implied) options.

TOKENS(address)

This returns an array of values including the attribute (key) names. It retrieves the data in their basic types and returns
Options can take the following options. Each invocation of TOKENS() can choose one or more of the options, passed in as JSON.
`{"name": true}`: Valid values are true or false. Default is true.
`{"case":"lower"}`: Valid values are "upper" and "lower". Default is neither — return the case in the data.
`{"specials": true}`: Preserves strings with special characters such as email addresses, URLs, and hyphenated phone numbers. The default is false.
Let's use all of the options on TOKENS() to see what it produces.

Query:

```
SELECT
TOKENS ({"info":"Name is keshav email is keshav@couchbase.com Blog is
http://planetnosql.com twitter handle is @rkeshavmurthy"},
{"specials":true, "names":true, "case":"lower"}) AS words
Result:
[
 {
   "words": [
     "info",
     "planetnosql",
     "http://planetnosql.com",
     "rkeshavmurthy",
     "name",
     "@rkeshavmurthy",
     "com",
     "http",
     "couchbase",
     "blog",
     "twitter",
     "is",
     "keshav",
     "handle",
     "email",
     "keshav@couchbase.com"
   ]
 }
]
```

With names set to true, you'll get the field name, in this case info.
With specials set to true, you'll get delimited words as well as special works like keshav@couchbase.com or @rkeshavmurthy as a single word to index.
With case set to lower, all of the results will be in the lower case
Let's look at an example usage on the travel-sample data.

```
SELECT TOKENS(url) AS defaulttoken,
      TOKENS(url, {"specials":true, "case":"UPPER"}) AS specialtoken
FROM `travel-sample` h WHERE h.type = 'hotel'
LIMIT 1;
[
  {
    "defaulttoken": [
      "uk",
      "www",
      "http",
      "yha",
      "org"
    ],
    "specialtoken": [
      "ORG",
      "WWW",
      "HTTP://WWW.YHA.ORG.UK",
      "YHA",
      "UK",
      "HTTP"
    ]
  }
]
```

In this case, TOKENS() converted all of the URL data into UPPER case and added the full URL in addition to delimited words.
Use {"case":"lower"} or {"case":"upper"} to have case insensitive search. Index creation and querying can use this and other parameters in combination. As you saw in the examples, these parameters should be passed within the query predicates as well. The parameters and values have to match exactly for N1QL to pick up and use the index correctly.
Here is an example of how you can create the index and use it your application.

```
Index definition:
CREATE INDEX idx_url_upper_special on `travel-sample`(
 DISTINCT ARRAY v for v in
      tokens(url, {"specials":true, "case":"UPPER"})
 END ) where type = 'hotel' ;
Query:
SELECT name, address, url
FROM `travel-sample` h
WHERE ANY  v in tokens(url, {"specials":true, "case":"UPPER"})
    SATISFIES v = "HTTP://WWW.YHA.ORG.UK"
   END
AND h.type = 'hotel' ;
Results:
[
 {
   "address": "Capstone Road, ME7 3JE",
   "name": "Medway Youth Hostel",
   "url": "http://www.yha.org.uk"
 }
]
Query Plan:
      {
         "#operator": "DistinctScan",
         "scan": {
           "#operator": "IndexScan",
           "index": "idx_url_upper_special",
           "index_id": "ec73b96583ac2edf",
           "keyspace": "travel-sample",
           "namespace": "default",
           "spans": [
             {
               "Range": {
                 "High": [
                   "\"HTTP://WWW.YHA.ORG.UK\""
                 ],
                 "Inclusion": 3,
                 "Low": [
                   "\"HTTP://WWW.YHA.ORG.UK\""
                 ]
               }
             }
           ],
```

Summary

Couchbase 4.6 has introduced a new way to split the JSON document into simpler values using TOKENS(). Using the flexible array indexing in Couchbase, you can now create secondary indexes for scalar and array values. This is a high-performance alternative to the LIKE predicate. Its performance is so much more likable.

References

1. Couchbase : http://www.couchbase.com
2. N1QL: http://query.couchbase.com
3. Couchbase Full Text Search: http://developer.couchbase.com/documentation/server/current/fts/full-text-intro.html
4. SPLIT and CONQUER: https://dzone.com/articles/split-and-conquer-efficient-string-search-with-n1q
5. A Couchbase Index Technique for LIKE Predicates With Wildcard: https://dzone.com/articles/a-couchbase-index-technique-for-like-predicates-wi

SPLIT and CONQUER: Efficient String Search With N1QL in Couchbase

Author: Keshav Murthy

Consider my DZone article: Concurrency Behavior: MongoDB vs. Couchbase. It's a 10+ page article. For any application, indexing and supporting searching within the full text is a big task. The best practice for search is to have tags or labels for each article and then search on those tags. For this article, tags are: CONCURRENCY, MONGODB, COUCHBASE, INDEX, READ, WRITE, PERFORMANCE, SNAPSHOT, and CONSISTENCY.

Let's put this into a JSON document.

```
Document Key: "k3"
{
"tags": "CONCURRENCY,MONGODB,COUCHBASE,INDEX,READ,WRITE,PERFORMANCE,SNAPSHOT,CONSISTENCY",
"title": "Concurrency Behavior: MongoDB vs. Couchbase"
}
```

It's one one thing to store the tags. It's another thing to search for a tag within the string. How do you search for the COUCHBASE tag within the tags? How do you search for a specific string within this "tags"?

In SQL, you can do the following:

```
SELECT title, url FROM articles WHERE tags LIKE "%COUCHBASE%";
Query Explain:
                  "~children": [
                       {
                           "#operator": "PrimaryScan",
                           "index": "#primary",
                           "keyspace": "articles",
                           "namespace": "default",
                           "using": "gsi"
                       },
```

Create an index on tags, you get the following plan:

```
CREATE INDEX idx_articles_tags on articles(tags);
EXPLAIN SELECT title, url FROM articles WHERE tags LIKE "%COUCHBASE%";
Query Plan:
             "#operator": "Sequence",
              "~children": [
                  {
                      "#operator": "IndexScan",
                      "index": "idx_articles_tags",
                      "index_id": "ca25a044d8383f1",
                      "keyspace": "articles",
                      "namespace": "default",
                      "spans": [
                          {
                              "Range": {
                                  "High": [
                                      "[]"
                                  ],
                                  "Inclusion": 1,
                                  "Low": [
                                      "\"\""
                                  ]
                              }
                          }
                      ],
```

You need to use the pattern “%COUCHBASE%” because the string pattern could be anywhere within the tags string. Whether you do a primary scan (table scan) or an index scan, this query is going to be slow on a large data set, as it has to scan the entire data set or the index to correctly evaluate the predicate.

Two observations here:

1. While the predicate "%COUCHBASE%" looks for the word COUCHBASE anywhere in the string, it's usually in there as a separate word, tag or label.
2. Applications and users typically search for the whole word.

Now, let's see how N1QL makes this fast, easy, and efficient.

Let’s first look at the SPLIT() function. SPLIT() can take any string and a separator, and return array strings.

```
SELECT SPLIT("CONCURRENCY,MONGODB,COUCHBASE,INDEX,READ,WRITE,PERFORMANCE,SNAPSHOT,CONSISTENCY",
",");
   "results": [
       {
           "$1": [
               "CONCURRENCY",
               "MONGODB",
               "COUCHBASE",
               "INDEX",
               "READ",
               "WRITE",
               "PERFORMANCE",
               "SNAPSHOT",
               "CONSISTENCY"
           ]
       }
   ]
```

Once you have an array, you can query the data based on array predicates.

Note: System:dual always has a single document with null in it.

```
SELECT COUNT(*) boolcount
FROM system:dual
```

```
WHERE
ANY s in
SPLIT("CONCURRENCY,MONGODB,COUCHBASE,INDEX,READ,WRITE,PERFORMANCE,SNAPSHOT,CONS
ISTENCY", ",")
   SATISFIES s = "COUCHBASE"
END;
[
 {
   "boolcount": 1
 }
]
```

Let’s try to find something that doesn’t exist. Since the predicate is evaluated to be False, nothing is returned.

```
SELECT COUNT(*) boolcount
FROM system:dual
WHERE
ANY s in
SPLIT("CONCURRENCY,MONGODB,COUCHBASE,INDEX,READ,WRITE,PERFORMANCE,SNAPSHOT,CONS
ISTENCY", ",")
   SATISFIES s = "TRANSACTION"
END;
[
 {
   "boolcount": 0
 }
]
```

We can convert the tags string into an array of strings and then lookup the value in the array instead of searching within the whole string.

Now, we've converted the problem from string search to array look up.

Couchbase 4.5 has a flexible array indexing capability to index arrays. Exploiting this, we can look up the value in the array index very efficiently.

Here's the approach:

Let's try this end-to-end example on Couchbase 4.5 or above.

1. Create a bucket called **articles**.
2. CREATE primary index

```
CREATE PRIMARY INDEX ON articles;
```

3. INSERT the following documents.

```
INSERT INTO articles(KEY, VALUE) VALUES(“k1”,
{ "tags": "JSON,N1QL,COUCHBASE,BIGDATA,NAME,data.gov,SQL",
"title": "What's in a New York Name? Unlock data.gov Using N1QL "
}),
VALUES(“k2”,
       {
           "tags": "TWITTER,NOSQL,SQL,QUERIES,ANALYSIS,HASHTAGS,JSON,COUCHBASE,ANALYTICS,INDEX",
           "title": "SQL on Twitter: Analysis Made Easy Using N1QL"
       }),
VALUES(“k3”,
       {
           "tags":
"CONCURRENCY,MONGODB,COUCHBASE,INDEX,READ,WRITE,PERFORMANCE,SNAPSHOT,CONSISTENCY",
           "title": "Concurrency Behavior: MongoDB vs. Couchbase"
       }),
VALUES(“k4”,
       {
           "tags": "COUCHBASE,N1QL,JOIN,PERFORMANCE,INDEX,DATA MODEL,FLEXIBLE,SCHEMA",
           "title": "JOIN Faster With Couchbase Index JOINs"
       }),
VALUES (“k5”,
       {
           "tags": "NOSQL,NOSQL
BENCHMARK,SQL,JSON,COUCHBASE,MONGODB,YCSB,PERFORMANCE,QUERY,INDEX",
           "title": "How Couchbase Won YCSB"
       });
```

4. CREATE the ARRAY INDEX

```
CREATE INDEX `itagarray` ON `articles`
((DISTINCT (ARRAY `wrd` FOR `wrd` IN SPLIT(`tags`, ",") END)))
```

5. Let’s query the data now.

```
EXPLAIN SELECT articles.*
FROM articles
WHERE ANY wrd in SPLIT(tags, ",") SATISFIES wrd = "COUCHBASE" END;
```

This explain shows the array index is used and the spans created for the index scan.

```
                    {
                        "#operator": "DistinctScan",
                        "scan": {
                            "#operator": "IndexScan",
                            "index": "itagarray",
                            "index_id": "266223a6846cac2b",
                            "keyspace": "articles",
                            "namespace": "default",
                            "spans": [
                                {
                                    "Range": {
                                        "High": [
                                            "\"COUCHBASE\""
                                        ],
                                        "Inclusion": 3,
                                        "Low": [
                                            "\"COUCHBASE\""
                                        ]
                                    }
                                }
                            ],
                            "using": "gsi"
                        }
                    },
```

Let’s get greedy. Now that we have an efficient way to look up the tags, let’s make it case insensitive.

1. Create the array index, but convert the data into lower case before you index it.

```
CREATE INDEX `itagarraylower`
  ON `articles`
(DISTINCT ARRAY `wrd` FOR `wrd` IN SPLIT(LOWER(`tags`), ",") END);
```

2. Start querying.

```
EXPLAIN SELECT articles.*
FROM articles
WHERE ANY wrd in SPLIT(LOWER(tags), ",") SATISFIES wrd = "couchbase" END;
               "~children": [
                   {
                       "#operator": "DistinctScan",
                       "scan": {
                           "#operator": "IndexScan",
                           "index": "itagarraylower",
                           "index_id": "3ace3e48e2124ffe",
                           "keyspace": "articles",
                           "namespace": "default",
                           "spans": [
                               {
                                   "Range": {
                                       "High": [
                                           "\"couchbase\""
                                       ],
                                       "Inclusion": 3,
                                       "Low": [
                                           "\"couchbase\""
                                       ]
                                   }
                               }
                           ],
                           "using": "gsi"
                       }
                   },
```

You can even put a LOWER expression on the tag you’re looking for to make the query generalized.

```
You can even put a LOWER expression on the tag you’re looking for to make the query generalized.
EXPLAIN SELECT articles.*
FROM articles
WHERE
   ANY wrd in SPLIT(LOWER(tags), ",")
       SATISFIES wrd = LOWER("couchbase")
   END;
```

More Use Cases

Tags aren’t the ONLY type of data stored in regular string pattern. Tags are NOT the only ones to benefit from the technique. Here are some other examples:

- Airline route information

“SFO:LAX:FLG:PHX:HOU:MCO”

- Variable list of employee skills

“Java,C,C++,Python,perl”

- CSV processing: Handling data import from CSV.

3RJA043,Acura Integra,Honda,GSR,172,189482,1996,,California

- Hashtags search

#SQL#NoSQL#JSON#appdev#database#Optimizer#indexing

Conclusion

N1QL is designed to handle the flexible schema and structure of JSON. Array indexing helps you to process and exploit arrays in useful ways.

Performance Ingredients for NoSQL: Intersect Scans in N1QL

Author: Prasad Varakur

Couchbase Server N1QL Query Engine is SQL for JSON

It’s not an exaggeration to say that query processing has produced tons of papers and patents. It is one of the most complex and complicated areas in the database realm. With the proliferation of NoSQL/document databases and more adoption into the enterprise, there is high demand and a requirement for sophisticated query processing capabilities for NoSQL beyond k/v get/set accesses. Remember, the JSON data model is the foundation of document databases that brings the most value to NoSQL (among others like scale-out, HA, and performance) with its fundamentally different capabilities (w.r.t RDBMS) such as flexible schema and hierarchical and multi-valued data.

In this blog, I will try to explain some of the performance ingredients and optimization techniques used for efficient JSON query processing. More specifically, I will cover intersect scans, which leverages multiple indexes for a query and brings subtle performance characteristics. For simplicity, I will use the latest Couchbase Server 4.6 N1QL engine and the packaged travel-sample data for examples. Note that N1QL is the most sophisticated query engine and the most comprehensive implementation of SQL (in fact, SQL++) for JSON data model.

Fundamentals of Performance

The performance of any system follows physics. The basic two rules can be (loosely) stated as:

1. Quantity: Less work is more performance.
2. Quality: Faster work is more performance.

Query processing is no different and it also tries to optimize both these factors in various forms and scenarios to bring efficiency. Each optimization is different and results in a different amount of performance benefit. As the saying goes, every nickel adds to a dollar, and the N1QL engine keeps incorporating performance enhancements to run queries as efficiently as possible.

Intersect Scans: Using Multiple Indexes for AND Queries

As you might already know, secondary indexes (just "indexes" for short) are the biggest performance techniques used by query processing. Database indexes help with efficiently mapping and finding document attributes to documents. For example, with user profile employee documents, a key-value store can fetch a particular document only if the corresponding document_key is available. However, creating an index on the `name` attribute of the document maintains mapping of all `name` values to the set of documents holding a specific `name` value. This is called index-lookup or IndexScan operation.

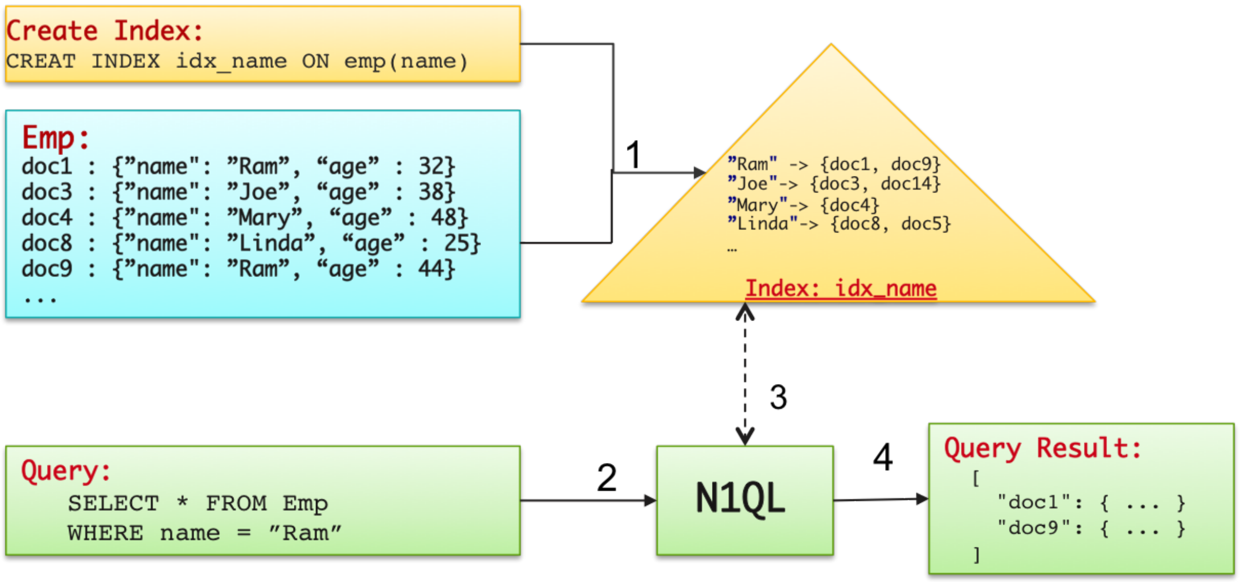


Figure1: Processing simple query with matching secondary index

When one index satisfies lookup for a query predicate, it is called a simple IndexScan. However, when a query has multiple AND predicate conditions, then different available indexes may satisfy different conditions in the WHERE-clause. In such case, N1QL uses the technique called Intersect Scan, where both the indexes are used to find documents matching the predicates. The technique is based on the premise that the intersection of the sets of documents obtained by IndexScan lookups from each of the satisfying indexes is equal to the logical-AND of applying respective individual predicates. Note that this is related, but very different from using composite indexes, where a single index is created on multiple fields used in different predicates of the query. Will cover it later in the blog.

Let's understand this with some simpler example. Consider following query which has predicates on `name` and `age` , and also assume we have two indexes created on the two fields used in the WHERE-clause.

```
CREATE INDEX idx_name ON emp(name);
CREATE INDEX idx_age ON emp(age);
SELECT * FROM emp WHERE name = “Ram” and age < 35;
```

In this case, both the indexes can satisfy the query predicate, but neither can fully help the two predicate conditions. By using simple IndexScan, the N1QL query engine can do following:

- Use the `name` index *idx_name*:
    - In this case, index *idx_name* is looked up to first find all the documents keys which match `name = “Ram”`.
        - Then, respective documents are fetched and the second filter `age < 35` is applied
        - Finally, query returns the set of documents which matched both the filers.
- Use the `age` index *idx_age*.
    - In this case, the index *idx_age* is looked up first, to find all the document keys which match `age < 35`.
    - Then, respective documents are fetched and the second filter `name = “Ram”` is applied.
    - Finally, the query returns the set of documents which matched both the filers.

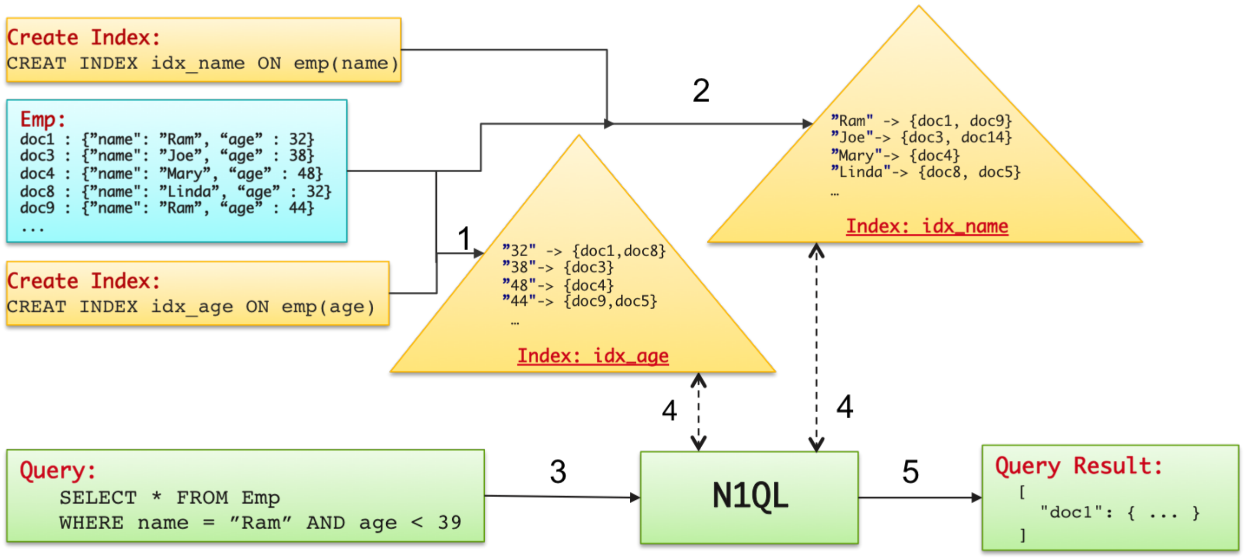
Both approaches make sense and produce correct results. However, they have very significant and subtle performance characteristics. The performance depends on a property called selectivity of the predicate conditions, which is defined as the ratio of the number of documents matching particular predicate to the total number of documents considered. For instance:

- If the there 100 documents in the `emp` bucket and five employees have name "Ram," then the selectivity of the predicate `(name = ‘Ram’)` is 5/100 = 0.05 (or 5% for simplicity).
- Similarly, if 40% of the 100 employees are aged less than 35, then the selectivity of `(age < 35)` is 0.4.
- Finally, the selectivity of the conjunctive predicate will be <= min(0.05, 0.4).

Now you might be wondering what these selectivities have anything to do with the query performance. Let’s connect the dots by looking at the two IndexScan approaches we talked above:

- Use the name index idx_name.
  - Index lookup finds five documents matching (name = ‘Ram’) and returns the corresponding document keys to N1QL.
  - The five documents are fetched, and second filter (age < 35) is applied. Assume two documents satisfy this condition (i.e., selectivity of both predicates together is 0.02 or 2%).
  - The two documents matching both the filters are returned as the final result.
- Use the age index *idx_age*.
  - Index lookup finds 40 documents matching (age < 35) and returns the corresponding 40 document keys to N1QL.
  - The 40 documents are fetched, and second filter (name = ‘Ram’) is applied. Only two documents will satisfy this condition.
  - The two documents matching both the filters are returned as the final result.

Now, it’s very apparent that the first approach using the index *idx_name* is much more efficient than the second approach of using *idx_age*. The first approach is doing less work in each of the steps. In particular, fetching five documents instead of 40 is huge!



```
Create Index:
CREAT INDEX idx_name ON emp(name)

Emp:
doc1 : {"name": "Ram", "age" : 32}
doc3 : {"name": "Joe", "age" : 38}
doc4 : {"name": "Mary", "age" : 48}
doc8 : {"name": "Linda", "age" : 32}
doc9 : {"name": "Ram", "age" : 44}
...

Create Index:
CREAT INDEX idx_age ON emp(age)

Query:
   SELECT * FROM Emp
   WHERE name = "Ram" AND age < 39

Query Result:
  [
    "doc1": { ... }
  ]
```

```
"Ram" -> {doc1, doc9}
"Joe"-> {doc3, doc14}
"Mary"-> {doc4}
"Linda"-> {doc8, doc5}
...

"32" -> {doc1,doc8}
"38"-> {doc3}
"48"-> {doc4}
"44"-> {doc9,doc5}
...
```

Figure2: Processing conjunctive query with IntersectScan on two indexes

We can generalize this and say it is always efficient to evaluate predicates with lower selectivity values, followed by higher selectivities. Relational databases understand these factors very well, and their cost-based query optimizers maintain enough statistics to figure the best index for the query. N1QL is a rule-based query optimizer and solves this problem smartly using Intersect Scans to both the indexes. The steps are:

1. Send parallel request to both indexes with respective predicates.
2. Index lookup of idx_name with (name = ‘Ram’), which returns five document keys.
3. Index lookup of idx_age with (age < 35), which returns 40 document keys.
4. The intersection of the two sets of the keys is performed, which returns the two document keys that should satisfy both the filters.
5. The two documents are fetched, predicates are reapplied (to cover any potential inconsistencies between index and data), and final results are returned. Note that, for performance reasons, indexes are maintained asynchronously to reflect any data or document changes. N1QL provides various consistency levels that can be specified per query.

Performance Characteristics of IntersectScan

The following observations can be made from the above five steps of how IntersectScan's work:

- IntersectScan can be more performant when the final result of the conjunctive predicate is much smaller than that of individual filters.
- IntersectScan does index lookups to all satisfying indexes. This may be unnecessary, for example, when we know a specific index is best, or when some index lookups don't filter any documents (such as duplicate indexes, or when an index is subset of another). So, when we have a lot of satisfying indexes, InterSect scans can be more expensive (than using a specific index).
- The Subset of satisfying indexes can be provided to the query using the USE INDEX (idx1, idx2, ...) clause, to override N1QL default behavior of picking all satisfying indexes for the IntersectScan.
- As there are pros and cons of using IntersectScans, always check the EXPLAIN output of your performance sensitive queries and make sure if the IntersectScans are used appropriately.

- Look at my blog on N1QL performance improvements in Couchbase Server 4.6 release for few such subtle optimizations related to IntersectScans.

Comparing With Composite Indexes: Flexibility vs. Performance

As it is clear by now, the purpose of IntersectScans is to efficiently run queries with conjunctive queries. Composite indexes are an excellent alternative to doing the same, and yes, there are trade-offs. A composite index is an index with multiple fields as index-keys. For example, the following index is created with `name` and `age`:

```
CREATE INDEX idx_name_age ON emp(name, age);
```

This index can very well fully satisfy the query discussed earlier and perform as well as (or slightly better) than the IntersectScan. However, this index cannot help a query with a filter on just `age` field, such as:

```
Q1: SELECT * FROM emp WHERE age = 35
```

That will violate the prefix-match constraint of the composite indexes. That means that to use a composite index, all the prefixing index-keys/fields must be specified as part of the WHERE-clause. In this example, `name` must be used in the WHERE-clause to be able to use the index *idx_name_age*. as in:

```
Q2: SELECT * FROM emp WHERE age = 35 AND name IS NOT MISSING;
```

There are two problems with this rewritten query Q2:

1. Q2 is not equivalent to Q1. For example, Q1 may return more documents which has a missing `name` field.

2. The selectivity of the spurious predicate `(name IS NOT MISSING)` may be really high, thus pretty much converting the indexScan into a full scan. That means the index lookup will go through all index entries which have non-missing `name`, and then subsequently check for`(age = 35)`. In contrast, when using the index *idx_age*, the index entry for `(age = 35)` is a straight point-lookup and returned instantaneously.

Further, it is important to understand that the size of the composite index gets larger with the number of index keys. Lookups/scans on a bigger index will be more expensive, than those on a smaller index (on one field). To summarize:

- Composite indexes are a better choice to optimize specific queries.
- Simple indexes (on one or more fields), along with IntersectScans are a better choice for ad hoc queries which need the flexibility of conjunctive predicates on various fields.

Example

Let's look at a live example with travel-sample data and precreated indexes `def_type` and `def_sourceairport` shipped with the product. Ran these tests with Couchbase Server 4.6 DP release on my Mac Pro laptop. The query finds the destination airports and number of stops/hops, starting from SFO.

Use the index def_type:

```
EXPLAIN SELECT destinationairport, stops
FROM `travel-sample`
USE INDEX (def_type)
WHERE type = "route" AND sourceairport = "SFO";
[
 {
   "plan": {
     "#operator": "Sequence",
     "~children": [
       {
         "#operator": "IndexScan",
         "index": "def_type",
         "index_id": "e23b6a21e21f6f2",
         "keyspace": "travel-sample",
         "namespace": "default",
         "spans": [
           {
             "Range": {
               "High": [
                 "\"route\""
               ],
               "Inclusion": 3,
               "Low": [
                 "\"route\""
               ]
             }
           }
         ],
         "using": "gsi"
       },
       {
         "#operator": "Fetch",
         "keyspace": "travel-sample",
         "namespace": "default"
       },
       {
         "#operator": "Parallel",
         "~child": {
           "#operator": "Sequence",
           "~children": [
             {
               "#operator": "Filter",
               "condition": "(((`travel-sample`.`type`) = \"route\") and
((`travel-sample`.`sourceairport`) = \"SFO\"))"
             },
             {
               "#operator": "InitialProject",
               "result_terms": [
                 {
                   "expr": "(`travel-sample`.`destinationairport`)"
                 },
                 {
                   "expr": "(`travel-sample`.`stops`)"
                 }
               ]
             },
             {
               "#operator": "FinalProject"
             }
           ]
         }
       }
     ]
   },
   "text": "SELECT destinationairport, stops\nFROM `travel-sample`\nUSE INDEX
(def_type) \nWHERE type = \"route\" AND sourceairport = \"SFO\";"
 }
]
```

This query took on average 650msec. Note that the selectivity of `(type = 'route')` is `0.76 (i.e., 24,024 of 31,592 docs).`

```
SELECT (SELECT RAW count(*) FROM `travel-sample` t where type = "route")[0]/
      (SELECT RAW count(*) FROM `travel-sample` t1)[0] as selectivity;
[
 {
   "selectivity": 0.7604456824512534
 }
]
```

Similarly, the selectivity of `(sourceairport = 'SFO')` is ~0.0079 (i.e., 249 of 31,592 docs).

```
SELECT (SELECT RAW count(*) FROM `travel-sample` t where sourceairport =
"SFO")[0]/
      (SELECT RAW count(*) FROM `travel-sample` t1)[0]  as selectivity;
[
 {
   "selectivity": 0.007881742213218537
 }
]
```

Hence, using the index *def_sourceairport* should be much more efficient.

Use the index *def_sourceairport*:

```
EXPLAIN SELECT destinationairport, stops
FROM `travel-sample`
USE INDEX (def_sourceairport)
WHERE type = "route" AND sourceairport = "SFO";
[
 {
   "plan": {
     "#operator": "Sequence",
     "~children": [
       {
         "#operator": "IndexScan",
         "index": "def_sourceairport",
         "index_id": "c36ffb6c9739dcc9",
         "keyspace": "travel-sample",
         "namespace": "default",
         "spans": [
           {
             "Range": {
               "High": [
                 "\"SFO\""
```

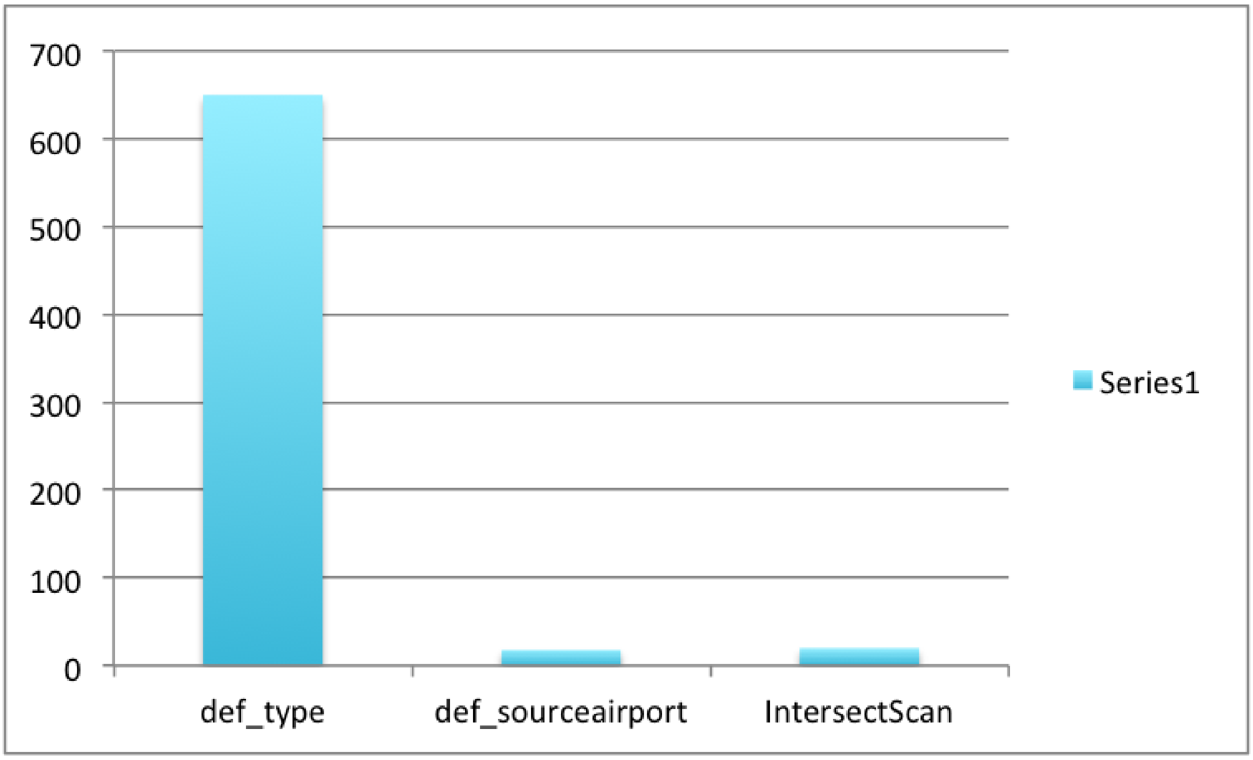
```
                ],
                "Inclusion": 3,
                "Low": [
                  "\"SFO\""
                ]
              }
            }
          ],
          "using": "gsi"
        },
        {
          "#operator": "Fetch",
          "keyspace": "travel-sample",
          "namespace": "default"
        },
        {
          "#operator": "Parallel",
          "~child": {
            "#operator": "Sequence",
            "~children": [
              {
                "#operator": "Filter",
                "condition": "(((`travel-sample`.`type`) = \"route\") and
((`travel-sample`.`sourceairport`) = \"SFO\"))"
              },
              {
                "#operator": "InitialProject",
                "result_terms": [
                  {
                    "expr": "(`travel-sample`.`destinationairport`)"
                  },
                  {
                    "expr": "(`travel-sample`.`stops`)"
                }
              ]
            },
            {
              "#operator": "FinalProject"
            }
          ]
        }
      }
    ]
  },
  "text": "SELECT destinationairport, stops\nFROM `travel-sample`\nUSE INDEX
(def_sourceairport) \nWHERE type = \"route\" AND sourceairport = \"SFO\";"
 }
]
```

As expected, this query ran much more efficiently, and took (on average) 18msec. If no specific index is used (or multiple indexes are specified), N1QL does IntersectScan with all available satisfying indexes:

```
EXPLAIN SELECT destinationairport, stops
FROM `travel-sample`
USE INDEX (def_sourceairport, def_type)
WHERE type = "route" AND sourceairport = "SFO";
[
 {
   "plan": {
     "#operator": "Sequence",
     "~children": [
       {
         "#operator": "IntersectScan",
         "scans": [
           {
             "#operator": "IndexScan",
    "index": "def_type",
    "index_id": "e23b6a21e21f6f2",
    "keyspace": "travel-sample",
    "namespace": "default",
    "spans": [
      {
        "Range": {
          "High": [
            "\"route\""
          ],
          "Inclusion": 3,
          "Low": [
            "\"route\""
          ]
        }
      }
    ],
    "using": "gsi"
  },
  {
    "#operator": "IndexScan",
    "index": "def_sourceairport",
    "index_id": "c36ffb6c9739dcc9",
    "keyspace": "travel-sample",
    "namespace": "default",
    "spans": [
      {
        "Range": {
          "High": [
            "\"SFO\""
          ],
          "Inclusion": 3,
          "Low": [
                    "\"SFO\""
                  ]
                }
              }
            ],
            "using": "gsi"
          }
        ]
      },
      {
        "#operator": "Fetch",
        "keyspace": "travel-sample",
        "namespace": "default"
      },
      {
        "#operator": "Parallel",
        "~child": {
          "#operator": "Sequence",
          "~children": [
            {
              "#operator": "Filter",
              "condition": "(((`travel-sample`.`type`) = \"route\") and
((`travel-sample`.`sourceairport`) = \"SFO\"))"
            },
            {
              "#operator": "InitialProject",
              "result_terms": [
                {
                  "expr": "(`travel-sample`.`destinationairport`)"
                },
                {
                  "expr": "(`travel-sample`.`stops`)"
                }
              ]
            },
            {
              "#operator": "FinalProject"
            }
          ]
        }
      }
    ]
  },
  "text": "SELECT destinationairport, stops\nFROM `travel-sample`\nUSE INDEX
(def_sourceairport, def_type) \nWHERE type = \"route\" AND sourceairport =
\"SFO\";"
 }
]
```

This query also ran very efficiently in 20msec.

Summary



Run times with the individual indexes and IntersectScan are shown in this chart. Note that the latency with the IntersectScan is almost same as that with the index *def_sourceairport* because the selectivities of the conjunctive predicate is the same as that of `(sourceairport = 'SFO')`. Basically, the `soruceairport` field exists only in the `type = 'route'` documents in `travel-sample`.

Hope this gives some insights into the intricacies involved in achieving query performance and the interplay between various factors such as the query, predicates, selectivities, and index-type.

Couchbase Server 4.6 Developer Preview is available now. It has a bunch of cool performance optimizations in N1QL. Try it out and let me know any questions, or just how awesome it is.

The Latest Performance Tricks in N1QL

Author: Prasad Varakur

Boost N1QL Queries With Couchbase 4.5.1

In this article, I will share few awesome N1QL performance enhancements available in Couchbase Server 4.5.1. While some of these generically help all N1QL queries, a few improvements are specific in nature and significantly boost the performance of some kind of queries. For each of the improvements, I briefly explain the feature and show some numbers.
My earlier article Apps in N1QL Land: Magic With Pretty, Suffix, Update, and More, talks about salient N1QL enhancements in 4.5.1 release whose performance characteristics will be explored here. More N1QL features are described in this blog
n1ql-functionality-enhancements-in-couchbase-server-4.5.1

Concurrent Bulk Fetch Processing

As with any mature query processing engine, N1QL processes queries in multiple phases, as shown in Figure 1. N1QL also parallelizes many phases whenever possible (indicated by the four boxes/arrows in Figure 1). In this query processing, one of the expensive phases is Fetch, in which the query engine needs to retrieve all the relevant and matching documents from data service nodes for further processing. This occurs when the query does not have a satisfying covering index.

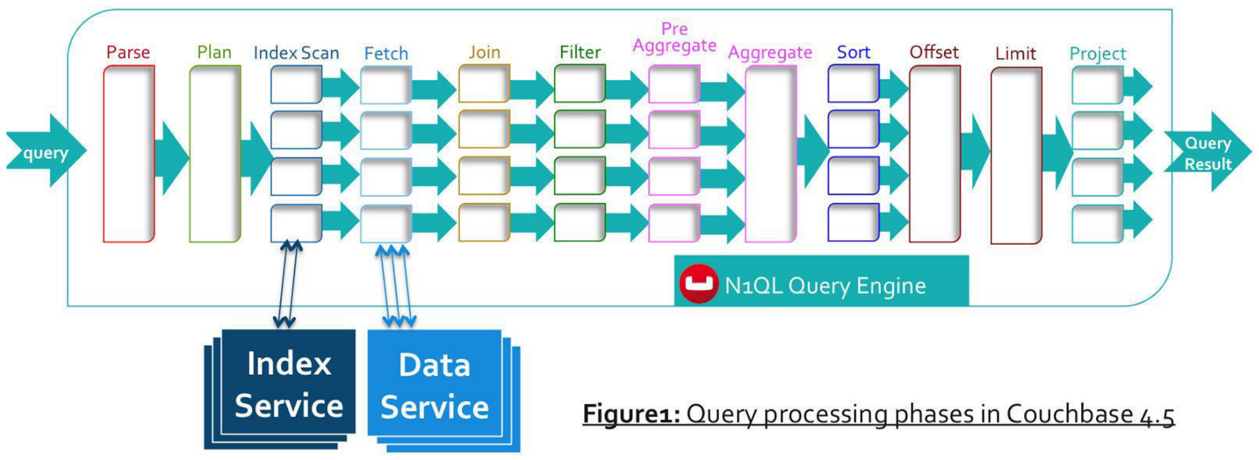


Figure1: Query processing phases in Couchbase 4.5

For instance, consider following query that needs to process all airport-documents. This uses the **travel-sample** bucket that is shipped with Couchbase Server.

```
CREATE INDEX def_type ON `travel-sample`(type);
SELECT * FROM `travel-sample` WHERE type = "airport";
```

While processing this query, N1QL can use the index **def_type** (which is pre-created when the sample bucket is installed) and gets all the document keys matching the predicate. Note that N1QL uses the **predicate-push** down technique to index to retrieve all airport-document keys.
Then, in the Fetch phase, N1QL fetches all the full documents corresponding to the keys obtained in step1.

Prior to Couchbase Server 4.5.1, N1QL does use multiple threads to process the fetch. However, when fetching a lot of documents, especially with many queries bombarding on N1QL, the thread management becomes very important for performance. To get a feel, consider a moderate load of 200 queries per second and each query trying to fetch 1,000 documents. It's a non-trivial task to efficiently create and manage threads to process those 200K fetch requests while being optimally aligned with the available CPU and cores.

In Couchbase Server 4.5.1, N1QL enhanced the management of **fetch-phase** threads to use a common pool of worker threads to process all the fetch requests (see all threads in a box, in Figure 2 below). The number of worker threads is optimally chosen according to the available CPUs so that all fetches are concurrently processed minimizing thread management overhead. This approach uses CPU and memory resources more efficiently, reuses threads, and prevents unnecessary creation and recreation of temporary threads.

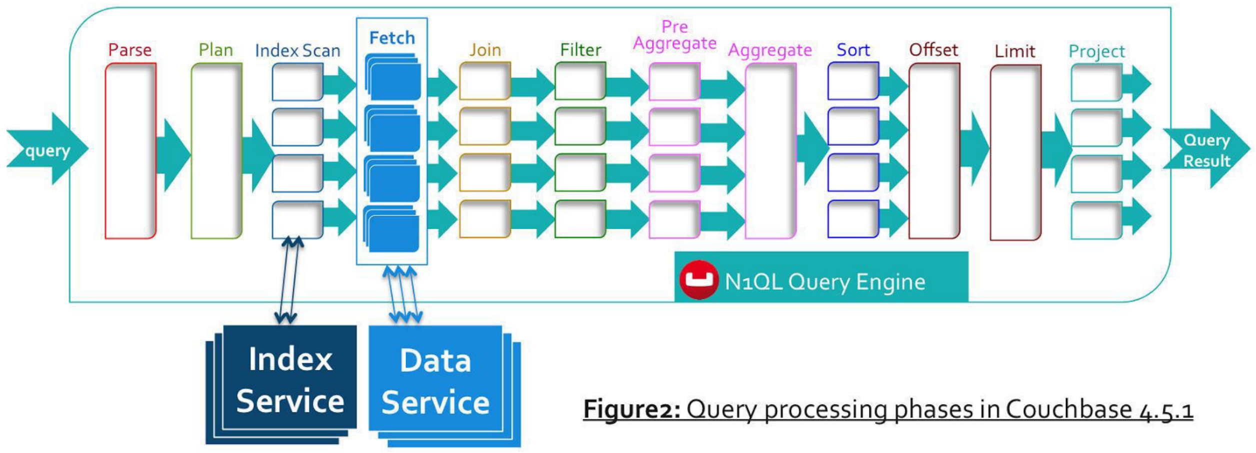
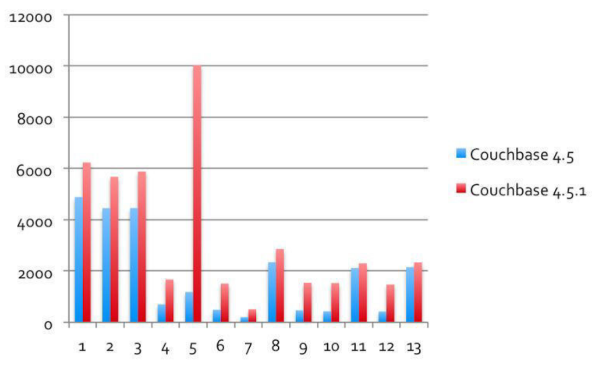


Figure2: Query processing phases in Couchbase 4.5.1

Note that as with earlier releases, N1QL continues to use parallel processing (dictated by configuration parameter max_parallelism) to retrieve documents with multiple parallel fetch requests to data service nodes. In 4.5.1, documents retrieved in each fetch request are concurrently processed using a worker-thread-pool. The thread pool is automatically sized for the available CPUs and cores.



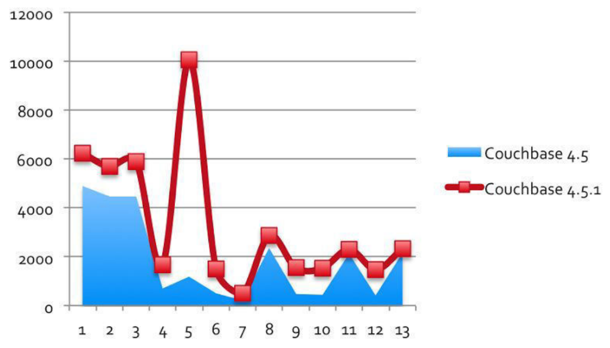


Figure 3: N1QL throughput tests (y-axis is number queries/sec)

This is not just some fancy way of processing the queries, but it shows **real** results. At least, in internal throughput tests, I have noticed a significant performance boost for range queries and some point-queries. The above charts show the performance gain is consistently better, and in some cases, tests got up to eight times throughput (y-axis). Obviously, better performance gains are realized if the query fetches more documents. The x-axis is just the test case number, showing different tests with varying variables such as point/range queries, index storage type (MOI, GSI), scan consistency etc.

In summary, it is an interesting journey to discover the performance hidden in the thread management overheads. Kudos to the smart engineers! Note that the performance numbers vary depending on various factors such as:

- **Type of index storage**, i.e., ForestDB vs. Memory Optimized Indexes.
- **Query scan consistency**, i.e., **not-bounded** or **request_plus**.
- **Query predicates** that dictate the number of document keys returned by the index and hence the number of documents fetched and processed.
- **Operating system**, i.e., Linux, Windows Server, etc

Efficient LIKE ‘%foo%’ Queries

Pattern matching regular expressions is certainly one of the expensive operations in query processing. Typically, LIKE ‘foo%’ queries can be implemented efficiently with a standard index because the first few characters are fixed. However, not very often we can find efficient techniques to process LIKE ‘%foo%’, where the search string contains a leading wildcard.

In Couchbase server 4.5.1, N1QL introduced a novel technique to solve this problem efficiently, by creating an Array Index on all possible suffix substrings of the index key. The new N1QL function SUFFIXES() produces an array of all possible suffixes. For example, the query SELECT name FROM default WHERE name LIKE “%foo%” is rewritten into creating the suffix-array-index and magically replacing the LIKE “%foo%” predicate with LIKE “foo%” (without the leading wildcard). This brings magnitude performance boost to LIKE queries. This smart technique is explained in detail in my earlier blog Apps in N1QL Land: Magic With Pretty, Suffix, Update, and More.

Let’s look at an example showing that the performance gain is real and tangible. In my laptop testing, I got an impressive ~6.5 times faster latency. This uses the **travel-sample** documents and bucket shipped with Couchbase Server.

Using the normal index def_type, queries are run using the Couchbase query CBQ shell.

```
cbq> SELECT * FROM `travel-sample` USE INDEX(def_type)
   > WHERE type = "landmark" AND title LIKE "%land%";
   ....
    "metrics": {
        "elapsedTime": "120.822583ms",
        "executionTime": "120.790099ms",
        "resultCount": 227,
        "resultSize": 279707
    }
```

Using the suffixes-based array index and accordingly modified equivalent query:

```
cbq> CREATE INDEX idx_title_suffix
       ON `travel-sample`(DISTINCT ARRAY s FOR s IN SUFFIXES(title) END)
       WHERE type = "landmark";
cbq> SELECT * FROM `travel-sample` USE INDEX(idx_title_suffix)
  > WHERE type = "landmark" AND
  >       ANY s IN SUFFIXES(title) SATISFIES s LIKE "land%" END;
...
    "metrics": {
        "elapsedTime": "18.817681ms",
        "executionTime": "18.795954ms",
        "resultCount": 227,
        "resultSize": 279707
    }
```

Query Setting `pretty` Saves ~30%

It is an interesting fact that, on average, the white space overhead in a pretty-formatted JSON documents is around 30%. You can check the overhead in your JSON documents as explained below (in Linux):

- If you have a formatted JSON document, you can trim all the white space characters: `varakurprasad$ cat pretty-json-doc.json | tr -d " \t\n\r" > not-pretty-json-doc.json`
- If you have an unformatted JSON document, you can simply format it using python2.6 or later version: `varakurprasad$ cat not-pretty-json-doc.json | python -m json.tool > pretty-json-doc.json`
- Of course, if you have the documents in Couchbase already, you can use this new N1QL parameter `pretty`to check the overhead.

This formatting overhead can be a significant component in your query performance, especially when the queries produce large result documents. More interestingly, note that all the pretty formatting does not make much sense to an application program that is running the queries. In fact, the applications mostly use a JSON parser to process the query results and discard all the whitespace formatting.

`Pretty` is a new N1QL parameter introduced in 4.5.1 release that can enable or disable pretty formatting of the query results. The parameter can be:

- Set to `true` or `false` in the CBQ shell: `cbq> \SET -pretty false;`.
- Passed to the N1QL service process **CBQ-engine** as command line parameters.
- Passed as query parameter with REST calls.

Let's look at the numbers showing performance gain with `pretty`. Consider the `travel-sample` bucket, and the query `` select * from `travel-sample` ``

- With `pretty = true` , this query took ~6.16sec in my macbook-pro.
- With `pretty = false` , same query ran in 2.2 sec.
- That's a straight 3 times better performance! Find a detailed example **in my earlier blog Apps in N1QL Land: Magic With Pretty, Suffix, Update, and More.**

Streaming Through DISTINCT Operator

Typically, a DISTINCT operator used in queries is blocking in nature. This means that it first waits for all the input records to be available and then processes them to find the records with values of a given field in the record.

N1QL enhances the DISTINCT operator to stream process the input records. In 4.5.1, DISTINCT operator scans the input and emits the distinct values as it progresses. With the efficient streaming algorithm, now DISTINCT consumes much less memory besides CPU cycles. This significantly improves query performance, especially if you have a large document set, and are using distinct on an attribute with low cardinality (such as **state** or **country**).

Note that, this also preserves the order of input, and consequently retains and leverages the index-order of the documents. Hence, queries with ORDER BY clause will benefit further by avoiding sort-processing.

Altogether, this significantly improves the response times seen by the applications, as they will start receiving the query results as soon as they are available, while N1QL is processing and producing subsequent results. This optimization is applicable to usage of **distinct** in various scenarios such as, query projections, **union** queries, and part of **interesect**, **intersect all**, **except**, **except all**, etc.

Efficient Memory Usage for Processing Projections

Processing of projections in queries is a memory-intensive operation especially when using * as **projection-list**. With this enhancement, N1QL optimized the memory allocation and memory usage algorithm to consume much less memory and reduced the over allocation and buffer. Depending on query, we have seen magnitude reduction in memory usage (for example, few GB to a couple hundred MB) and overall faster query processing efficiently.

Optimization for Processing IN With Parameters

This improvement optimized processing of queries which have the IN-clause used with parameters. This optimization specifically improves the usage of the available secondary indexes more precisely for pushing down the IN-clause predicate to index. For example, following query takes the parameter `$city` which is used in the IN predicate:

```
cbq> \set -$city "Dublin" ;
cbq> explain select city from `travel-sample` where city IN ["Santa Monica",
$city];
```

Let's see how the query executes. Without this optimization, a full scan of the index is performed (when parameters are involved), which obviously results in a much more expensive execution of the query because full-scan is a range scan over all entries in the index and returns all documents keys in the index.

```
cbq> \set -$city "Dublin" ;
cbq> explain select meta().id, city from `travel-sample` where city IN ["Santa
Monica", $city];
{
   "requestID": "82251ffa-7a9b-4c79-92ef-f549b2e2ee9f",
   "signature": "json",
   "results": [
       {
           "plan": {
               "#operator": "Sequence",
               "~children": [
                   {
                       "#operator": "IndexScan",
                       "covers": [
                           "cover ((`travel-sample`.`city`))",
                           "cover ((meta(`travel-sample`).`id`))"
                       ],
                       "index": "def_city",
                       "index_id": "64a2b5261a86839d",
                       "keyspace": "travel-sample",
                       "namespace": "default",
                       "spans": [
                           {
                               "Range": {
                                   "Inclusion": 0,
                                   "Low": [
                                       "null"
                                   ]
                               }
                           }
                       ],
                       "using": "gsi"
                   },
                   {
                       "#operator": "Parallel",
                       "~child": {
                           "#operator": "Sequence",
                           "~children": [
                               {
                                   "#operator": "Filter",
                                   "condition": "(cover
((`travel-sample`.`city`)) in [\"Santa Monica\", $city])"
                               },
                               {
                                   "#operator": "InitialProject",
                                   "result_terms": [
                                       {
                                           "expr": "cover
((meta(`travel-sample`).`id`))"
                                       },
                                       {
```

```
                                                    "expr": "cover
    ((`travel-sample`.`city`))"
                                                }
                                            ]
                                        },
                                        {
                                            "#operator": "FinalProject"
                                        }
                                    ]
                                }
                            }
                        ]
                    },
                    "text": "select meta().id, city from `travel-sample` where city IN
    [\"Santa Monica\", $city]"
                }
            ],
            "metrics": {
                "elapsedTime": "65.649232ms",
                "executionTime": "65.630462ms",
                "resultCount": 55,
                "resultSize": 2524
            }
```

Note that, following span is generated for the index lookup, with represents a full scan of the index.

```
                                "spans": [
                                    {
                                        "Range": {
                                            "Inclusion": 0,
                                            "Low": [
                                                "null"
                                            ]
```

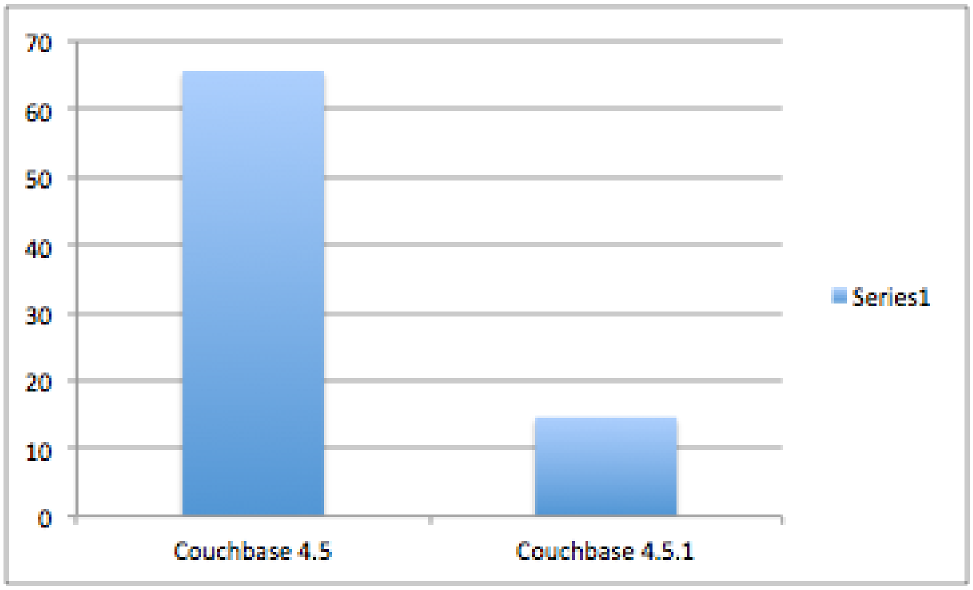
```
                          }
                      }
                  ],
```

Further, note that if it the query is not covered by the index scan (example: * in the projection), then it subsequently results in a fetch of all the documents from data service. However, most of these fetched documents may be discarded by N1QL when the IN predicate is processed later in the **filtering-phase** (refer to the N1QL execution phases in the beginning of this blog).

In Couchbase server 4.5.1, N1QL implemented an optimization for processing such queries. Now, N1QL generates precise lookup spans to index for each of the values in the **IN-clause** and avoids the expensive full scan. Note the `spans[]` array in the following `EXPLAIN` output that shows two index lookup spans generated for each of the IN-values (including the parameter `$city`).

```
cbq> \set -$city "Dublin" ;
cbq> explain select city from `travel-sample` where city IN ["Santa Monica", $city];
{
   "requestID": "aef94329-a1cb-4880-add0-c2efee7d1d8a",
   "signature": "json",
   "results": [
       {
           "plan": {
               "#operator": "Sequence",
               "~children": [
                   {
                       "#operator": "DistinctScan",
                       "scan": {
                           "#operator": "IndexScan",
                           "covers": [
                               "cover ((`travel-sample`.`city`))",
                               "cover ((meta(`travel-sample`).`id`))"
                           ],
                           "index": "def_city",
                           "index_id": "ad4c900681b6abe9",
                           "keyspace": "travel-sample",
                           "namespace": "default",
                           "spans": [
                               {
                                   "Range": {
                                       "High": [
                                           "\"Santa Monica\""
                                       ],
                                       "Inclusion": 3,
                                       "Low": [
                                           "\"Santa Monica\""
                                       ]
                                   }
                               },
                               {
                                   "Range": {
                                       "High": [
                                           "$city"
                                       ],
                                       "Inclusion": 3,
                                       "Low": [
                                           "$city"
                                       ]
                                   }
                               }
                           ],
                           "using": "gsi"
                       }
                   },
   "metrics": {
       "elapsedTime": "14.564755ms",
       "executionTime": "14.53648ms",
       "resultCount": 55,
       "resultSize": 2524
   }
```

Finally, note that this query ran almost 4.5 times faster (*y-axis* in the following chart is latency in ms). Note that, performance gain entirely depends on the selectivity of the IN predicate. That means that if the IN predicate matches only few documents, then the gain because of this optimization is much more pronounced than when the IN predicate matches many or all documents (in which case it tends to reach closer to a full scan anyway).



Summary

N1QL incorporated many performance and feature enhancements in Couchbase Server 4.5.1 as well as in the more recent Couchbase Server 4.6 Developer preview release. This article highlights some of the interesting performance enhancements. My earlier article Apps in N1QL Land: Magic With Pretty, Suffix, Update, and More, talks about salient N1QL enhancements in 4.5.1 release.

Apps in N1QL Land: Magic With Pretty, Suffix, Update, and More

Author: Prasad Varakur

Couchbase/N1QL

Couchbase is all about enabling more and more enterprise applications to leverage and adopt NoSQL/JSON data model. And, N1QL is the wonderland for applications which need all the great benefits of the JSON/Document databases, and don't want to lose the power of query language like SQL. In fact, N1QL is SQL++ and is built with tons of features to simplify the transition from traditional relational databases, and achieve best of both worlds.
In the latest release, Couchbase Server 4.5.1 brings multiple functionality, usability, and performance improvements in N1QL. While some of the new improvements enhance existing functionality, others, such as `SUFFIXES()` and `pretty`, enrich N1QL querying with magnitude performance improvement.
In this blog, I will show glimpses of the wonder land, and explain some of this magic. See what’s new and release notes for full list of N1QL enhancements. Kudos N1QL team!!

Indexing With SUFFIXES() for Efficient LIKE Queries

Pattern matching is a widely used functionality in SQL queries, and is typically achieved using the LIKE operator. Especially, efficient wildcard matching is very important. LIKE ‘foo%’ can be implemented efficiently with a standard index, as it has fixed prefix substring. But LIKE ‘%foo%’ is generally expensive. Such pattern matching with leading wildcard is vital for every application that has a search box to match partial words or to smart-suggest matching text. For example:

- A travel booking site that wants to pop-up matching airports as the user starts to enter a few letters of the airport name.
- A user finding all e-mails with a specific word or partial word in the subject.
- Finding all topics of a forum or blog posts with specific keywords in the title.

In Couchbase Server 4.5.1, N1QL addresses this problem by adding a new string function —SUFFIXES(), and combining that with the Array Indexing functionality introduced in Couchbase Server 4.5. Together, it brings a magnitude of difference to the performance of LIKE queries with leading wildcards such as LIKE "%foo%". The core functionality of SUFFIXES() is very simple. Basically, it produces an array of all possible suffix substrings of a given string. For example,

```
SUFFIXES("N1QL") = [ "N1QL", "1QL", "QL", "L" ]
```

The following picture depicts a unique technique to combine the `SUFFIXES()` function with Array Indexing to boost `LIKE` query performance.

1. Step 1 (in blue) shows the array of suffix substrings generated by `SUFFIXES()` for `doc1.title`
2. Step 2 (in yellow) shows the Array Index created with the suffix substrings generated in step 1. Note that the index-entry for`"wood"` points to `doc1` and `doc4`, as that is one of the suffix substrings of titles of both the documents. Similarly, `"ood"` points to `doc1`, `doc4`, and `doc8`.
3. Step 3 (in green) runs a query equivalent to `SELECT title FROM bucket WHERE title LIKE "%wood%"`. The LIKE predicate is transformed to use the Array Index using the ANY construct. See the documentation for more details on using array indexing.
    - Note that the leading wildcard is removed in the new `LIKE "wood%"` predicate.
    - This is accurate transformation because the array index lookup for `"wood"` points to all documents whose title has trailing substring `"wood"`
4. In Step 4, N1QL looks up in the Array Index to find all documents matching `"wood%"`. That returns `{doc1, doc3, doc4}`, because:
    - The index lookup produces a span, which gets documents from `"wood"` to `"wooe"`

    - doc1 and doc4 are matched because of index entry "wood" that is generated by the SUFFIXES() when creating the array index.
    - doc3 is matched because of its corresponding index-entry for "woodland"
5. Finally, in step 5, N1QL returns the query results.

Let's see a working example with the travel-sample documents/bucket, which is shipped with the Couchbase Server product. This showed a **10-12x boost** in performance for the example query.

1. Assume a document with a string field whose value is few words of text or a phrase. For example, title of a landmark, address of a place, name of restaurant, full name of a person/place etc., For this explanation, we consider title of landmarkdocuments in travel-sample.
2. Create secondary index on title field using SUFFIXES() as:
```
CREATE INDEX idx_title_suffix
ON `travel-sample`(DISTINCT ARRAY s FOR s IN SUFFIXES(title) END)
WHERE type = "landmark";
```
3. SUFFIXES(title) generates all possible suffix substrings of title, and the index will have entries for each of those substrings, all referencing to corresponding documents.
4. Now consider following query, which finds all docs with substring "land" in title. This query produces following plan and runs in roughly 120ms on my laptop. You can clearly see, it fetches all landmark documents, and then applies the LIKE "%land%"predicate to find all matching documents.

```
EXPLAIN SELECT * FROM `travel-sample` USE INDEX(def_type) WHERE type = "landmark"
AND title LIKE "%land%";
[
 {
   "plan": {
     "#operator": "Sequence",
     "~children": [
       {
         "#operator": "IndexScan",
         "index": "def_type",
         "index_id": "e23b6a21e21f6f2",
         "keyspace": "travel-sample",
         "namespace": "default",
         "spans": [
           {
             "Range": {
               "High": [
                 "\"landmark\""
               ],
               "Inclusion": 3,
               "Low": [
                 "\"landmark\""
               ]
             }
           }
         ],
         "using": "gsi"
       },
       {
         "#operator": "Fetch",
         "keyspace": "travel-sample",
         "namespace": "default"
       },
       {
         "#operator": "Parallel",
         "~child": {
           "#operator": "Sequence",
           "~children": [
             {
               "#operator": "Filter",
               "condition": "(((`travel-sample`.`type`) = \"landmark\") and
((`travel-sample`.`title`) like \"%land%\"))"
             }
           ]
       }
```

In Couchbase 4.5.1, this query can be rewritten to leverage the array index idx_title_suffix created in (2) above.

```
EXPLAIN SELECT title FROM `travel-sample` USE INDEX(idx_title_suffix) WHERE type =
"landmark" AND
ANY s IN SUFFIXES(title) SATISFIES s LIKE "land%"  END;
[
 {
   "plan": {
     "#operator": "Sequence",
     "~children": [
       {
         "#operator": "DistinctScan",
         "scan": {
           "#operator": "IndexScan",
           "index": "idx_title_suffix",
           "index_id": "75b20d4b253214d1",
           "keyspace": "travel-sample",
           "namespace": "default",
           "spans": [
             {
               "Range": {
                 "High": [
                   "\"lane\""
                 ],
                 "Inclusion": 1,
                 "Low": [
                   "\"land\""
                 ]
               }
             }
           ],
           "using": "gsi"
         }
       },
       {
         "#operator": "Fetch",
         "keyspace": "travel-sample",
         "namespace": "default"
       },
       {
         "#operator": "Parallel",
         "~child": {
           "#operator": "Sequence",
           "~children": [
             {
               "#operator": "Filter",
               "condition": "(((`travel-sample`.`type`) = \"landmark\") and any
`s` in suffixes((`travel-sample`.`title`)) satisfies (`s` like \"land%\") end)"
             },
```
Note that:

- The new query in (4) uses LIKE “land%”, instead of LIKE “%land%”. The former predicate, with no leading wildcard '%', produces a much more efficient index lookup than the latter one, which can’t push down the predicate to index.
- The array index idx_title_suffix is created with all possible suffix substrings of title, and hence lookup for any suffix substring of title can find successful match.
- In the above 4.5.1 query plan in (4), N1QL pushes down the LIKE predicate to the index lookup, and avoids additional pattern-matching string processing. This query ran in 18ms.
- In fact, with following covering Array Index, the query ran in 10ms, which is 12x faster.

```
CREATE INDEX idx_title_suffix_cover
ON `travel-sample`(DISTINCT ARRAY s FOR s IN SUFFIXES(title) END, title)
WHERE type = "landmark";
```

See this blog for details on a real-world application of this feature.

UPDATE Nested Arrays

Enterprise applications often have complex data and need to model JSON documents with multiple levels of nested objects and arrays. N1QL supports complex expressions and language constructs to navigate and query such documents with nested arrays. N1QL also supports Array Indexing, with which secondary indexes can be created on array elements, and subsequently queried.

In Couchbase Server 4.5.1, the UPDATE statement syntax is improved to navigate nested arrays in documents and update specific fields in nested array elements. The FOR-clause of the UPDATE statement is enhanced to evaluate functions and expressions, and the new syntax supports multiple nested FOR expressions to access and update fields in nested arrays.

Consider the following document with a nested array like:

```
{
  items: [
    {
      subitems: [
        {
          name: "N1QL"
        },
        {
          name: "GSI"
        }
      ]
    }
  ],
  docType: "couchbase"
}
```

The new UPDATE syntax in 4.5.1 can be used in different ways to access and update nested arrays:

```
UPDATE default SET s.newField = 'newValue'
FOR s IN ARRAY_FLATTEN(items[*].subitems, 1) END;

UPDATE default
SET s.newField = 'newValue'
FOR s IN ARRAY_FLATTEN(ARRAY i.subitems FOR i IN items END, 1) END;

UPDATE default
SET i.subitems = ( ARRAY OBJECT_ADD(s, 'newField', 'newValue')
                  FOR s IN i.subitems END ) FOR i IN items END;
```

Note that:

- The SET-clause evaluates functions such as OBJECT_ADD() and ARRAY_FLATTEN()
- FOR constructs can be used in a nested fashion with expressions to process array elements at different nest-levels.

For a working example, consider the sample bucket travel-sample, shipped with 4.5.1.

1. First, let’s add a nested array of special flights to the array schedule in the travel-sample bucket, for some documents.

```
UPDATE `travel-sample`
SET schedule[0] = {"day" : 7, "special_flights" :
                [ {"flight" : "AI444", "utc" : "4:44:44"},
                     {"flight" : "AI333", "utc" : "3:33:33"}
                   ]}
        WHERE type = "route" AND destinationairport = "CDG" AND sourceairport = "TLV";
```

2. The following UPDATE statement adds a third field to each special flight:

```
UPDATE `travel-sample`
SET i.special_flights = ( ARRAY OBJECT_ADD(s, 'newField', 'newValue' )
                          FOR s IN i.special_flights END )
                                       FOR i IN schedule END
WHERE type = "route" AND destinationairport = "CDG" AND sourceairport =
"TLV";
SELECT schedule[0] from `travel-sample`
WHERE type = "route" AND destinationairport = "CDG" AND sourceairport =
"TLV" LIMIT 1;
[
 {
   "$1": {
     "day": 7,
     "special_flights": [
       {
         "flight": "AI444",
         "newField": "newValue",
         "utc": "4:44:44"
       },
       {
         "flight": "AI333",
         "newField": "newValue",
         "utc": "3:33:33"
       }
     ]
   }
 }
]
```

Performance with new Query setting

Pretty!! Pretty!! `pretty`!! Yes, `pretty` is a super impressive new query setting in N1QL that can enable or disable pretty formatting of query results. You might ask, what's so great about it? And why anybody may want to disable the beautiful pretty formatting of the JSON output. Answer is **P E R F O R M A N C E**!!

As the old adage goes, there are no free lunches. Pretty formatting query results comes with its own expense:

1. First, a quick fact is that the white space (tabs, spaces, newlines) characters in a pretty-formatted JSON document consume almost a third of its size.
    - So, simply cutting down on the beautification will save all the raw bytes flowing over the network.
    - Moreover, consider the corresponding saving on the memory and processing resources of N1QL service.
    - Altogether, savings are pretty significant.
2. The pretty output format is good for human readable scenarios, with human-manageable result sizes.
    - However, real world applications and computer programs run queries much more often than humans — and process much bigger query results.
    - For these, what matters is **performance** and **efficiency**; not pretty formatting. In fact, such formatting is an overhead to the JSON parser and application and is usually discarded. Typically, applications have their own presentation layer to format the data appropriately for respective users.

The new query parameter `pretty` in 4.5.1 allows to enable/disable formatting a query result. The parameter can be:

- Set to `true` or `false`in the cbq shell.

```
cbq> \SET -pretty false;
```

```
varakurprasad$ ./cbq -u=Administrator -p=password
Connected to : http://localhost:8091/. Type Ctrl-D or \QUIT to exit.
cbq> \SET -pretty false;
cbq> select id, name from `travel-sample` limit 1;
{
"requestID": "e9d885bc-2378-4345-bf44-a336fbabce77",
"signature": {"id":"json","name":"json"},
"results": [
{"id":10,"name":"40-Mile Air"}
],
"status": "success",
"metrics": {"elapsedTime": "9.829246ms","executionTime":
"9.805407ms","resultCount": 1,"resultSize": 30}
}
```

- Passed to the N1QL service process `cbq-engine` as command line parameters.
- Passed as Query parameter with REST calls. See example below.

By default, it is set to `true`. When set to `false`, the whitespace characters are stripped from the query results. The performance benefits are significantly visible when queries produce large results, and it, of course, depends on the percentage of white-space overhead in your documents.
For example, following query which selects all documents from travel-sample, run almost **3x faster** when `pretty = false`. Also, note the size of the result set, which is one-third of the pretty formatted result.

With `pretty = true`

```
varakurprasad$ time curl -v http://localhost:8093/query/service -d
"pretty=true&statement=SELECT * from \`travel-sample\`" | tail -15
* Hostname was NOT found in DNS cache
 % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                Dload  Upload   Total   Spent    Left  Speed
 0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0*
Trying ::1...
* Connected to localhost (::1) port 8093 (#0)
* Server auth using Basic with user 'travel-sample'
> POST /query/service HTTP/1.1
> Authorization: Basic dHJhdmVsLXNhbXBsZTpoZWxsbw==
> User-Agent: curl/7.37.1
> Host: localhost:8093
> Accept: */*
> Content-Length: 51
> Content-Type: application/x-www-form-urlencoded
>
} [data not shown]
* upload completely sent off: 51 out of 51 bytes
< HTTP/1.1 200 OK
< Content-Type: application/json; version=1.6.0
< Date: Sat, 15 Oct 2016 02:04:09 GMT
< Transfer-Encoding: chunked
<
{ [data not shown]
100  103M    0  103M  100    51  16.6M      8  0:00:06  0:00:06 --:--:-- 17.2M
* Connection #0 to host localhost left intact
               ],
               "sourceairport": "TLV",
               "stops": 0,
               "type": "route"
           }
       }
   ],
   "status": "success",
   "metrics": {
       "elapsedTime": "6.165034483s",
       "executionTime": "6.164993497s",
       "resultCount": 31591,
       "resultSize": 107830610
   }
}
real 0m6.208s
user 0m5.704s
sys 0m0.373s
```

With `pretty = false`

Note that the total resultSize now is only 36.7MB, and the query ran in 2.2sec. This is **3X better performance** compared to `pretty=true` , which took 6.2sec and `returned 107.8MB.`

```
varakurprasad$ time curl -v http://localhost:8093/query/service -d
"pretty=false&statement=SELECT * from \`travel-sample\`" | tail -5
* Hostname was NOT found in DNS cache
 % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                Dload  Upload   Total   Spent    Left  Speed
 0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0*
Trying ::1...
* Connected to localhost (::1) port 8093 (#0)
* Server auth using Basic with user 'travel-sample'
> POST /query/service HTTP/1.1
> Authorization: Basic dHJhdmVsLXNhbXBsZTpoZWxsbw==
> User-Agent: curl/7.37.1
> Host: localhost:8093
> Accept: */*
> Content-Length: 52
> Content-Type: application/x-www-form-urlencoded
>
} [data not shown]
* upload completely sent off: 52 out of 52 bytes
< HTTP/1.1 200 OK
< Content-Type: application/json; version=1.6.0
< Date: Sat, 15 Oct 2016 02:03:29 GMT
< Transfer-Encoding: chunked
<
{ [data not shown]
100 35.1M    0 35.1M  100    52  15.9M     23  0:00:02  0:00:02 --:--:-- 15.9M
"status": "success",
"metrics": {"elapsedTime": "2.148354775s","executionTime":
"2.148323137s","resultCount": 31591,"resultSize": 36754457}
}
real 0m2.223s
user 0m1.977s
sys 0m0.141s
```

Enhancements to Dynamic Object Construction

N1QL already supports construction of JSON objects dynamically in queries. This immensely helps in creating specifically constructed result objects in query projection-lists. Couchbase server 4.5.1 extends the power of expressions and enriches the dynamic object creation and object processing in N1QL queries.

1. In Couchbase Server 4.5.1, N1QL allows both names and values of object fields to be arbitrary expressions. In earlier versions, the names of fields are required to be static strings. If a name does not evaluate to a string, the result of the object construction is NULL. For example:

```
SELECT { UPPER(callsign) || "_key" : callsign || ":" || country || ":" || name }
AS myobj
FROM `travel-sample`
WHERE type = 'airline' limit 1;
[
 {
   "myobj": {
     "MILE-AIR_key": "MILE-AIR:United States:40-Mile Air"
   }
 }
]
```

2. When constructing objects in a N1QL query, the names of fields in name-value pairs is made optional in 4.5.1. For example, the following query implicitly assigns names "type" and "name" for respective values:

```
SELECT {type, name} AS myobj FROM `travel-sample` LIMIT 1;
[
 {
   "myobj": {
     "type": "airport"
     "name": "airport_123"
   }
 }
]
```

New Array Function: `ARRAY_INTERSECT()`

ARRAY_INTERSECT() function takes two or more arrays as parameters and returns the intersection of the input arrays as the result, i.e the array containing values that are present in all the input arrays. It returns an empty array if there are no common array elements. For more information, see documentation. For example, following query finds the hotels that are liked by Brian or Lilian, in the travel-sample bucket shipped with Couchbase Server 4.5.1.

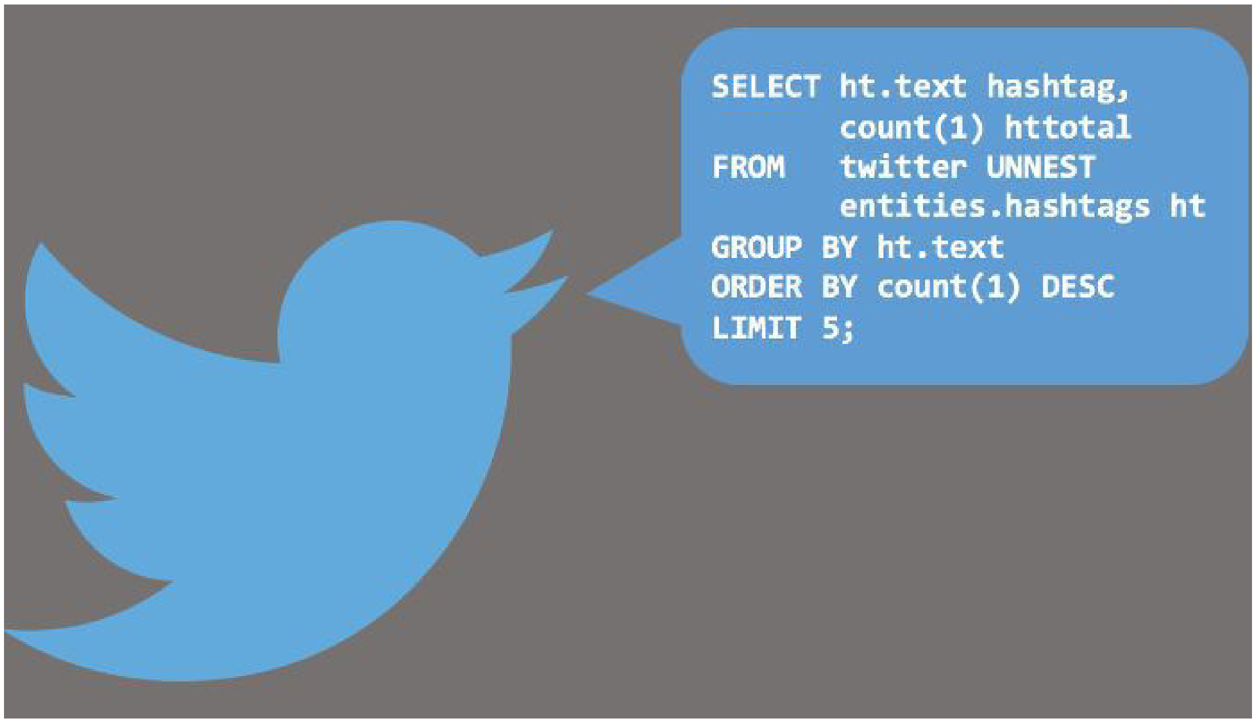
```
SELECT meta().id, ARRAY_INTERSECT(public_likes, ["Brian Kilback", "Lilian
McLaughlin"]) AS likes
FROM `travel-sample` WHERE type = 'hotel'
ORDER BY likes DESC
LIMIT 4;
[
 {
   "id": "hotel_10025",
   "likes": [
     "Lilian McLaughlin",
     "Brian Kilback"
   ]
 },
 {
   "id": "hotel_10026",
   "likes": []
 },
 {
   "id": "hotel_10064",
   "likes": []
 },
 {
   "id": "hotel_10063",
   "likes": []
 }
]
```

Download Couchbase Server 4.5.1 and give it a try. Learn more at N1QL documentation. Let me know any questions/comments, or just how awesome it is! Cheers!

Use Cases

SQL on Twitter: Analysis Made Easy Using N1QL

Author: Keshav Murthy



```
SELECT ht.text hashtag,
       count(1) httotal
FROM   twitter UNNEST
       entities.hashtags ht
GROUP BY ht.text
ORDER BY count(1) DESC
LIMIT 5;
```

"If I had more time, I would have written shorter letter" — *Blaise Pascal*

There have been lengthy articles on analyzing Twitter data. From Cloudera: here, here, and here. More from Hortonworks here and here. This one from Couchbase is going to be short, save the examples and results.

Step 1: Install Couchbase 4.5. Use the Couchbase console create a bucket called Twitter and CREATE PRIMARY INDEX on Twitter using the query workbench or cbq shell.

```
    CREATE PRIMARY INDEX ON twitter;
```

Step 2: Request your Twitter archive. Once you receive it, unzip it. (You can use larger twitter archives as well):

```
cd <to the unzipped location>/data/js/tweets
Step 3:
    $ for i in `ls`;
           do
             grep -i -v ^Grailbird $i > $i.out ;
           done
```

Step 4: Update your IP, username, and password before you run this:

```
    for i in `ls *.out`;
    do
       /opt/couchbase/bin/cbbackupmgr  json -host http://127.0.0.1:8091
    --username Administrator --password password --bucket twitter
    --dataset file:///home/keshav/mytweets/data/js/tweets/$i --format list
    --generate-key %id_str%;
    done
```

Step 5: There is no step 5!

Log into Couchbase's query workbench or cbq shell and start playing! Simply use SQL-based N1QL to query and play with the data. This online interactive tutorial will get you started with N1QL.

Here are the example queries on my twitter archive.

1. Give me the count of my tweets.

```
SELECT COUNT(*) my_tweet_count
FROM   twitter
LIMIT  1;
```

```
Results:
```

```
[
 {
   "my_tweet_count": 1658
 }
]
```

2. Get me a sample Twitter document.

```
SELECT *
FROM   twitter
LIMIT  1;
```

Results: Twitter document is rich. It has nested objects, arrays, and arrays of objects.

```
[
 {
   "twitter": {
     "created_at": "2011-08-19 18:09:31 +0000",
     "entities": {
       "hashtags": [
         {
           "indices": [
            79,
            88
          ],
          "text": "informix"
        },
        {
          "indices": [
            89,
            99
          ],
          "text": "warehouse"
        }
      ],
      "media": [],
      "urls": [
        {
          "display_url": "bit.ly/pkFdF4",
          "expanded_url": "http://bit.ly/pkFdF4",
          "indices": [
            113,
            132
          ],
          "url": "http://t.co/GnKGAKB"
        }
      ],
      "user_mentions": []
    },
    "geo": {},
    "id": 104615993220927490,
    "id_str": "104615993220927488",
    "source": "<a href=\"http://twitter.com\" rel=\"nofollow\">Twitter Web
Client</a>",
    "text": "No tuning required! Lester took his queries from ~10 hours to 15 minutes
using #informix #warehouse accelerator. http://t.co/GnKGAKB",
    "user": {
      "id": 282131568,
      "id_str": "282131568",
        "name": "Keshav Murthy",
        "profile_image_url_https":
"https://pbs.twimg.com/profile_images/670081620205023233/rHlKlkMC_normal.jpg",
        "protected": false,
        "screen_name": "rkeshavmurthy",
        "verified": false
      }
    }
  }
]
```

3. What days did I tweet most?

```
SELECT SUBSTR(created_at, 0, 10) tweet_date,
      COUNT(1) tweet_count
FROM   twitter
GROUP  BY SUBSTR(created_at, 0, 10)
ORDER  BY COUNT(1) DESC
LIMIT  5;
[
 {
   "tweet_count": 67,
   "tweet_date": "2013-11-05"
 },
 {
   "tweet_count": 60,
   "tweet_date": "2013-11-06"
 },
 {
   "tweet_count": 42,
   "tweet_date": "2014-04-30"
 },
 {
   "tweet_count": 41,
   "tweet_date": "2013-11-04"
 },
 {
   "tweet_count": 41,
   "tweet_date": "2014-04-28"
 }
]
```

4. Give me the top 5 hashtags and counts in my tweets:

```
SELECT    ht.text hashtag,
          COUNT(1) htcount
FROM      twitter UNNEST entities.hashtags ht
GROUP BY ht
ORDER BY COUNT(1) DESC
LIMIT 5;
[
  {
    "hashtag": "ibmiod",
    "htcount": 133
  },
  {
    "hashtag": "informix",
    "htcount": 31
  },
  {
    "hashtag": "IBMIOD",
    "htcount": 30
  },
  {
    "hashtag": "informix",
    "htcount": 26
  },
  {
    "hashtag": "Informix",
    "htcount": 21
  }
]
```

(Yes, I worked for Informix and IBM!)

5. How many tweets have I done on Couchbase, N1QL, NoSQL, or SQL?

Because hashtags are stored in an array, you need to UNNEST it so you can group by the hashtab.

```
SELECT   UPPER(ht.text) hashtag,
         COUNT(1)       htcount
FROM     twitter UNNEST entities.hashtags ht
WHERE    upper(ht.text) IN ['COUCHBASE', 'N1QL', 'NOSQL', 'SQL']
GROUP BY UPPER(ht.text)
ORDER BY COUNT(1) DESC
[
  {
    "hashtag": "NOSQL",
    "htcount": 258
  },
  {
    "hashtag": "COUCHBASE",
    "htcount": 162
  },
  {
    "hashtag": "SQL",
    "htcount": 64
  },
  {
    "hashtag": "N1QL",
    "htcount": 18
  }
]
```

6. Let’s see who I’ve mentioned in my tweets and how many times?

```
SELECT   UPPER(um.screen_name) umention,
        COUNT(1)               htcount
FROM     twitter UNNEST entities.user_mentions um
GROUP BY upper(um.screen_name)
ORDER BY count(1) DESC ;
```

I've only given partial results below. @N1QL and @Couchbase were top mentions. Note Twitter itself doesn't store the @ character in its data.

```
[
  {
    "htcount": 104,
    "umention": "N1QL"
  },
  {
    "htcount": 80,
    "umention": "COUCHBASE"
  },
…
]
```

7. Let’s get all the tweets I’ve mentioned @sangudi, creator of N1QL.

```
SELECT SUBSTR(created_at, 0, 10)    posted,
      text                          AS tweet
FROM   twitter
WHERE
        ANY u IN entities.user_mentions
             SATISFIES u.screen_name = 'sangudi'
        END
ORDER BY SUBSTR(created_at, 0, 10) DESC ;
[
 {
   "posted": "2016-06-10",
   "tweet": "JOIN and enjoy in Couchbase Server 4.5. tx 2
@sangudi\n#SQL #NoSQL #Couchbase #MongoDB
#JSON\nhttps://t.co/X9E0ghcx4L https://t.co/AYnetU5MHF"
 },
 {
   "posted": "2016-05-14",
   "tweet": "Brining SQL to NoSQL: Rich, declarative Query for NoSQL,
with @sangudi \n at @NoCOUG \nhttps://t.co/mnpPYKNQeA\n#Couchbase
#NoSQL #SQL #JSON"
 },
…
]
```

While this works fine, it scans the whole bucket using primary scan.

```
EXPLAIN SELECT SUBSTR(created_at, 0, 10)    posted,
      text                          AS tweet
FROM   twitter
WHERE
        ANY u IN entities.user_mentions
```

```
                SATISFIES u.screen_name = 'sangudi'
           END
    ORDER BY SUBSTR(created_at, 0, 10) DESC ;

       "plan": {
         "#operator": "Sequence",
         "~children": [
           {
             "#operator": "Sequence",
             "~children": [
               {
                 "#operator": "PrimaryScan",
                 "index": "#primary",
                 "keyspace": "twitter",
                 "namespace": "default",
                 "using": "gsi"
               },
```

Let’s create an index on this array element to make it go faster.

```
    CREATE INDEX idxtwittername on twitter
        (ALL ARRAY  u.screen_name FOR u IN entities.user_mentions END);
```

Now, see the plan for the same query. This uses the index and pushes down the predicate to the index, making the query faster.

```
    EXPLAIN SELECT SUBSTR(created_at, 0, 10)    posted,
          text                          AS tweet
    FROM   twitter
    WHERE
            ANY u IN entities.user_mentions
                SATISFIES u.screen_name = 'sangudi'
```

```
        END
ORDER BY SUBSTR(created_at, 0, 10) DESC ;
       {
         "#operator": "Sequence",
         "~children": [
           {
             "#operator": "DistinctScan",
             "scan": {
               "#operator": "IndexScan",
               "index": "idxtwittername",
               "index_id": "df30f58c0e0b9677",
               "keyspace": "twitter",
               "namespace": "default",
               "spans": [
                 {
                   "Range": {
                     "High": [
                       "\"sangudi\""
                     ],
                     "Inclusion": 3,
                     "Low": [
                       "\"sangudi\""
                     ]
                   }
                 }
               ],
               "using": "gsi"
             }
           },
```

Couchbase 4.5 makes it very easy to ingest JSON so you can get insight into your data. For more advanced questions and advanced usage, use array.

Try it out with your own Twitter data or a public JSON archive. Create indices on fields and arrays. Ask more questions, find more insights!

What's in a New York Name? Unlock data.gov Using N1QL

Author: Keshav Murthy

JSON Files: Data.gov

Data.gov, started in 2009, has about 189,000 datasets. Data is published in XML, CSV, JSON, HTML, PDF, and other formats. Data.gov aims to improve public access to high value, machine-readable datasets generated by the Executive Branch of the Federal Government. Lots of this data comes from the Socrata database. They also provide Socrata APIs to retrieve the subset of the data that you need. Data is valuable. Insights are more valuable. Instead of working with data trickle, let’s load all the data and analyze them.

We start this series using a dataset on a simple and seemingly inconsequential decision parents make: baby names. Obviously, parents take this decision quite seriously. If New York is the melting pot, let’s see what baby names those parents choose.
The techniques shown here will help you analyze not just this dataset, but all of the JSON datasets in data.gov. Let’s try to understand all of Obama’s open data secrets!

The Most Popular Baby Names by Sex and Mother's Ethnic Group, New York City dataset has collected baby names in New York City from 2011 to 2014. You can see the raw data at this link.

The data is in a single JSON document and consists of two fields: meta and data. Meta contains the description of the structure of the document and data, AKA metadata. Data contains data. This structure comes from Socrata.

Here's is the structure.

```
{
 "meta" : {
   "view" : {
     "id" : "25th-nujf",
     "name" : "Most Popular Baby Names by Sex and Mother's Ethnic
Group, New York City",
     "attribution" : "Department of Health and Mental Hygiene
(DOHMH)",
     "averageRating" : 0,
     "category" : "Health",
     "createdAt" : 1382724894,
     "description" : "The most popular baby names by sex and mother's
ethnicity in New York City.",
     "displayType" : "table",
...
    "columns" : [ {
       "id" : -1,
       "name" : "sid",
       "dataTypeName" : "meta_data",
       "fieldName" : ":sid",
       "position" : 0,
       "renderTypeName" : "meta_data",
       "format" : { }
     }, {
       "id" : -1,
       "name" : "id",
       "dataTypeName" : "meta_data",
       "fieldName" : ":id",
       "position" : 0,
       "renderTypeName" : "meta_data",
       "format" : { }
     }
  ...
  ]
"data" : [ [ 1, "EB6FAA1B-EE35-4D55-B07B-8E663565CCDF", 1, 1386853125,
"399231", 1386853125, "399231", "{\n}", "2011", "FEMALE", "HISPANIC",
"GERALDINE", "13", "75" ]
, [ 2, "2DBBA431-D26F-40A1-9375-AF7C16FF2987", 2, 1386853125,
"399231", 1386853125, "399231", "{\n}", "2011", "FEMALE", "HISPANIC",
"GIA", "21", "67" ]
, [ 3, "54318692-0577-4B21-80C8-9CAEFCEDA8BA", 3, 1386853125,
"399231", 1386853125, "399231", "{\n}", "2011", "FEMALE", "HISPANIC",
"GIANNA", "49", "42" ]
...
]
}
```

Rest of the article is divided into two sections:

1. Transforming the data.gov JSON dataset to be easily accessible.
2. Querying the New York names dataset with N1QL.

1. Transforming data.gov JSON

While the data is JSON, the whole document is stored as a single document. Let’s see how fast we can get insights.

We use Couchbase, a distributed NoSQL database and its built-in N1QL query language designed to be SQL for JSON. You need to download and install Couchbase 4.5 to follow along the following examples.

Step 1

Create two buckets, datagov and nynames, and create a primary index on both. Just 100 megabytes is sufficient for this data sets. And then create primary indices on both.

```
CREATE PRIMARY INDEX ON datagov;
CREATE PRIMARY INDEX ON nynames;
```

Step 2

Download the JSON document from this link.

Step 3

Create a file (insert.sql) with the INSERT statement and append the document. You need to surround the source data with the INSERT statement highlighted here in blue.

```
INSERT INTO datagov VALUES("nyname01", {
 "meta" : {
   "view" : {
     "id" : "25th-nujf",
     "name" : "Most Popular Baby Names by Sex and Mother's Ethnic
Group, New York City",
     "attribution" : "Department of Health and Mental Hygiene
(DOHMH)",
     "averageRating" : 0,
     "category" : "Health",
     "createdAt" : 1382724894,
     "description" : "The most popular baby names by sex and mother's
ethnicity in New York City.",
     "displayType" : "table",
     "downloadCount" : 1782,
     "indexUpdatedAt" : 1465427458,
     "newBackend" : false,
     "numberOfComments" : 0,
     "oid" : 6785037,
     "publicationAppendEnabled"
….
});
```

Step 4

Simply execute the statement via cbq.

```
$ cbq < insert.sql
```

Step 5

Now you have the whole data as a single document in a Couchbase bucket. This document is a well-formed JSON. Because the data is stored as series of scalar values of arrays of arrays, querying can be quite complex.

Step 6

Let's simplify this by transforming the data into flattened JSON. First, let's list the full set of data fields.

```
SELECT meta.`view`.columns[*].fieldName
   FROM datagov;
[
 {
   "fieldName": [
     ":sid",
     ":id",
     ":position",
     ":created_at",
     ":created_meta",
     ":updated_at",
     ":updated_meta",
     ":meta",
     "brth_yr",
     "gndr",
     "ethcty",
     "nm",
     "cnt",
     "rnk"
   ]
 }
]
```

To transform the data, you can use the query below.

```
INSERT INTO nynames (KEY UUID(), VALUE kname)
SELECT {":sid":d[0],
       ":id":d[1],
  ":position":d[2],
  ":created_at":d[3],
  ":created_meta":d[4],
  ":updated_at":d[5],
  ":updated_meta":d[6],
        ":meta":d[7],"brth_yr":d[8],
   "brth_yr":d[9],
   "ethcty":d[10],
   "nm":d[11],
   "cnt":d[12],
   "rnk":d[13]} kname
FROM (SELECT d FROM datagov UNNEST data d) as u1;
```

```
{
 "results": [],
 "metrics": {
   "elapsedTime": "3.578776104s",
   "executionTime": "3.578736949s",
   "resultCount": 0,
   "resultSize": 0,
   "mutationCount": 13962
 }
}
```

Better method: I’ve saved the best for last.The query in step (8) works fine. However, this is the old way. This requires you two-hand code the field names. There has to be a better, generalized way. With N1Ql, there is a better way.

```
INSERT INTO nynames
      (
           KEY UUID(),
           value o
      )
SELECT o
FROM   (
       SELECT meta.`view`.columns[*].fieldName f,
              data
       FROM   datagov) d
       UNNEST data d1
LET o = OBJECT p:d1[ARRAY_POSITION(d.f, p)] FOR p IN d.f END ;
{
 "results": [],
 "metrics": {
   "elapsedTime": "3.765045363s",
   "executionTime": "3.765005394s",
   "resultCount": 0,
   "resultSize": 0,
   "mutationCount": 13962
 }
}
```

Let’s first understand this query before we can appreciate it.

The INSERT INTO clause simply generates UUID() and the document o projected from the SELECT into the bucket nynames.

In the SELECT, there is a subquery that’s projecting the column list we saw before and the entire dataset. In the main SELECT, the data is unnested to flatten it. Now you have three objects to handle.

d.f: List of field names.

d.data: Full data of baby names, arrays of arrays.

d1: An array for each baby name and its statistics

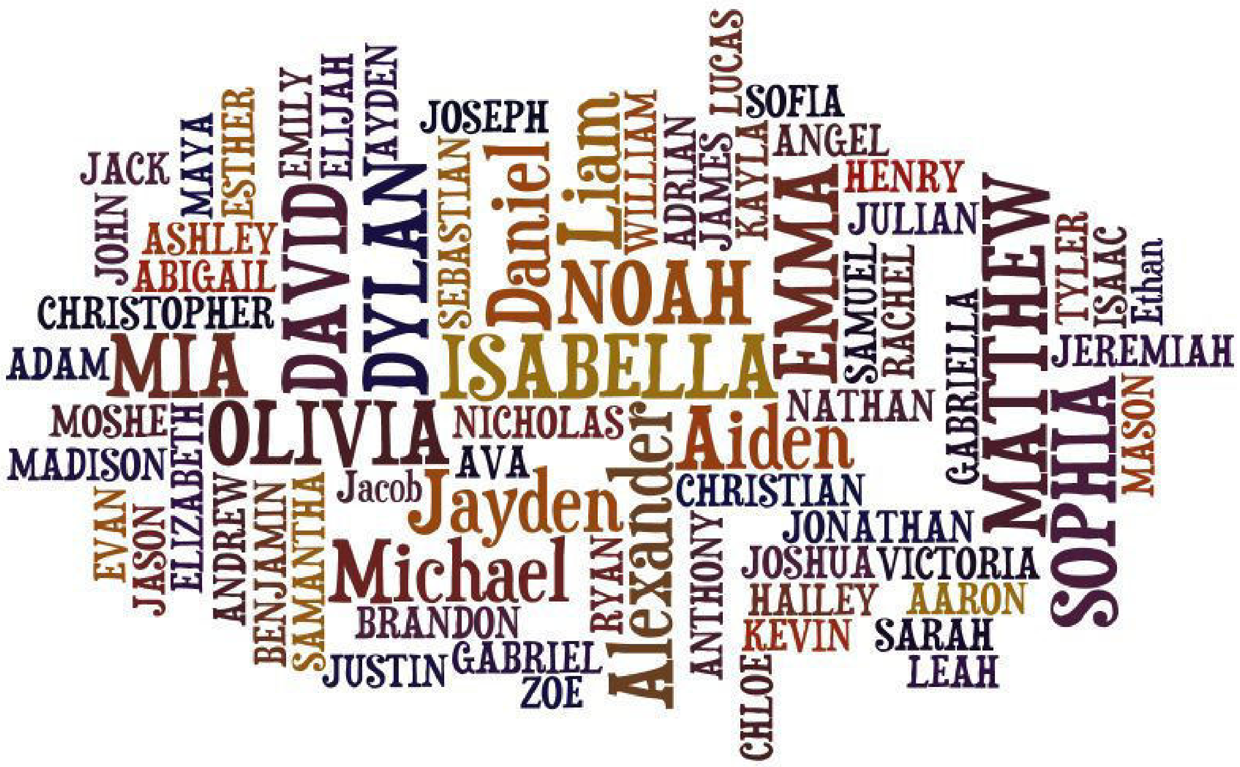
Let’s look at the expression:

```
   LET o = OBJECT p:d1[ARRAY_POSITION(d.f, p)] FOR p IN d.f END ;
```

This creates a JSON document by creating name:value pairs by having the variable p iterate through the names in d.f (names) and the expression d1[array_position(d.f, p)] selecting the corresponding data from the flattened array d.

Many thanks to Sitaram Vemulapalli for writing this beautiful query.

2. Querying the New York Names Dataset



As you can see, this JSON structure isn't conducive for reporting or query processing. So, let's flatten this. N1QL makes it very easy to transform this array of scalars to easily manipulate JSON data. See the section at the latter half of the article to see how to transform this data into a flattened JSON like below.

Once you do the transformation, you have all the name data in a bucket called nynames. Now, let's ask some common questions and see what we understand!

Q1: Let’s get some sample data. Since we have used UUID() as primary keys and this query uses the primary key, the data you get here are random.

```
SELECT *
FROM nynames
LIMIT 2;
[
 {
   "nynames": {
     ":created_at": 1465395604,
     ":created_meta": "399231",
     ":id": "D3F1A794-A634-4DA4-AA66-96987BB21E92",
     ":meta": null,
     ":position": 13240,
     ":sid": 13240,
     ":updated_at": 1465395604,
     ":updated_meta": "399231",
     "brth_yr": "2014",
     "cnt": "27",
     "ethcty": "BLACK NON HISPANIC",
     "gndr": "MALE",
     "nm": "Joseph",
     "rnk": "45"
   }
 },
 {
   "nynames": {
     ":created_at": 1386853125,
     ":created_meta": "399231",
     ":id": "9AC42CB0-2A84-4497-A907-DF7BF645C953",
     ":meta": "{\n}",
     ":position": 374,
     ":sid": 374,
     ":updated_at": 1386853125,
     ":updated_meta": "399231",
     "brth_yr": "2011",
     "cnt": "22",
     "ethcty": "WHITE NON HISPANIC",
     "gndr": "FEMALE",
     "nm": "JENNA",
     "rnk": "69"
   }
 }
]
```

Q2. What's the total count of names and babies in the New York dataset?

This data covers the 2,811 most popular names. Let's see which name is the most popular and which name is the least popular, at least in this dataset.

```
SELECT COUNT(DISTINCT nm ) as namecount,
     SUM(TONUMBER(cnt))  as babycount
FROM nynames;
[
 {
   "babycount": 482131,
   "namecount": 2811
 }
]
SELECT name, SUM(TONUMBER(cnt)) as nc
FROM nynames
GROUP BY name
ORDER BY SUM(TONUMBER(cnt)) DESC
LIMIT 1;
[
 {
   "nc": 10,
   "nm": "Kameron"
 }
]
```

Let’s find the top 10 popular names in New York.

```
SELECT nm as name, SUM(TONUMBER(cnt)) as namecount
FROM nynames
GROUP BY nm
ORDER BY SUM(TONUMBER(cnt)) DESC
LIMIT 10;
[
 {
   "name": "JAYDEN",
   "namecount": 4040
 },
 {
   "name": "JACOB",
   "namecount": 3437
 },
 {
   "name": "ETHAN",
   "namecount": 3403
 },
 {
   "name": "DANIEL",
   "namecount": 3009
 },
 {
   "name": "MATTHEW",
   "namecount": 2984
 },
 {
   "name": "MICHAEL",
   "namecount": 2972
 },
 {
   "name": "ISABELLA",
   "namecount": 2929
 },
 {
   "name": "SOPHIA",
   "namecount": 2809
 },
 {
   "name": "DAVID",
   "namecount": 2764
 },
 {
   "name": "AIDEN",
   "namecount": 2698
 }
```

```
]
```

So, they repeat boys names much more often than girls names!

Q4. What's the total count of babies by gender?

```
SELECT gndr as gender, SUM(TONUMBER(cnt)) as babycount
FROM nynames
GROUP BY gndr;
[
 {
   "babycount": 272048,
   "gender": "MALE"
 },
 {
   "babycount": 210083,
   "gender": "FEMALE"
 }
]
```

Q5. What are the top three groups in terms of number of children?

```
SELECT ethcty AS Ethnicity,
       brth_yr AS BirthYear,
       SUM(tonumber(cnt)) AS NameCount
FROM nynames
GROUP BY ethcty, brth_yr
ORDER BY SUM(TONUMBER(cnt)) DESC
LIMIT 3;
[
 {
   "BirthYear": "2011",
   "Ethnicity": "WHITE NON HISPANIC",
   "NameCount": 99512
 },
 {
   "BirthYear": "2011",
   "Ethnicity": "HISPANIC",
   "NameCount": 97768
 },
 {
   "BirthYear": "2011",
   "Ethnicity": "BLACK NON HISPANIC",
   "NameCount": 42164
 }
]
```

Q6. Now, let’s see Most popular starting alphabet

```
SELECT SUBSTR(LOWER(`nm`), 0, 1) alphabet,
        SUM(TONUMBER(cnt)) ncount
FROM nynames
GROUP BY substr(lower(`nm`), 0, 1)
ORDER BY sum(tonumber(cnt)) DESC
LIMIT 10;
[
  {
    "alphabet": "a",
    "ncount": 73562
  },
  {
    "alphabet": "j",
    "ncount": 61599
  },
  {
    "alphabet": "m",
    "ncount": 48090
  },
  {
    "alphabet": "s",
    "ncount": 36204
  },
  {
    "alphabet": "e",
    "ncount": 35552
  },
  {
    "alphabet": "c",
    "ncount": 27191
  },
  {
    "alphabet": "l",
    "ncount": 26865
  },
  {
    "alphabet": "d",
    "ncount": 21026
  },
  {
    "alphabet": "n",
    "ncount": 17195
  },
    {
      "alphabet": "k",
      "ncount": 16632
    }
  ]
```

Conclusion

Data.gov has tons of data. This data unlocks how the US government has worked in the past eight years. You can question the data to understand what worked and what didn't.

N1QL provides the key link between just raw data to queryable form. In the transformation query, data specific field names are never used. That means that this query can be used for all of the JSON data sets in this format at data.gov. This is the power of N1QL, SQL for JSON: expressive, powerful, and completed language for querying, transforming, and manipulating JSON data.

Go ahead. Try this on a data set you love on data.gov.

Related Information:

1. **Couchbase: http://www.couchbase.com**
2. **N1QL: http://query.couchbase.com**
3. **Data.gov: http://www.data.gov**
4. **JSON datasets in Data.gov: JSON Data Sets.**

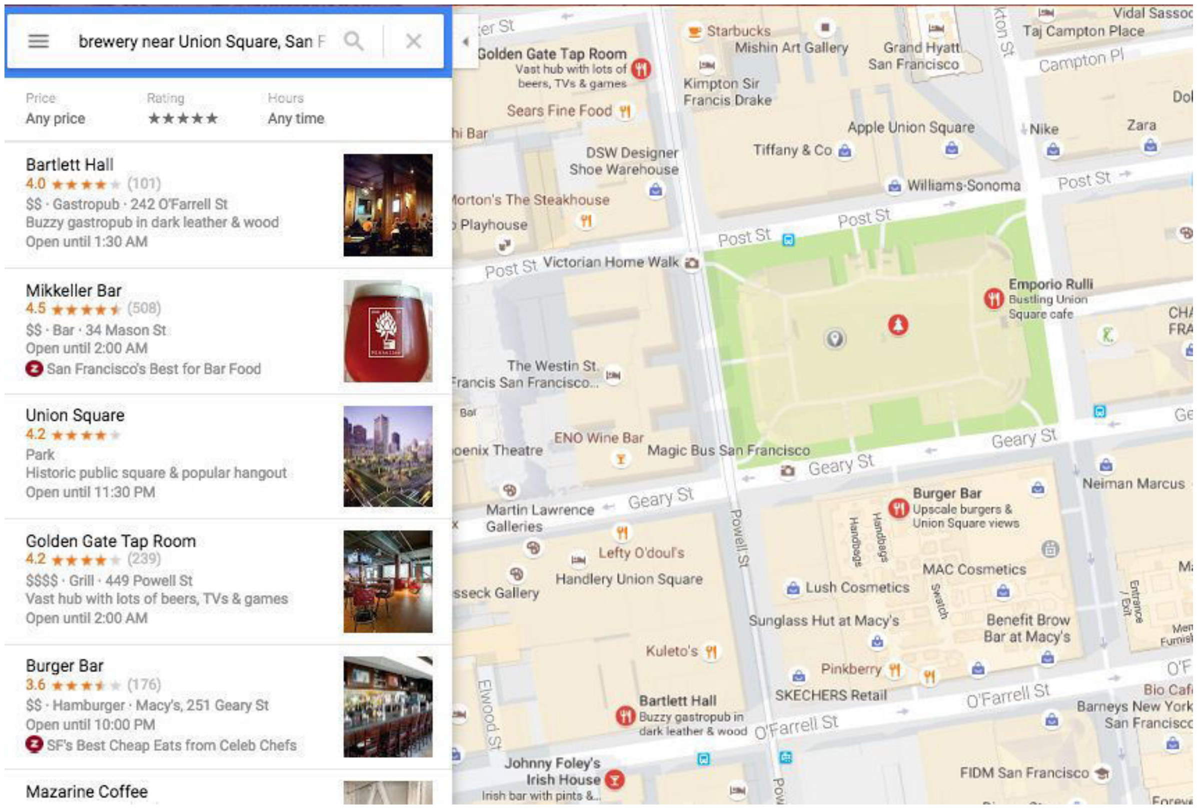
How to Speed Up Spatial Search in Couchbase N1QL

Author: Keshav Murthy

Location-based services like Yelp, Uber, and Foursquare are ubiquitous. This article shows an easy way to use Couchbase N1QL to issue spatial queries and use GSI index to speed up those queries to meet your SLAs.

Whether you just finished a long SF Giants extra innings win or spent the afternoon running up and down the Lombard Street in San Francisco, sometimes, you just want a cold one, really fast. When you search on your app, you expect the answer quickly, with ratings, distance, and all. Thanks to GPS, triangulation, and IP-based location detection, apps can determine your current location.

Google does this. Yelp does this. Uber does this. How can you use your spatial data to provide location-based services or information?



Bartlett Hall
4.0 ★★★★ (101)
$$ · Gastropub · 242 O'Farrell St
Buzzy gastropub in dark leather & wood
Open until 1:30 AM

Mikkeller Bar
4.5 ★★★★★ (508)
$$ · Bar · 34 Mason St
Open until 2:00 AM
San Francisco's Best for Bar Food

Union Square
4.2 ★★★★
Park
Historic public square & popular hangout
Open until 11:30 PM

Golden Gate Tap Room
4.2 ★★★★ (239)
$$$$ · Grill · 449 Powell St
Vast hub with lots of beers, TVs & games
Open until 2:00 AM

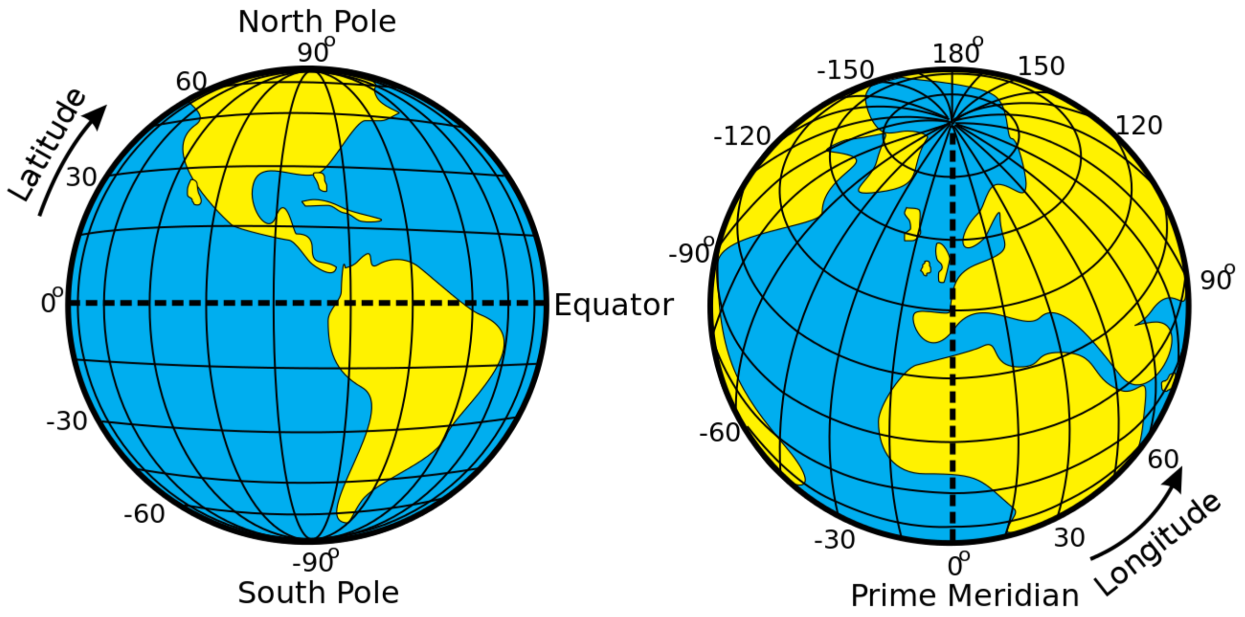
Burger Bar
3.6 ★★★★ (176)
$$ · Hamburger · Macy's, 251 Geary St
Open until 10:00 PM
SF's Best Cheap Eats from Celeb Chefs

Mazarine Coffee

Here is an example of spatial data for two breweries from the beer-sample data set shipped with Couchbase.

```
    {
      "geo": {
        "accuracy": "ROOFTOP",
        "lat": 37.7825,
        "lon": -122.393
      },
      "name": "21st Amendment Brewery Cafe"
    },
    {
      "geo": {
        "accuracy": "RANGE_INTERPOLATED",
        "lat": 50.7668,
        "lon": 4.3081
      },
      "name": "3 Fonteinen Brouwerij Ambachtelijke Geuzestekerij"
    }
```

Each numerical pair, geo.lat and geo.lon represent latitude and longitude on earth. You’re in Union Square, San Francisco with latlong (37.21, -123.38). And you have a database with breweries and their locations.



These are the typical questions to answer for your application.

1. Search breweries within 10 kilometers of my location.
2. Give the distances to each of these breweries.
3. Give me a sorted list of these breweries.

The application may have additional concerns about relevance, sorting by distance, rating, etc. Let's focus on finding answers to the three questions above and doing so efficiently.

1. Search Breweries Within 10 Kilometers of My Location

First, let's look at the sample data shipped with Couchbase.

```
SELECT * FROM `beer-sample` WHERE type = 'brewery' LIMIT 1;
[
 {
   "beer-sample": {
     "address": [
       "563 Second Street"
     ],
     "city": "San Francisco",
     "code": "94107",
     "country": "United States",
     "description": "The 21st Amendment Brewery offers a variety of award
winning house made brews and American grilled cuisine in a comfortable loft
like setting. Join us before and after Giants baseball games in our outdoor
beer garden. A great location for functions and parties in our semi-private
Brewers Loft. See you soon at the 21A!",
     "geo": {
       "accuracy": "ROOFTOP",
       "lat": 37.7825,
       "lon": -122.393
     },
     "name": "21st Amendment Brewery Cafe",
     "phone": "1-415-369-0900",
     "state": "California",
     "type": "brewery",
     "updated": "2010-10-24 13:54:07",
     "website": "http://www.21st-amendment.com/"
   }
 }
]
```

This is a small dataset to show full examples. The same approach described in this article will work on larger data sets as well. Bigger data sets will have bigger performance benefits!

```
SELECT COUNT(*) FROM `beer-sample` WHERE type = 'brewery' ;
[
 {
   "$1": 1412
 }
]
```

This dataset contains a geolocation for most of the breweries. We'll be using that to find the nearest breweries.

2. Give the Distances to Each of These Breweries

Let's use Union Square, San Francisco with latlong (37.21, -123.38).
That's easy. simply execute this query. You'll have the results. Observe the address, city, and geo-information in the result. All are within San Francisco, with latitude and longitude pretty close to our starting point.

```
SELECT country, state, city, name, geo, address
FROM `beer-sample`
WHERE (RADIANS(geo.lat) >= 0.658234696881 AND RADIANS(geo.lat) <=
0.660743280942)
AND (RADIANS(geo.lon) >= -2.13805724718 AND  RADIANS(geo.lon) <=
-2.13488305105)
AND  acos(sin( RADIANS(37.7859357)) * sin (RADIANS(geo.lat)) + cos(
RADIANS(37.7859357 ))  * cos(RADIANS(geo.lat)) * cos (RADIANS(geo.lon) -
RADIANS( -122.4107226))) <= 0.00125568984461
[
 {
   "address": [
     "1195-A Evans Avenue"
   ],
   "city": "San Francisco",
   "country": "United States",
    "geo": {
      "accuracy": "RANGE_INTERPOLATED",
      "lat": 37.7387,
      "lon": -122.381
    },
    "name": "Speakeasy Ales and Lagers",
    "state": "California"
  },
  {
    "address": [
      "3435 Cesar Chavez #227"
    ],
    "city": "San Francisco",
    "country": "United States",
    "geo": {
      "accuracy": "RANGE_INTERPOLATED",
      "lat": 37.7481,
      "lon": -122.419
    },
    "name": "Shmaltz Enterprises",
    "state": "California"
  },
  {
    "address": [
      "1705 Mariposa Street"
    ],
    "city": "San Francisco",
    "country": "United States",
    "geo": {
      "accuracy": "ROOFTOP",
      "lat": 37.7635,
      "lon": -122.401
    },
    "name": "Anchor Brewing",
    "state": "California"
  },
  {
    "address": [
      "1398 Haight Street"
    ],
    "city": "San Francisco",
    "country": "United States",
    "geo": {
      "accuracy": "RANGE_INTERPOLATED",
      "lat": 37.7702,
      "lon": -122.445
    },
    "name": "Magnolia Pub and Brewery",
    "state": "California"
  },
  {
    "address": [],
    "city": "San Francisco",
    "country": "United States",
    "geo": {
      "accuracy": "APPROXIMATE",
      "lat": 37.7749,
      "lon": -122.419
    },
    "name": "Golden Gate Park Brewery",
    "state": "California"
  },
  {
    "address": [],
    "city": "San Francisco",
    "country": "United States",
    "geo": {
      "accuracy": "APPROXIMATE",
      "lat": 37.7749,
      "lon": -122.419
    },
    "name": "Shmaltz Brewing Company",
    "state": "California"
  },
  {
    "address": [
      "650 Fifth Street #403"
    ],
    "city": "San Francisco",
    "country": "United States",
    "geo": {
      "accuracy": "ROOFTOP",
      "lat": 37.7756,
      "lon": -122.398
```

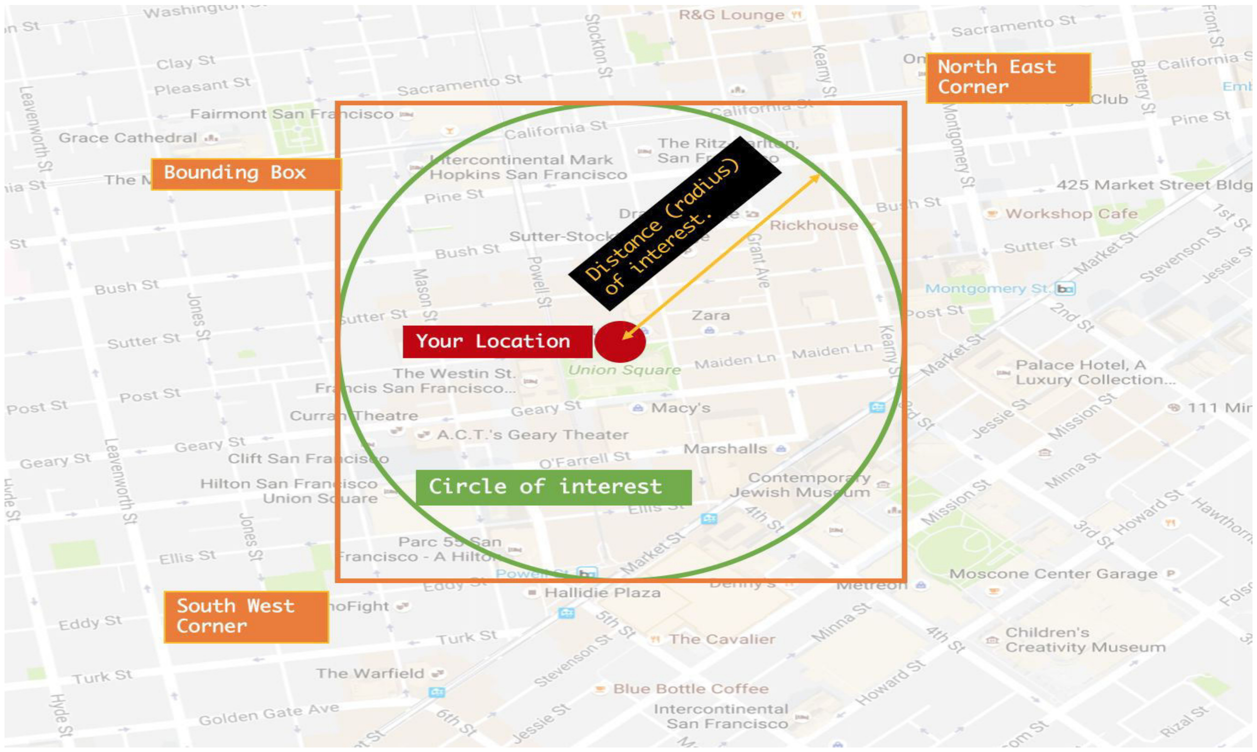
```
   },
   "name": "Big Bang Brewery (Closed)",
   "state": "California"
 },
 {
   "address": [
     "563 Second Street"
   ],
   "city": "San Francisco",
   "country": "United States",
   "geo": {
     "accuracy": "ROOFTOP",
     "lat": 37.7825,
     "lon": -122.393
   },
   "name": "21st Amendment Brewery Cafe",
   "state": "California"
 },
 {
   "address": [
     "661 Howard Street"
   ],
   "city": "San Francisco",
   "country": "United States",
   "geo": {
     "accuracy": "ROOFTOP",
     "lat": 37.7855,
     "lon": -122.4
   },
   "name": "ThirstyBear Brewing",
   "state": "California"
 }
]
```

Now, you ask, how did I come up with this gobbledygook query? What query should I run if the location is different? Those are fair questions. Let's take a detour here.

The approach and the math are described by Jan Philip Matuschek in this paper. There are several implementations of this approach to finding the bounding box for a given latlong. I used the Python implementation by Jeremy Fein to find the bounding box.

Once you have the bounding box, the paper shows you how to write the SQL query and deal with the 180th meridian problem, as well. The next step is to simply write the query in N1QL and execute it!

Here's the full implementation of this in Python. This example python program shows you how to retrieve the results from the N1QL query.



Given the location and area of interest:

- The location and distance give you the circle of interest.
- Given the same info, calculate the bounding box and determine its corner geolocation.
- Determine if this bounding box crosses 180th meridian or not.
- The N1QL query formed has two sets of predicates.
    - Determine all the points within the bounding box.

    - From the qualified items in the step above, filter the points falling outside the circle.
- You have the result now!

Let's look at the main portion of this.

```
    # Parameters
    # loc = GeoLocation.from_degrees(37.21, -123.38) # Union Square, San Francisco
    # distance = 10  # kilometer
    # Get the latitude, longitude and distance
    lat =  float(sys.argv[1])
    lon =  float(sys.argv[2])
    distance =  float(sys.argv[3])
    # Calculate the radians.
    loc = GeoLocation.from_degrees(lat, lon)
    # Calculate the bounding box for the location and given distance
    SW_loc, NE_loc = loc.bounding_locations(distance)
    # Are we crossing the 180th meridian or not
    meridian180condition = (" OR " if (SW_loc.rad_lon > NE_loc.rad_lon) else " AND ")
    # Generate the query
    query = "SELECT country, state, city, name, geo FROM `beer-sample` WHERE " + \
    "(RADIANS(geo.lat) >= " + str(SW_loc.rad_lat) + " and RADIANS(geo.lat) <= " +
str(NE_loc.rad_lat) + ") and " \
    "(RADIANS(geo.lon) >= " + str(SW_loc.rad_lon) +  meridian180condition + " RADIANS(geo.lon) <=
" + str(NE_loc.rad_lon) + ")"  \
    + " AND " \
    " acos(sin( RADIANS(" + str(loc.deg_lat) + ")) * sin (RADIANS(geo.lat)) + cos( RADIANS(" +
str(loc.deg_lat) +  " )) " \
    + " * cos(RADIANS(geo.lat)) * cos (RADIANS(geo.lon) - RADIANS( " + str(loc.deg_lon) + "))) <=
" + str(distance/6371.0)
    print query
```

Yes, it's that simple.

- Given the location, lat, long, and distance, calculate the bounding box.
- Execute the SQL with the expression to get the results.

Now, this query executes in about 356 milliseconds. You say, "That's too much for my customer wanting to get a cold one really quickly!" I agree. Let's look at the query plan.

```
        "~children": [
          {
            "#operator": "PrimaryScan",
            "index": "beer_primary",
            "keyspace": "beer-sample",
            "namespace": "default",
            "using": "gsi"
          },
```

PrimaryScan is never a good approach to get the best performance. As that data grows, the time taken will grow more than linearly, consuming additional network, CPU, memory resources, etc. It'll increase the latency and reduce throughput.

Let’s look at the query we used again.

```
SELECT country, state, city, name, geo, address
FROM `beer-sample`
WHERE (RADIANS(geo.lat) >= 0.658234696881
      AND RADIANS(geo.lat) <= 0.660743280942)
AND   (RADIANS(geo.lon) >= -2.13805724718
      AND  RADIANS(geo.lon) <= -2.13488305105)
AND   acos(sin( RADIANS(37.7859357)) * sin (RADIANS(geo.lat)) +
      cos( RADIANS(37.7859357 ))  * cos(RADIANS(geo.lat)) *
      cos (RADIANS(geo.lon) - RADIANS( -122.4107226)))
       <= 0.00125568984461
```

In this query, the first two expressions simply operate on geo.lat, geo.lon with the RADIANS() function on the values. We can create a GSI Index on it.

```
CREATE INDEX idx_brewery_geo_latlon ON
      `beer-sample`(RADIANS(geo.lat), RADIANS(geo.lon));
```

Now, let’s execute the same query again to see the performance.

```
{
   "requestID": "8d85b9e0-1c65-4e88-835d-2e1d5a7a52b7",
   "signature": {
       "address": "json",
       "city": "json",
       "country": "json",
       "geo": "json",
       "name": "json",
       "state": "json"
   },
   "results": [
       {
           "address": [
                "1195-A Evans Avenue"
            ],
            "city": "San Francisco",
            "geo": {
                "accuracy": "RANGE_INTERPOLATED",
                "lat": 37.7387,
                "lon": -122.381
            },
            "name": "Speakeasy Ales and Lagers"
        },
....
    "status": "success",
    "metrics": {
        "elapsedTime": "8.341ms",
        "executionTime": "8.311ms",
        "resultCount": 9,
        "resultSize": 3405
    }
}
```

Wow, that's 41 times improvement for this meager data set! Imagine the improvements you'll see for an even larger data set. It's like reducing a six-hour flight to eight minutes!

Again, referring back to the paper on geo locations, we know the way to calculate the distance between two geo locations:

$$\text{dist} = \arccos(\sin(\text{lat}_1) \cdot \sin(\text{lat}_2) + \cos(\text{lat}_1) \cdot \cos(\text{lat}_2) \cdot \cos(\text{lon}_1 - \text{lon}_2)) \cdot R$$

R is the radius of the earth.

We simply do that calculation in the SELECT statement to calculate the distance and then orbit the distance (expression is aliased to distance in the SELECT clause).

```
SELECT name, geo, address, (acos(sin( RADIANS(37.7859357)) * sin
(RADIANS(geo.lat)) +
      cos( RADIANS(37.7859357 ))  * cos(RADIANS(geo.lat)) *
      cos (RADIANS(geo.lon) - RADIANS( -122.4107226))) * 6371) distance
FROM `beer-sample`
WHERE (RADIANS(geo.lat) >= 0.658234696881
      AND RADIANS(geo.lat) <= 0.660743280942)
AND   (RADIANS(geo.lon) >= -2.13805724718
      AND  RADIANS(geo.lon) <= -2.13488305105)
AND   acos(sin( RADIANS(37.7859357)) * sin (RADIANS(geo.lat)) +
      cos( RADIANS(37.7859357 ))  * cos(RADIANS(geo.lat)) *
      cos (RADIANS(geo.lon) - RADIANS( -122.4107226)))
       <= 0.00125568984461
ORDER BY distance asc
LIMIT 5;
[
 {
   "address": [
     "661 Howard Street"
   ],
   "distance": 0.9435275834385553,
   "geo": {
     "accuracy": "ROOFTOP",
     "lat": 37.7855,
     "lon": -122.4
   },
   "name": "ThirstyBear Brewing"
 },
 {
   "address": [],
   "distance": 1.426534124430217,
   "geo": {
     "accuracy": "APPROXIMATE",
     "lat": 37.7749,
     "lon": -122.419
   },
   "name": "Shmaltz Brewing Company"
 },
 {
   "address": [],
   "distance": 1.426534124430217,
   "geo": {
     "accuracy": "APPROXIMATE",
     "lat": 37.7749,
     "lon": -122.419
   },
   "name": "Golden Gate Park Brewery"
 },
 {
   "address": [
     "650 Fifth Street #403"
   ],
   "distance": 1.6034394307234021,
   "geo": {
     "accuracy": "ROOFTOP",
     "lat": 37.7756,
     "lon": -122.398
   },
   "name": "Big Bang Brewery (Closed)"
 },
 {
   "address": [
     "563 Second Street"
   ],
   "distance": 1.6036323776739865,
   "geo": {
     "accuracy": "ROOFTOP",
     "lat": 37.7825,
     "lon": -122.393
   },
   "name": "21st Amendment Brewery Cafe"
 }
]
```

3. Give Me a Sorted List of These Breweries

The previous query will give you the qualifying nearest breweries. Now, you want additional filters like ratings, cost, etc. The additional requirements are bread and butter requirements for B-Tree indexes. Simply add new fields to your index.

```
CREATE INDEX idx_beer_sample_loc_rating_cost on `beer-sample`
   (RADIANS(geo.lat), RADIANS(geo.lon), ratings, cost)
   WHERE type = 'brewery';
```

Now, let's look for the nearest breweries with a 4-star rating and a cost level of "medium."

```
SELECT country, state, city, name, geo, address ,
      (acos(sin( RADIANS(37.7859357)) * sin (RADIANS(geo.lat)) +
            cos( RADIANS(37.7859357 ))  * cos(RADIANS(geo.lat)) *
            cos (RADIANS(geo.lon) - RADIANS( -122.4107226))) * 6371) AS distance
FROM `beer-sample`
WHERE (RADIANS(geo.lat) >= 0.658234696881
      AND RADIANS(geo.lat) <= 0.660743280942)
AND   (RADIANS(geo.lon) >= -2.13805724718
      AND  RADIANS(geo.lon) <= -2.13488305105)
AND   acos(sin( RADIANS(37.7859357)) * sin (RADIANS(geo.lat)) +
      cos( RADIANS(37.7859357 ))  * cos(RADIANS(geo.lat)) *
      cos (RADIANS(geo.lon) - RADIANS( -122.4107226)))
       <= 0.00125568984461
AND ratings = 4
AND cost = 'medium'
AND type = 'brewery'
ORDER BY distance ASC;
```

See the explanation below. See the span information. The index is the filtering location, rating, and cost in one big swoop! The more filtering the index does, the faster you get your results. That improves both latency, throughput, and utilization. Finally, the query simply retrieves the qualified breweries, calculates the specific distance, and sorts them. Use this technique if you need to incorporate spatial search on your app.

Maybe your boss will buy you a cold one for it!

```
{
  "#operator": "Sequence",
  "~children": [
    {
      "#operator": "IndexScan",
      "index": "idx_beer_sample_loc_rating_cost",
      "index_id": "bf5bed9f5b237493",
      "keyspace": "beer-sample",
      "namespace": "default",
      "spans": [
        {
          "Range": {
            "High": [
              "0.660743280942",
              "(-2.13488305105)",
          "4",
          "\"medium\""
        ],
        "Inclusion": 3,
        "Low": [
          "0.658234696881",
          "(-2.13805724718)",
          "4",
          "\"medium\""
        ]
      }
    }
  ],
  "using": "gsi"
},
```

References

- Couchbase
- N1QL
- Finding Points Within a Distance of a Latitude/Longitude Using Bounding Coordinates

Monitoring and Tooling

Mastering the Couchbase N1QL Shell: Connection Management and Security

Author: Isha Kandaswamy

This three-part series discusses the new and improved cbq shell. Cbq is the command line shell that helps you write and run N1QL queries and scripts interactively. The N1QL query engine parses, interprets, plans and executes each N1QL statement and returns the results in JSON. This response is available to the shell, which then displays the result. All shell commands start with a backslash, " \ ". All other commands are treated as N1QL statements. It is important to note that all cbq commands are case insensitive. Command line options, on the other hand, are case sensitive.

The command line options and shell commands that help with scripting can be classified into multiple categories. In this first article of three, we shall talk about Connection Management and Security. The examples provided are specific to Couchbase Server 4.5. Future versions will introduce additional options, commands, and features.

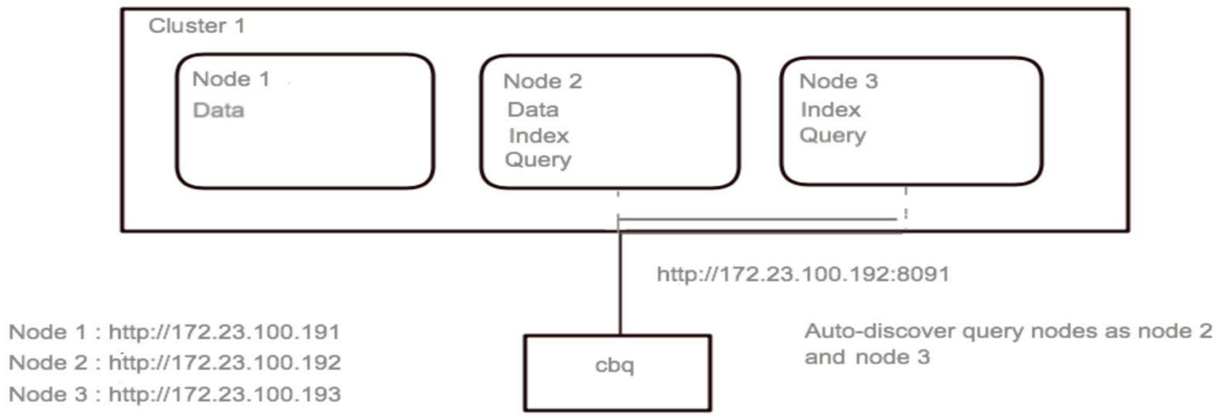
Connection Management

Couchbase can be deployed in multiple configurations. You can have a single node with all the services enabled on that node, or have a multi-node cluster with each node running different services, such as data, query, and/or index. The N1QL shell, cbq, allows you to connect to either a cluster endpoint or a single query service instance running on a specific node. This can be achieved in three ways.

- Command line option -e / --engine

- Command line argument, ./cbq <url>(as the last argument to the shell. Details below)
- Shell command \CONNECT when within a shell session (Details below)

Couchbase Server uses specific network ports for communication. Port 8091, the Administration Port, and 8093, the query service port, are defaults. In order to connect to the cluster, the url will contain port 8091. To connect to a specific query service in order to run a query on that node, the url will contain port 8093.


Cluster 1
Node 1
Data
Node 2
Data
Index
Query
Node 3
Index
Query
http://172.23.100.192:8091
Node 1 : http://172.23.100.191
Node 2 : http://172.23.100.192
Node 3 : http://172.23.100.193
cbq
Auto-discover query nodes as node 2 and node 3

Suppose you have a multi-node cluster with a mix of services among the nodes-data services, query services, and index services. The cbq shell can connect to any node in the cluster and automatically discover the query services across the cluster. As a user, you do not need to know or care about which nodes are running query services. If you connect to the cluster (details below), the cbq shell will round robin your queries among all the query services in the cluster. This allows you to load balance work among query nodes.

In the diagram above, we have a three node cluster. Node 1 (172.23.100.191) is configured as a data node, Node 2 (172.23.100.192) with all three services (Data, Index and Query) and Node 3 (172.23.100.193) with just Index and Query.

When we use the shell to connect to this cluster, we pass in the URL to a cluster endpoint, namely http://172.23.100.191:8091 as follows:

```
./cbq http://172.23.100.191:8091
```

The shell will automatically connect and discover that Node 2 and Node 3 are running query services, and then round robin the queries across these nodes. This saves us the trouble of having to identify query services within a large cluster.

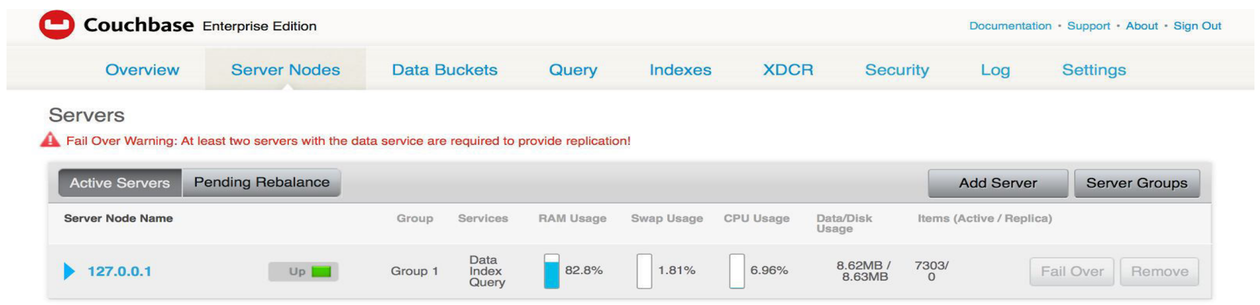
Remember that we can use any node in the cluster:

```
./cbq http://172.23.100.192:8091
```

```
./cbq http://172.23.100.193:8091
```

Another advantage of the shell being cluster-aware is that when nodes are added or removed, the shell will continue to auto detect query nodes, and queries will automatically be redirected to the current query nodes in the cluster.

Suppose we have a single node Couchbase deployment that is provisioned with all three services - data, query, and index. This can be seen in the diagram below.



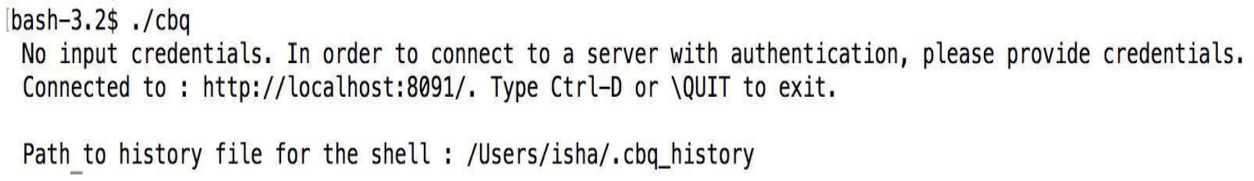
Fail Over Warning: At least two servers with the data service are required to provide replication!

Server Node Name		Group	Services	RAM Usage	Swap Usage	CPU Usage	Data/Disk Usage	Items (Active / Replica)
127.0.0.1	Up	Group 1	Data Index Query	82.8%	1.81%	6.96%	8.62MB / 8.63MB	7303/ 0

When bringing up cbq, if we don't pass in an argument or a -e option, then we are automatically connected to the default cluster IP, namely localhost and port 8091 or "http://localhost:8091". These are all equivalent:

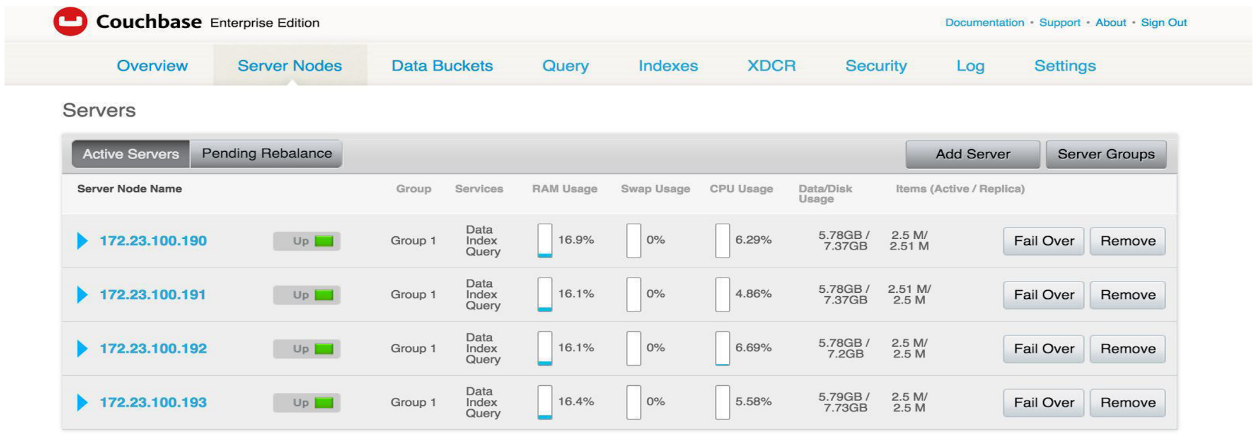
- `./cbq`
- `./cbq -e=http://localhost:8091`
- `./cbq http://localhost:8091`
- `cbq > \CONNECT http://localhost:8091;`

This being a single node deployment of Couchbase, the shell detects the local query service and queries against that. In the message below, the absence of credentials is not an error, since this server does not have any password-protected buckets. (Accessing such buckets will be discussed in the next section.)



```
bash-3.2$ ./cbq
 No input credentials. In order to connect to a server with authentication, please provide credentials.
 Connected to : http://localhost:8091/. Type Ctrl-D or \QUIT to exit.

 Path to history file for the shell : /Users/isha/.cbq_history
```

We can also explicitly connect to a specific query node. In the image below, we see a four node cluster.



Server Node Name		Group	Services	RAM Usage	Swap Usage	CPU Usage	Data/Disk Usage	Items (Active / Replica)		
172.23.100.190	Up 🟩	Group 1	Data Index Query	16.9%	0%	6.29%	5.78GB / 7.37GB	2.5 M/ 2.51 M	Fail Over	Remove
172.23.100.191	Up 🟩	Group 1	Data Index Query	16.1%	0%	4.86%	5.78GB / 7.37GB	2.51 M/ 2.5 M	Fail Over	Remove
172.23.100.192	Up 🟩	Group 1	Data Index Query	16.1%	0%	6.69%	5.78GB / 7.2GB	2.5 M/ 2.5 M	Fail Over	Remove
172.23.100.193	Up 🟩	Group 1	Data Index Query	16.4%	0%	5.58%	5.79GB / 7.73GB	2.5 M/ 2.5 M	Fail Over	Remove

Say we want to connect to the query node on 172.23.100.192, then we can issue the shell command

```
./cbq -e=http://172.23.100.192:8093
```

OR

```
cbq >  \CONNECT http://172.23.100.192:8093;
```

This will now allow us to run all statements against the query service on Node 172.23.100.192.

The URL that we connect to is made up of two components, the prefix and a port number. The prefix can be any valid IP or hostname. The full URL consists of a scheme followed by host and a port.

For the \CONNECT command, if the user is already connected, they are disconnected and connected to the new endpoint. We can close an existing connection without exiting the shell using the \DISCONNECT command. This also allows us to switch between different connections in a given session. If the shell is not connected to any endpoint, then \DISCONNECT throws an error that the shell is not connected.

One option when trying to run the same set of queries between multiple Couchbase clusters (made easy by the history and scripting features of the shell, which will be discussed in a later article in this series), is to bring up the shell without any connection, using the -ne or -no-engine option. Then we can use the \CONNECT command to switch between the different clusters.

Security

Couchbase Server provides security features that allow administrators to ensure secure deployment.

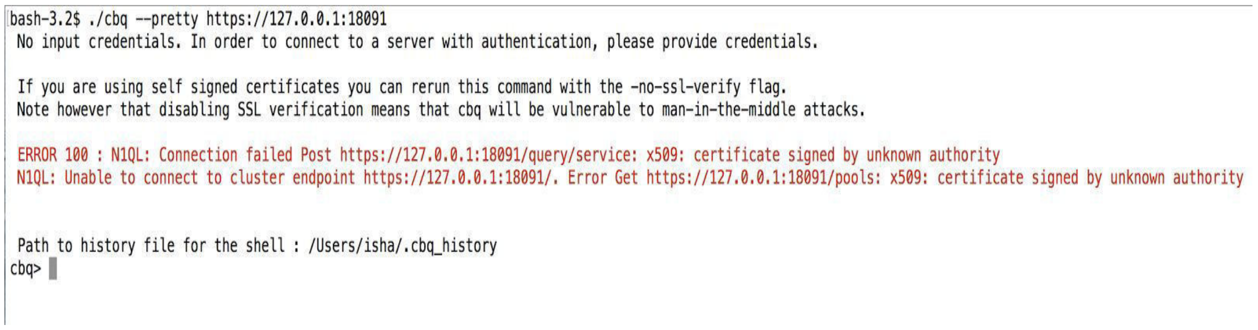
You can use these features with cbq shell in order to interact with Couchbase securely.

cbq provides a set of command line options in order to connect securely to any Couchbase cluster. (Refer to the table below to see all the available options.)

When using SSL secure transport (https:// URL) to connect from the shell, we need to take into account the two kinds of certificates that Couchbase supports – self-signed certificates and CA (Certificate Authority) signed certificates. Once we connect using SSL, we have two courses of action. Either we can skip verifying the certificates returned from the cluster or accept a certificate and verify it. In order to skip verification of the certificates, we use the --no-ssl-verify option. cbq will issue a warning " Disabling SSL verification means that cbq will be vulnerable to man-in-the-middle attacks."

```
./cbq --no-ssl-verify https://172.23.107.159:18091
```

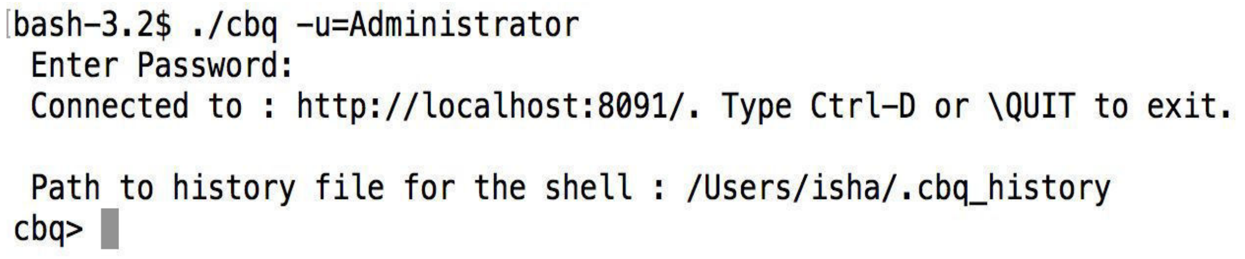
When using SSL-encrypted connections using a self-signed certificate, we need to specify this flag. If not, cbq will throw an error, since it assumes that the certificate is signed by an unknown authority (X509: certificate signed by unknown authority).



```
bash-3.2$ ./cbq --pretty https://127.0.0.1:18091
 No input credentials. In order to connect to a server with authentication, please provide credentials.

 If you are using self signed certificates you can rerun this command with the -no-ssl-verify flag.
 Note however that disabling SSL verification means that cbq will be vulnerable to man-in-the-middle attacks.

 ERROR 100 : N1QL: Connection failed Post https://127.0.0.1:18091/query/service: x509: certificate signed by unknown authority
 N1QL: Unable to connect to cluster endpoint https://127.0.0.1:18091/. Error Get https://127.0.0.1:18091/pools: x509: certificate signed by unknown authority


 Path to history file for the shell : /Users/isha/.cbq_history
cbq> 
```

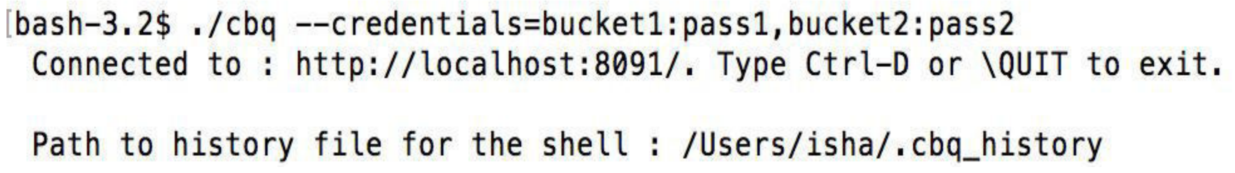
An option to pass in certificates, –cacert, will be added to the shell in the future. This will allow the user to specify a CA certificate that will then be used to verify the identity of the server.

In general, even for unencrypted connections, there are two types of credentials defined by the N1QL REST API, bucket credentials and role-based access credentials (for example Administrator). Each of these are passed in differently. In order to pass in role-based credentials we use the -u or --user option along with the -p or --password option. If the password option is not specified on startup, then the user is prompted for the password.



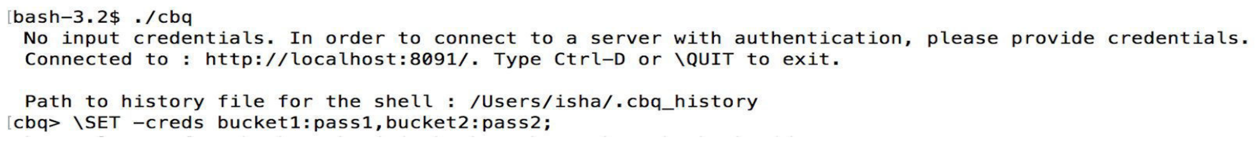
```
bash-3.2$ ./cbq -u=Administrator
 Enter Password:
 Connected to : http://localhost:8091/. Type Ctrl-D or \QUIT to exit.

 Path to history file for the shell : /Users/isha/.cbq_history
cbq> 
```

In order to pass in bucket credentials, we can use the --credentials option.



```
bash-3.2$ ./cbq --credentials=bucket1:pass1,bucket2:pass2
 Connected to : http://localhost:8091/. Type Ctrl-D or \QUIT to exit.

 Path to history file for the shell : /Users/isha/.cbq_history
```

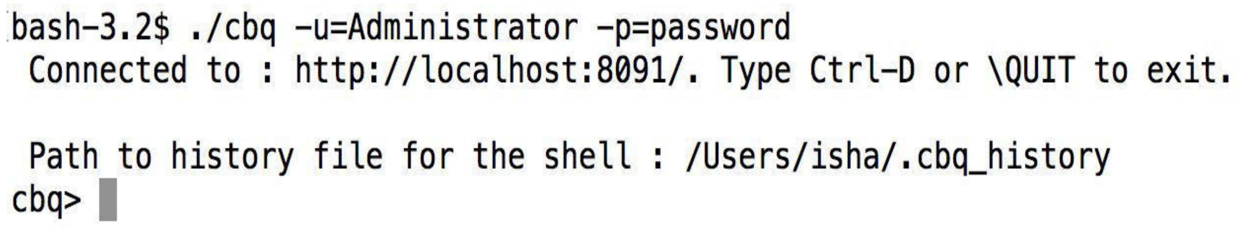
We can also set the creds parameter within a session.



```
bash-3.2$ ./cbq
 No input credentials. In order to connect to a server with authentication, please provide credentials.
 Connected to : http://localhost:8091/. Type Ctrl-D or \QUIT to exit.

 Path to history file for the shell : /Users/isha/.cbq_history
cbq> \SET -creds bucket1:pass1,bucket2:pass2;
```

Secure Connections to Protected Clusters With Mixed Configurations

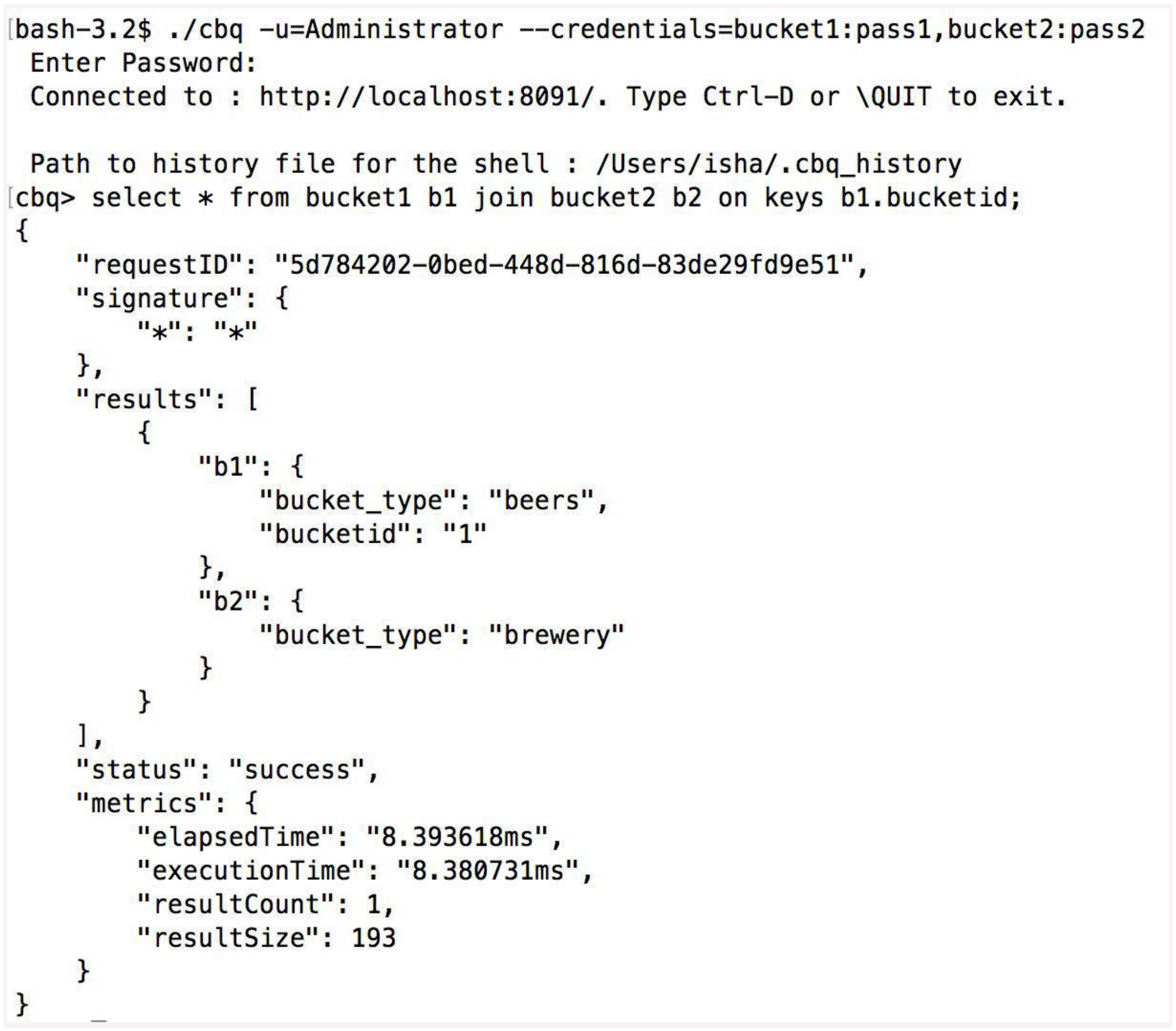
Couchbase data is stored in buckets. A bucket can optionally be protected with a bucket-specific password. The Couchbase cluster could contain only password-protected buckets, a mix of protected and unprotected buckets, or only unprotected buckets. In the cases where we have protected buckets in our Couchbase cluster, we need to pass in role based access credentials and / or bucket-specific credentials. Here is an example of role-based credentials.



```
bash-3.2$ ./cbq -u=Administrator -p=password
 Connected to : http://localhost:8091/. Type Ctrl-D or \QUIT to exit.

 Path to history file for the shell : /Users/isha/.cbq_history
cbq>
```

If the cluster deployed has a mix of protected and unprotected buckets, then we can pass in each buckets credentials individually as well. This can be done using the -c or --credentials option. When passing in these credentials, the format is a comma separated set of username:password. It is important to remember that any credentials passed in are valid only for that session. In Couchbase, the username of each bucket is the bucket name itself.

```
./cbq -c=beer-sample:pass1,travel-sample:pass2
```



```
[bash-3.2$ ./cbq -u=Administrator --credentials=bucket1:pass1,bucket2:pass2
 Enter Password:
 Connected to : http://localhost:8091/. Type Ctrl-D or \QUIT to exit.

 Path to history file for the shell : /Users/isha/.cbq_history
[cbq> select * from bucket1 b1 join bucket2 b2 on keys b1.bucketid;
{
    "requestID": "5d784202-0bed-448d-816d-83de29fd9e51",
    "signature": {
        "*": "*"
    },
    "results": [
        {
            "b1": {
                "bucket_type": "beers",
                "bucketid": "1"
            },
            "b2": {
                "bucket_type": "brewery"
            }
        }
    ],
    "status": "success",
    "metrics": {
        "elapsedTime": "8.393618ms",
        "executionTime": "8.380731ms",
        "resultCount": 1,
        "resultSize": 193
    }
}
```

This is very helpful when performing joins or writing subqueries, when the buckets accessed are password protected.

Suppose the user changes the password of a bucket, and wants to update the credentials that he/she has passed to the shell. This can easily be accomplished by using the N1QL REST API parameter creds. This value can be set using the \SET command.

```
cbq> \SET -creds beer-sample:pass1,travel-sample:newpass ;
```

But it is important to remember that the user needs to pass in the other credentials as well. Even if they haven't changed. This is because the value of the credentials is passed in for every query that is executed. Once we do this, we can now access the secure buckets as well.

In order to see what values have been set for the creds REST API parameter, we can use either the \ECHO command or the \SET command without any input arguments. However the value of the password is not displayed.

Using the \ECHO Command

```
cbq> \ECHO -creds;
```

```
beer-sample:*,travel-sample:*
```

Using the \SET Command

This will display all the different parameter settings within the shell session.

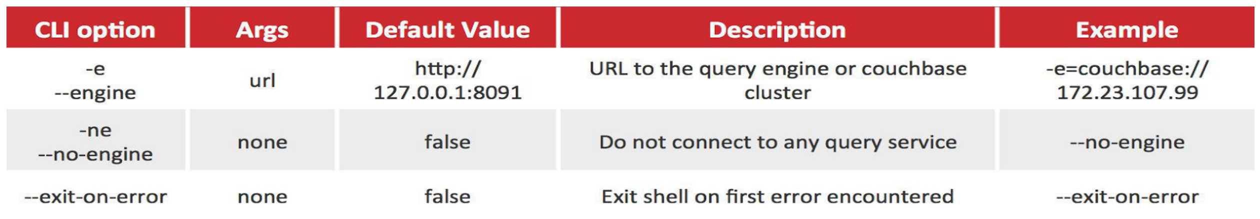
```
cbq> \SET;

Query Parameters :

Parameter name : creds

Value : [ "beer-sample:*, travel-sample:*" ]

…
```

Summary

Cbq is a powerful tool. It can manage connections to both a query service and full cluster, and can auto discover the query services in a Couchbase cluster. Queries are executed in a round-robin fashion against all the available query services. The shell also provides secure transport and authentication, so that users can securely access any data to which they are entitled.

Connection Management

Command Line Options



CLI option	Args	Default Value	Description	Example
-e --engine	url	http:// 127.0.0.1:8091	URL to the query engine or couchbase cluster	-e=couchbase:// 172.23.107.99
-ne --no-engine	none	false	Do not connect to any query service	--no-engine
--exit-on-error	none	false	Exit shell on first error encountered	--exit-on-error

Shell Commands

Command	Args	Description	Example
\CONNECT	url	URL to the query engine or couchbase cluster	cbq > \CONNECT couchbase:// 172.23.107.99; Connected to : http:// 172.23.107.99:8091. Type Ctrl-D or \QUIT to exit.
\DISCONNECT	none	Disconnect shell from the query service/ cluster	cbq > \DISCONNECT; Couchbase query shell not connected to any endpoint. Use \CONNECT command to connect.
\EXIT or \QUIT	none	Exit shell (Also Ctrl + D)	cbq> \EXIT;

Security

Command Line Options

CLI option	Args	Default Value	Description	Example
-u --user	username	none	Single username for logging into couchbase. If –p is not given then the user will be prompted for the password.	-u=Administrator
-p --password	password	none	Provides the corresponding password to the username. If username not present it displays an error.	-p=password
-c --credentials	Comma separated list of credentials separated by :	none	Bucket credentials	-c=beer-sample:password
--no-ssl-verify	none	false	Skip verification of certificates for https://	--no-ssl-verify

Query Performance Monitoring Made Easy With Couchbase N1QL

Author: Marco Greco

This article discusses the new monitoring capabilities of N1QL introduced in Couchbase Server 4.5 One of the major breakthroughs of Couchbase Server 4.0 was the introduction of a SQL-based query language specifically created to handle JSON documents: N1QL. Extolling the virtues of N1QL is not the aim of this article, but you can sample its capabilities here, here, and here.

What we are going to concentrate on here is how to use N1QL monitoring capabilities introduced in version 4.5, specifically with a goal to discover and fix poorly performing requests. Let's face it — even with all the good will in the world, there will be times when things don't work as expected, and when this happens you need tools to discover the culprit and investigate the erroneous behaviour. More importantly, and this is almost invariably overlooked, monitoring is not only for rainy days, but for good weather too! When the sun shines, you can use monitoring tools preventively to find the likely direction the next rain is going to come from.

But let's start from the beginning.

N1QL’s Monitoring Infrastructure

In Couchbase 4.5 N1QL exposes the active requests executing in the Query Service and the prepared statements dataset both via N1QL itself and via specific administrative REST endpoints. More interestingly, the query service can now track requests that have executed for longer than a predetermined amount of time in the completed requests log, again exposed via a REST endpoint and N1QL. All three endpoints and keyspaces report a host of useful request related data which, when aggregated, can give good insight as to how the N1QL service resources are being used.

Lifecycle of a N1QL Request

Before we delve into the nitty gritty of monitoring, it may be worth painting the relationship between the life of a N1QL request and the various monitoring datasets. A picture is worth a thousand words, so rather than describing in detail how things change in time, I’ll let some fancy infographics do the talking:

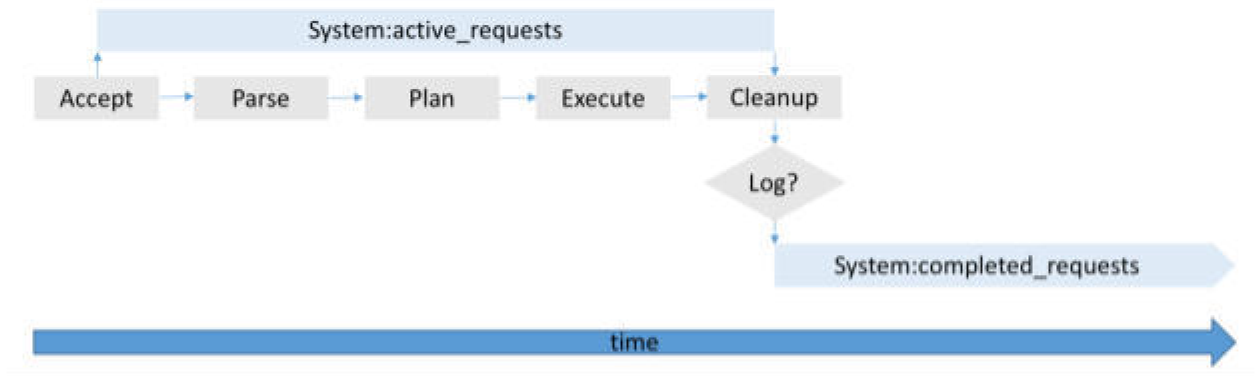


Fig. 1: Execution of a standalone request

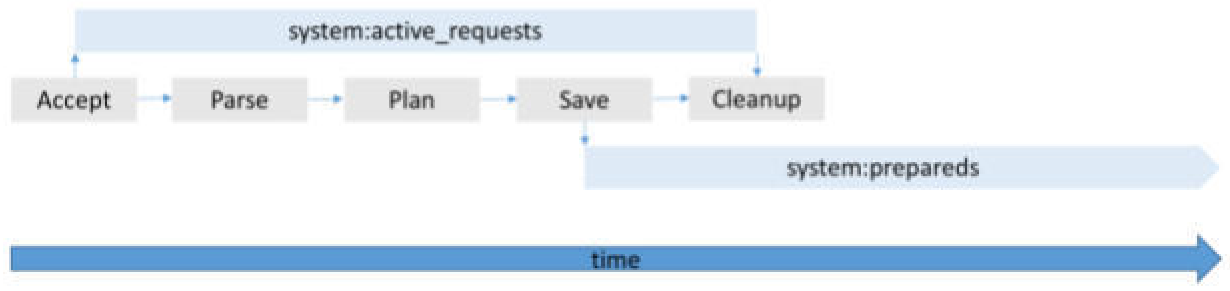


Fig. 2: Preparing a statement

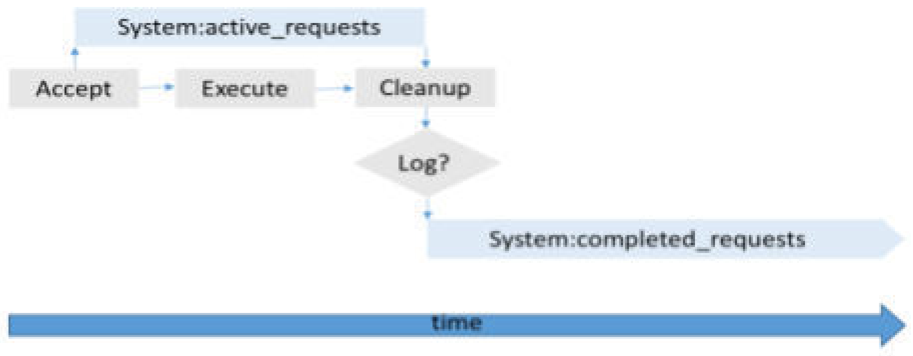


Fig. 3: Executing a prepared statement

REST Endpoints

- `http://<N1QL node IP address>:8093/admin/vitals`
- `http://<N1QL node IP address>:8093/admin/active_requests`
- `http://<N1QL node IP address>:8093/admin/completed_requests`
- `http://<N1QL node IP address>:8093/admin/prepareds`

To be clear, each endpoint only returns information collected from the specific query node in the URL accessed, and not the entire Couchbase cluster: to collect information from all nodes via REST, you will have to manually loop through all Query nodes.

You will note that an endpoint we have so far not mentioned has sneakily made its way through the previous endpoint list: /admin/vitals.

Vitals

Vitals, as the name implies, returns vital signs of the local query node.
Among the information provided is the node's uptime, local time, version, and more interesting stuff like memory and CPU usage, cores and allocated threads, garbage collector operation, request counts and request time percentiles.

```
curl -X GET http://localhost:8093/admin/vitals
```

Which produces documents of this sort:

```
{"uptime":"66h6m9.965439555s","local.time":"2016-11-04 11:55:31.340325632 +0000 GMT",
"version":"1.5.0","total.threads":213,"cores":8,"gc.num":8379306,
"gc.pause.time":"3.500846278s","gc.pause.percent":0,"memory.usage":6710208,
"memory.total":36982771064,"memory.system":52574456,"cpu.user.percent":0,
"cpu.sys.percent":0,"request.completed.count":61,"request.active.count":0,
"request.per.sec.1min":0,"request.per.sec.5min":0,"request.per.sec.15min":0,
"request_time.mean":"365.708343ms","request_time.median":"784.055µs",
"request_time.80percentile":"417.369525ms",
"request_time.95percentile":"2.347575326s",
"request_time.99percentile":"5.669627409s","request.prepared.percent":0}
```

The only supported method is GET, and the output is not pretty-formatted (by way of clarification, pretty is an attribute of JSON documents returned by the N1QL service which determines if the document is formatted or not and can be set at the service or request level to either true or false, but for admin endpoints, it is always set to false).

Active Requests

The active_request endpoints provide information about all active requests...

- `GET http://<N1QL node IP address>:8093/admin/active_requests`

`...individual active requests...`

- `GET http://<N1QL node IP address>:8093/admin/active_requests/<request id>`

...and, more importantly, the ability to cancel requests for whatever reason.

- `DELETE http://<N1QL node IP address>:8093/admin/active_requests/<request id>`

As an example

```
curl -X GET http://localhost:8093/admin/active_requests
```

Again, the resulting document is not pretty-formatted.

```
[{"elapsedTime":"1.525668112s","executionTime":"1.525635694s",
 "phaseCounts":{"Fetch":11280,"PrimaryScan":11716},
 "phaseOperators":{"Fetch":1,"PrimaryScan":1},
 "requestId":"0b1d33a7-db3f-4a8e-86ff-63aee969fbbe",
 "requestTime":"2016-11-04T12:03:13.25494174Z",
 "scanConsistency":"unbounded","state":"running",
 "statement":"select *  from `travel-sample`;"}]
```

Completed Requests

This endpoint reports the same information as active_requests, but for the completed requests cache.

- `GET http://<N1QL node IP address>:8093/admin/completed_requests`
- `GET http://<N1QL node IP address>:8093/admin/completed_requests/<request id>`
- `DELETE http://<N1QL node IP address>:8093/admin/completed_requests/<request id>`

Unlike active_requests, a DELETE on completed_requests only removes an entry from the cache. The completed request cache by default logs the last 4000 requests lasting more than 1 second that have run on the local node. The cache size and time threshold can be set via the /admin/settings endpoint via the "completed-limit" and "completed-threshold" fields, for example:

```
curl http://localhost:8093/admin/settings -u Administrator:password -d '{
"completed-limit": 10000 }'
```

Prepared Statements

...and again for prepared statements

- `GET http://<N1QL node IP address>:8093/admin/prepareds`
- `GET http://<N1QL node IP address>:8093/admin/prepareds/<prepared statement name>`
- `DELETE http://<N1QL node IP address>:8093/admin/prepareds/<prepared statement name>`

Getting Monitoring Information via System Keyspaces

A more interesting way to access the N1QL caches is via N1QL itself. For each of the admin REST endpoints (active_requests, completed_requests, prepareds) there is a corresponding keyspace in the system namespace. So, for example, active requests can be directly accessed from the query service via a simple

```
select * from system:active_requests;
```

While the REST API can provide monitoring information at a relatively low cost, using system keyspaces offers the opportunity to unleash all of N1QL's query capabilities over the monitoring information. We will explore a few more advanced examples to make the point. The examples below are specific to Couchbase Server 4.5.

Preparation

We will use the beer-sample bucket in order to provide some load for the following examples.

```
Create a secondary index on `beer-sample`(city)
```

Start up a few client applications preparing

```
    select *
     from `beer-sample`
     use keys["21st_amendment_brewery_cafe-amendment_pale_ale"];
```

and

```
    select *
     from `beer-sample`
     where city = "Lawton";
```

And then executing the statements above for, say, 10000 times per client.

Too much load on the node

Among the useful information reported by the system:active_requests keyspace there are two pieces of timing information:

- Elapsed time
- Execution time

The first starts being accrued as soon as the N1QL service becomes aware of the request, the second as soon as the request is being executed.

Under normal circumstances there should be very little difference in between the two times, so the following query...

```
    select avg(100*
              (str_to_duration(ElapsedTime) - str_to_duration(ExecutionTime))/
              str_to_duration(ElapsedTime)), count(*)
     from system:active_requests;
```

...should return only a few percentage points (say 1 to 3%):

```
{
    "requestID": "137b5777-d420-45e9-bbef-0b15197c9a93",
    "signature": {
        "$1": "number",
        "$2": "number"
    },
    "results": [
        {
            "$1": 3.210601953400783,
            "$2": 8
        }
    ],
    "status": "success",
    "metrics": {
        "elapsedTime": "1.938543ms",
        "executionTime": "1.910804ms",
        "resultCount": 1,
        "resultSize": 68
    }
}
```

The query itself should execute quite quickly. When the server is under stress, requests will most probably be queued before being executed and the ratio of elapsed to execution time will increase — to put it in perspective a value of 50 means that the request is spending just as much time queued as being executed. In general terms, once the node is under stress, the query above will plateau at a specific percentage point. On my development machine this is about 25% (although higher spikes will make an appearance from time to time), however, a tell tale sign of the node being under stress is that the elapsed to execution ratio of the query itself is quite high:

```
{
   "requestID": "8cafcd14-fbdb-48bd-a35d-9f0fb03752ae",
   "signature": {
       "$1": "number",
       "$2": "number"
   },
   "results": [
       {
           "$1": 35.770020645707156,
           "$2": 94
       }
   ],
   "status": "success",
   "metrics": {
       "elapsedTime": "29.999017ms",
       "executionTime": "3.864563ms",
       "resultCount": 1,
       "resultSize": 48
   }
}
```

A similar query can be run against the system:completed_requests keyspace. Again high execution to elapsed ratio is an indication that the system is under stress: long running queries on a node with spare capacity instead show a high execution time with a very queueing ratio.

Prepared Statements With Highest Total Execution Time

The system:prepareds keyspace returns average execution time and number of executions for each prepared statement.

Since the same statement can be prepared anonymously by several clients under several names, summing average service times multiplied by uses and grouped by statement text gives a good idea of the total execution cost of each statement.

```
select sum(str_to_duration(avgServiceTime)*uses/1000000000) ttime, statement
 from system:prepareds
 where avgServiceTime is not missing
 group by statement
 order by ttime;
```

The query above returns the overall execution time in seconds for each statement

```
[
 {
   "statement": "PREPARE SELECT * FROM `beer-sample` USE
KEYS[\"21st_amendment_brewery_cafe-amendment_pale_ale\"];",
   "ttime": 515.6239400000001
 },
 {
   "statement": "PREPARE select * from `beer-sample` where city = \"Lawton\";",
   "ttime": 1506.407181327
 }
]
```

Prepared Statements With Highest Execution Count

Extending the concept further, it's possible to determine the prepared statements that have been used the most.

For example...

```
select sum(uses) usecount, statement
 from system:prepareds
 where uses is not missing
 group by statement
 order by usecount desc;
```

...returns:

```
[
 {
   "statement": "PREPARE SELECT * FROM `beer-sample` USE
KEYS[\"21st_amendment_brewery_cafe-amendment_pale_ale\"];",
   "usecount": 200000
 },
 {
   "statement": "PREPARE select * from `beer-sample` where city = \"Lawton\";",
   "usecount": 5393
 }
]
```

Prepared Statements With Highest Prepare Count

...or the statements that have been prepared by the most clients:

```
select count(*) prepcount, statement
 from system:prepareds
 group by statement
 order by prepcount desc;
```

which yields:

```
[
 {
   "prepcount": 20,
   "statement": "PREPARE SELECT * FROM `beer-sample` USE
KEYS[\"21st_amendment_brewery_cafe-amendment_pale_ale\"];"
 },
 {
   "prepcount": 2,
   "statement": "PREPARE select * from `beer-sample` where city = \"Lawton\";"
 }
]
```

Requests Using Suboptimal Paths

Another important piece of information reported by both system:active_requests and system:completed_requests is the number of documents processed by each execution phase. This information is interesting for at least two different reasons. The first is, even though the system keyspaces do not report a request plan, and therefore individual query paths are not known, we can still draw a fair amount of information from the fact that specific phases are used. For example, primary scans are the N1QL equivalent of RDBMS's sequential scans - unless very small buckets are involved, where an ad hoc index adds no performance benefit, primary scans are best avoided. Similarly, you may have already heard that *covering indexes* (indexes whose keys encompass all the expressions in the select list) are preferable to non covering indexes, because when the index chosen is not covering, request execution will require a Fetch phase after the index scan, in order to access each document, which clearly requires further effort on the N1QL service part. The mere presence of a Fetch phase is a dead giveaway that covering indexes are not being used. The second reason, more obvious, is that document counts instantly give a sense of the cost of individual phases, and by extension, individual requests. We will now explore a few use cases exploiting phase counts.

Preparation

From cbq, execute a few.

```
select * from `travel-sample`;
```

Requests Using Primary Scans

As stated earlier, all we are looking for is requests where the PhaseCounts subdocuments has a PrimaryScan field:

```
select *
 from system:completed_requests
 where PhaseCounts.`PrimaryScan` is not missing;
```

Which in our case yields:

```
[
 {
   "completed_requests": {
     "ClientContextID": "02a9db12-1bb8-4669-b2c6-c6e1d50525d2",
     "ElapsedTime": "35.638999968s",
     "ErrorCount": 0,
     "PhaseCounts": {
       "Fetch": 31591,
       "PrimaryScan": 31591
     },
     "PhaseOperators": {
       "Fetch": 1,
       "PrimaryScan": 1
     },
     "RequestId": "ade384a9-367b-4241-8307-68cf923f9b8c",
     "ResultCount": 31591,
     "ResultSize": 107829252,
     "ServiceTime": "35.638946836s",
     "State": "completed",
     "Statement": "select * from `travel-sample`",
     "Time": "2016-11-10 13:26:53.80370308 -0800 PST"
   }
 },
 {
   "completed_requests": {
     "ClientContextID": "0afc6ce3-5c09-4fbd-9cec-6f8719a3dff1",
     "ElapsedTime": "42.179613979s",
     "ErrorCount": 0,
     "PhaseCounts": {
       "Fetch": 31591,
       "PrimaryScan": 31591
     },
     "PhaseOperators": {
       "Fetch": 1,
       "PrimaryScan": 1
     },
     "RequestId": "874d4493-af76-49e6-9b89-e9323c33915f",
     "ResultCount": 31591,
     "ResultSize": 107829252,
     "ServiceTime": "42.17955482s",
     "State": "completed",
     "Statement": "select * from `travel-sample`",
     "Time": "2016-11-10 13:27:45.641104619 -0800 PST"
   }
 },
 {
   "completed_requests": {
     "ClientContextID": "6850f34d-5d9c-49c1-91ac-88ce6265c5ac",
     "ElapsedTime": "36.769111048s",
     "ErrorCount": 0,
     "PhaseCounts": {
       "Fetch": 31591,
       "PrimaryScan": 31591
     },
     "PhaseOperators": {
       "Fetch": 1,
       "PrimaryScan": 1
     },
     "RequestId": "a6ebd56f-9562-4532-8d97-d26bacc2ce81",
     "ResultCount": 31591,
     "ResultSize": 107829252,
     "ServiceTime": "36.769063345s",
     "State": "completed",
     "Statement": "select * from `travel-sample`",
     "Time": "2016-11-10 13:26:05.510344804 -0800 PST"
   }
 }
]
```

Preparation

The example below requires a secondary index on `travel-sample`(type)

Execute the following query:

```
select id
 from `travel-sample`
 where type="route" and any s in schedule satisfies s.flight="AZ917" end;
```

Requests not Using Covering Indexes

In a similar way, we can find non covering indexes by stipulating that the requests we are looking for have both index scans and fetches in the PhaseCounts subdocument:

```
select *
```

```
 from system:completed_requests
 where PhaseCounts.`IndexScan` is not missing
   and PhaseCounts.`Fetch` is not missing;
```

Which, in our case yields:

```
[
 {
   "completed_requests": {
     "ClientContextID": "e5f0e69a-ff3e-47e4-90c7-881772c2e1b7",
     "ElapsedTime": "2.877312961s",
     "ErrorCount": 0,
     "PhaseCounts": {
       "Fetch": 24024,
       "IndexScan": 24024
     },
     "PhaseOperators": {
       "Fetch": 1,
       "IndexScan": 1
     },
     "RequestId": "12402dd8-430a-4c32-84c1-a59b88695e66",
     "ResultCount": 10,
     "ResultSize": 350,
     "ServiceTime": "2.8772514s",
     "State": "completed",
     "Statement": "select id from `travel-sample` where type=\"route\" and any
s in schedule satisfies s.flight=\"AZ917\" end;",
     "Time": "2016-11-10 13:10:00.04988112 -0800 PST"
   }
 }
]
```

Requests Using a Poorly Selective Index

Loosely speaking, Selectivity is that property of a predicate which indicates how good that predicate is at isolating those documents which are relevant for the query result set. In general terms, when using an index scan, we would like to have the scan return only the relevant document keys, and no more, so that later stages of the plan (eg, fetch or filter) only have to operate on relevant documents. A simple measure of poor selectivity of a predicate is the total number of keys (or documents) to be considered by the predicate divided by the number of documents that satisfy the predicate - the higher the number, the worse the selectivity (in general terms selectivity is defined as the number qualifying items divided by the total number of items).

Clearly, when an index path is in use, we would like that the relevant filters are as relevant as possible, and we can determine if this is the case by just measuring the number of documents fetched against the number of documents returned. The query below finds requests that do use indexes and that have a number of fetches to results ratio greater than ten. This means that the request fetches ten documents for each document that satisfies all the predicates in the where clause: you threshold may very well be different (and lower, I would suggest!).

```
select PhaseCounts.`Fetch` / ResultCount efficiency, *
 from system:completed_requests
 where PhaseCounts.`IndexScan` is not missing
  and PhaseCounts.`Fetch` / ResultCount > 10
 order by efficiency desc;
```

In our case, this yields:

```
[
 {
   "completed_requests": {
     "ClientContextID": "e5f0e69a-ff3e-47e4-90c7-881772c2e1b7",
     "ElapsedTime": "2.877312961s",
     "ErrorCount": 0,
     "PhaseCounts": {
       "Fetch": 24024,
       "IndexScan": 24024
     },
     "PhaseOperators": {
       "Fetch": 1,
       "IndexScan": 1
     },
     "RequestId": "12402dd8-430a-4c32-84c1-a59b88695e66",
     "ResultCount": 10,
     "ResultSize": 350,
     "ServiceTime": "2.8772514s",
     "State": "completed",
     "Statement": "select id from `travel-sample` where type=\"route\" and any
s in schedule satisfies s.flight=\"AZ917\" end;",
     "Time": "2016-11-10 13:10:00.04988112 -0800 PST"
   },
   "efficiency": 2402.4
 }
]
```

Determining Requests With the Highest Cost

As we were saying earlier, PhaseCounts give a ready sense of individual phases cost. Sum them up, and you can get a measure of the cost of individual requests. In the statement below we turn the PhaseCounts subdocument into an array in order to tot up the cost:

```
select Statement, array_sum(object_values(PhaseCounts)) cost
 from system:completed_requests
 order by cost desc;
```

This returns the statements we run earlier in order of decreasing cost:

```
[
       {
           "Statement": "select * from `travel-sample`",
           "cost": 63182
       },
       {
           "Statement": "select * from `travel-sample`",
           "cost": 63182
       },
       {
           "Statement": "select * from `travel-sample`",
           "cost": 63182
       },
       {
           "Statement": "select id from `travel-sample` where type=\"route\"
and any s in schedule satisfies s.flight=\"AZ917\" end",
           "cost": 48048
       }
]
```

Aggregating Execution Times for Completed Requests

Determining total execution times for individual statement texts is not only the preserve of prepared statements. A very similar technique can be used on system:completed_requests. Further, this can be combined with filtering for suboptimal paths. The following query looks for requests having used a primary scan, returning several aggregates for each individual statement:

```
select count(*) num,
      min(ServiceTime) minEx, max(ServiceTime) maxEx,
      avg(str_to_duration(ServiceTime)/1000000000) avgEx,
      Statement
 from system:completed_requests
 where PhaseCounts.PrimaryScan is not missing
 group by Statement;
```

Note that execution times are strings, and in principle they should be turned into durations in order to be aggregated properly. This is what we are doing for avg():

```
[
 {
   "Statement": "select * from `travel-sample`",
   "avgEx": 38.19585500033333,
   "maxEx": "42.17955482s",
   "minEx": "35.638946836s",
   "num": 3
 }
]
```

Canceling Requests

We have seen before that a rogue request can be cancelled using the REST API. This can be achieved through N1QL directly by simply deleting from system:active_requests:

```
delete from system:active_requests
 where RequestId="d3b607d4-1ff1-49c8-994e-8b7e6828f6a0";
```

(replace the request id in quotes with the RequestId of the request you wish to cancel).

But while REST can operate only on a request by request basis, N1QL allows for more creative operations. For example, we can stop all executing instances of a specific statement:

```
delete from system:active_requests
 where Statement = "select * from `travel-sample`";
```

Similarly, we can stop all instances of a specific prepared statement:

```
delete from system:active_requests
 where `Prepared.Name` = "s1";
```

Further, we can prevent all instances of the culprit prepared statement from executing again (at least, until it is prepared again!) by removing said instances from the prepared statement cache:

```
delete from system:prepareds
 where `statement` = "prepare select * from `travel-sample`";
```

In all of the examples above, please be mindful that we are using an equality predicate on the statement text: if you have several variations of the same statement (eg lower case versus upper case, or different amounts of spacing, or…), you'll either have to match them individually, or have a more comprehensive filter.

Summary

N1QL offers a wide array of powerful monitoring tools which can quickly identify poor performing statements or statements that are likely to become a bottleneck. Examples have been provided to detect several different types of requests which are interesting for resource planning purposes.

Author’s Note

The examples provided are specific to Couchbase Server 4.5. Future versions will introduce additional system keyspaces and contents, and some field names may be changed to a consistent format.

Migration and Competition

ELT Processing With Couchbase and N1QL

Author: Manuel Hurtado

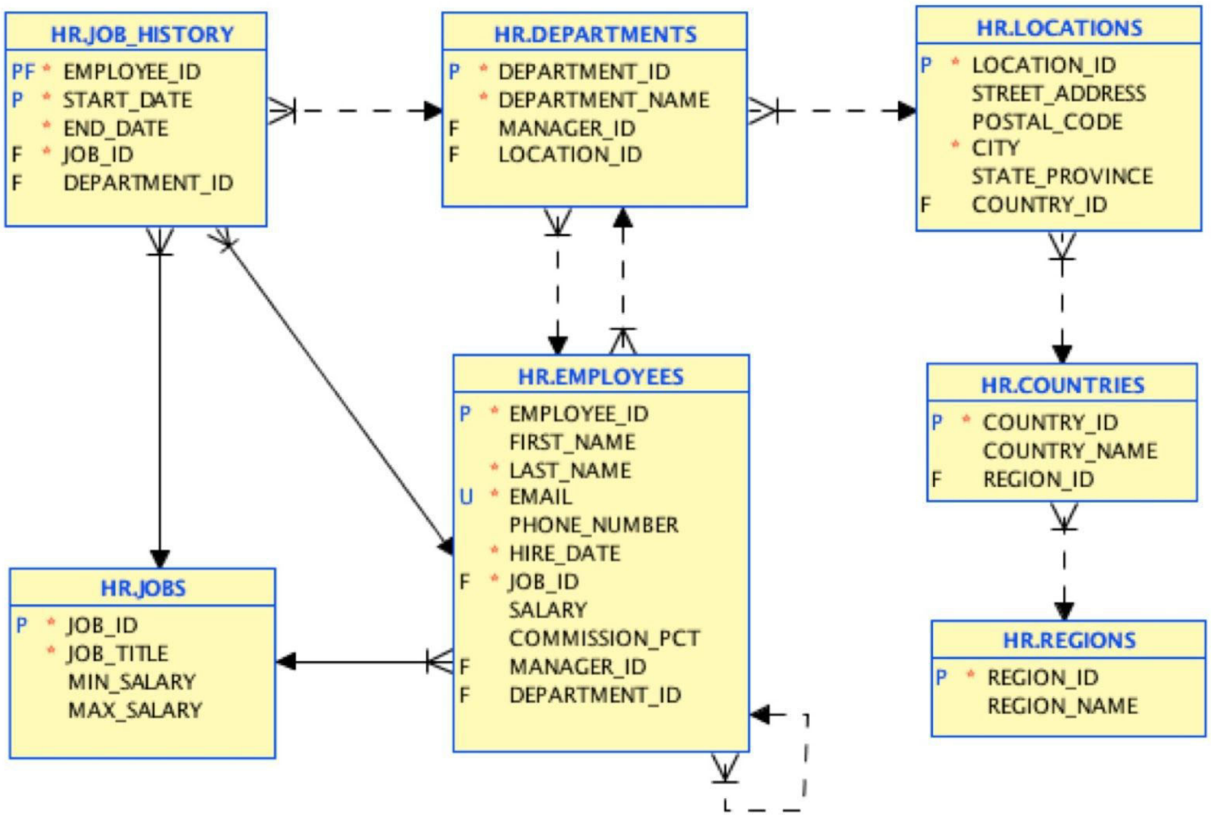
Moving data between data sources is one of the key activities in data integration projects. Traditionally, techniques around data movement have been part of Data Warehouse, BI, and analytics. More recently, Big Data, Data Lakes, and Hadoop, have been frequent players in this area. In this entry, we'll discuss how Couchbase's N1QL language can be used to make massive manipulation on the data in this kind of scenarios.

First, let us remember the two classical approaches when doing data movement:

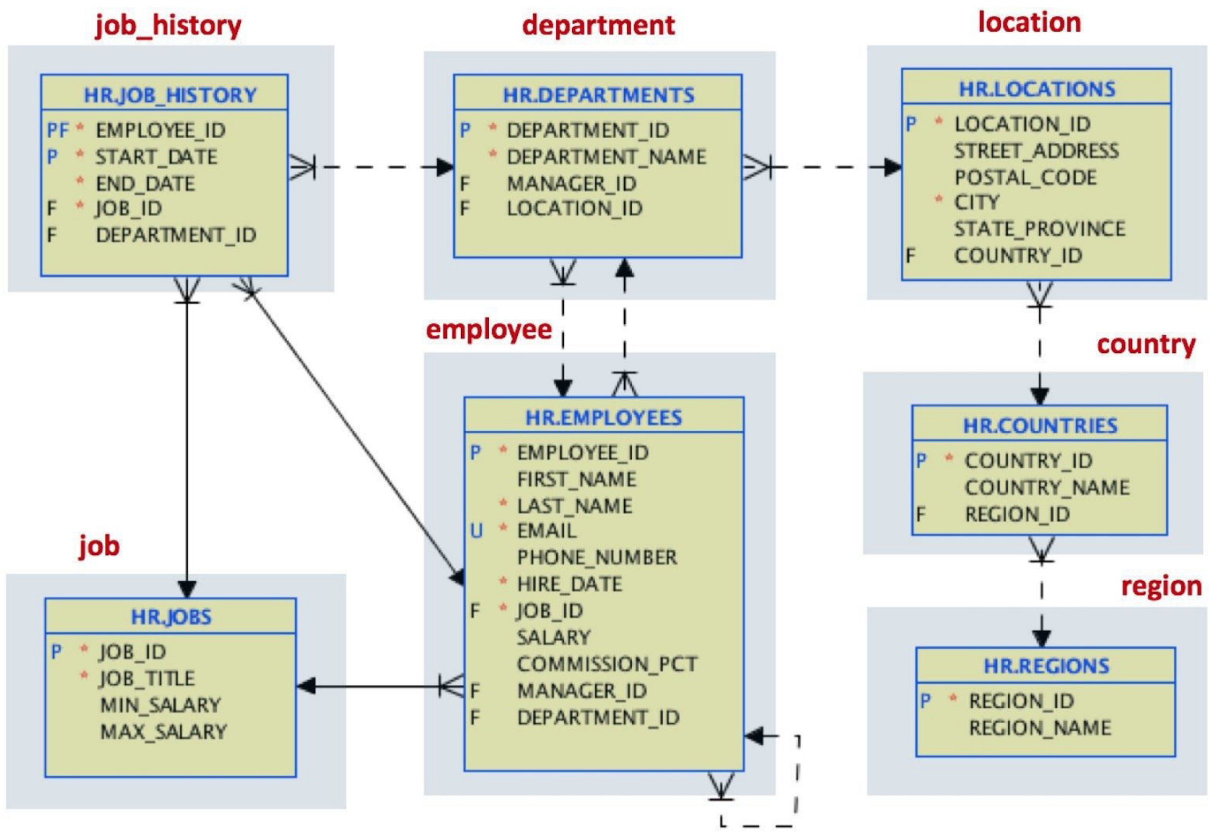
ETL (Extract-Transform-Load): With this model, the data is **extracted** (from the original data source), **transformed** (data is reformatted to fit in the target system), and **loaded** (in the target data store).
ELT (Extract- Load-Transform): With this model, the data is **extracted** (from the original data source), then **loaded** in the same format in the target system. Then, we do a **transformation** in the target system to obtain the desired data format.

We will focus on an **ELT** exercise in this example. Let's do a simple export from a relational database and load the data into Couchbase. We will use Oracle Database as our input data source, with the classical HR schema example built in, which models a Human Resources department.

This is the source data model:



In the first step, we will load the data with the same structure. There is a free tool you can use to perform this initial migration here. At the end, we will have JSON documents mapping this table model:

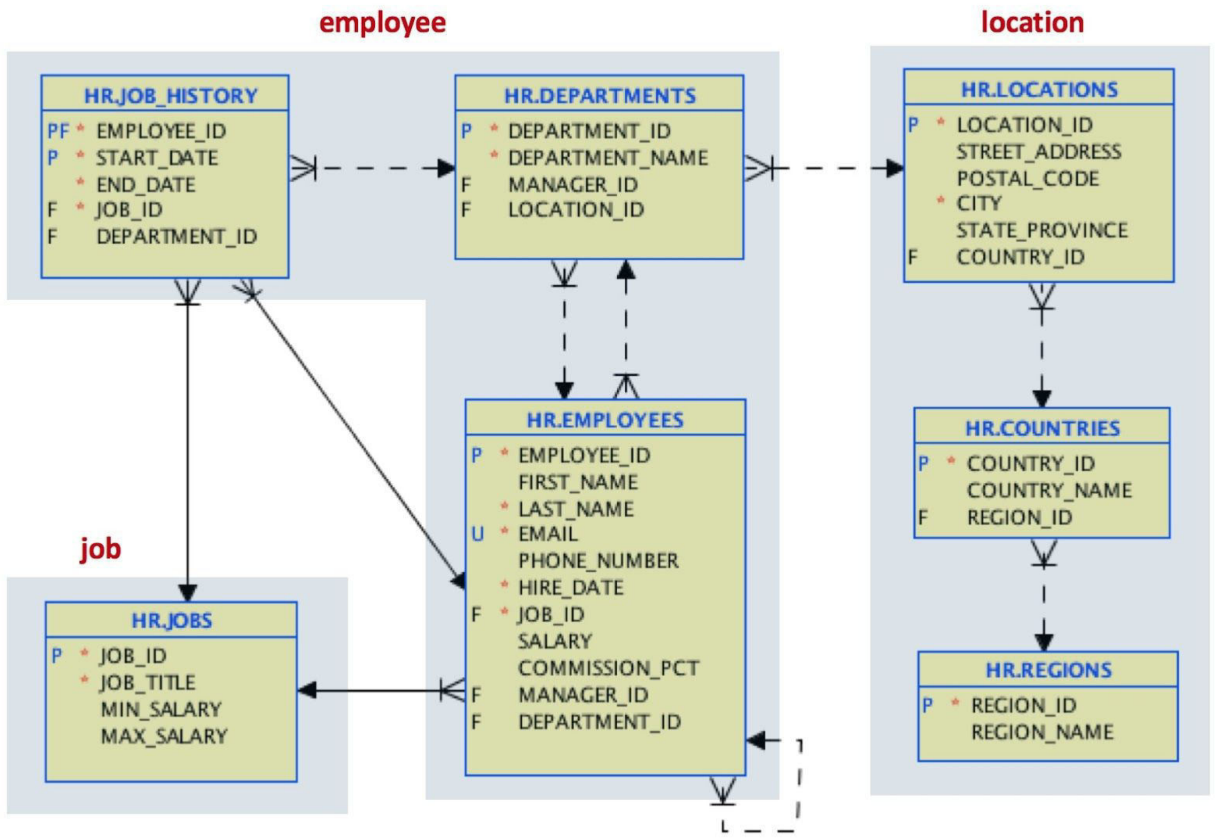


For example, a location document will look like this:

```
{
    "street_address": "2017 Shinjuku-ku",
    "city": "Tokyo",
    "state_province": "Tokyo Prefecture",
    "postal_code": "1689",
    "type": "locations",
    "location_id": 1200,
    "country_id": "JP"
}
```

This was an easy first step. However, this mapping table-to-document is often a sign of bad design in the NoSQL world. In NoSQL, it's common to de-normalize your data in favor of a more direct access path, embedding referenced data. The goal is to minimize database interactions and joins, looking for the best performance.

Let's assume that our use case is driven by a frequent access to the whole job history for employees. We decide to change our design to this one:



For locations, we are joining in a single location document the referenced data for both country and region.

For the employee document, we will embed the department data and include an array with the whole job history for each employee. This array support in JSON is a good improvement over foreign key references and joins in the relational world.

For the job document, we will maintain the original table structure.
Now that we have **extracted** and **loaded** the data, we will **transform** into this model to finish our **ELT** example. How can we do this job? It's time for N1QL!

N1QL is the SQL-like language included with Couchbase for data access and manipulation. In this example, we will use two buckets: HR, which maps to the original Oracle HR schema, and HR_DNORM, which will hold our target document model.

We have already loaded our HR schema. The next step is to create a bucket named HR_DNORM. Then, we will create a primary index in this new bucket:

```
CREATE PRIMARY INDEX ON HR_DNORM
```

Now it is time to create the location documents. These documents are composed of original locations, country, and region documents:

```
INSERT INTO HR_DNORM (key _k, value _v)
SELECT meta().id _k,
{
  "type":"location",
  "city":loc.city,
  "postal_code":loc.postal_code,
  "state_province":IFNULL(loc.state_province, null),
  "street_address":loc.street_address,
  "country_name":ct.country_name,
  "region_name":rg.region_name
} as _v
FROM HR loc
JOIN HR ct ON KEYS "countries::" || loc.country_id
JOIN HR rg ON KEYS "regions::" || TO_STRING(ct.region_id)
WHERE loc.type="locations"
```

A few things to note:

- We are using the projection of a SELECT statement to make the insert. In this example, the original data comes from a different bucket.
- JOINs are used in the original bucket to reference countries and regions
- The IFNULL function used to set explicitly null values for the field state_province.
- The TO_STRING function applied on a number field to reference a key.

Our original sample becomes this:

```
{
 "city": "Tokyo",
 "country_name": "Japan",
 "postal_code": "1689",
 "region_name": "Asia",
 "state_province": "Tokyo Prefecture",
 "street_address": "2017 Shinjuku-ku",
 "type": "location"
}
```

Note that we got rid of our references location_id and country_id. Now, it is time for our employee documents. We will do it in several steps. The first one is to create the employees from the original HR bucket, including department and actual job information:

```
INSERT INTO HR_DNORM (key _k, value _v)
SELECT meta().id _k,
{
  "type":"employees",
  "employee_id": emp. employee_id,
  "first_name": emp.first_name,
  "last_name": emp.last_name,
  "phone_number": emp.phone_number,
  "email": emp.email,
  "hire_date": emp.hire_date,
  "salary": emp.salary,
  "commission_pct": IFNULL(emp.commission_pct, null),
  "manager_id": IFNULL(emp.manager_id, null),
  "job_id": emp.job_id,
  "job_title": job.job_title,
  "department" :
  {
     "name" : dpt.department_name,
     "manager_id" : dpt.manager_id,
     "department_id" : dpt.department_id
  }
} as _v
FROM HR emp
JOIN HR job ON KEYS "jobs::" || emp.job_id
JOIN HR dpt ON KEYS "departments::" || TO_STRING(emp.department_id)
WHERE emp.type="employees" RETURNING META().id;
```

Second, we will use a temporary construction to build the job history array:

```
  INSERT INTO HR_DNORM (key _k, value job_history)
  SELECT "job_history::" || TO_STRING(jobh.employee_id) AS _k,
  {
  "jobs" : ARRAY_AGG(
    {
       "start_date": jobh.start_date,
       "end_date": jobh.end_date,
       "job_id": jobh.job_id,
       "department_id": jobh.department_id
    }
  )
  } AS job_history
  FROM HR jobh
```

```
WHERE jobh.type="job_history"
GROUP BY jobh.employee_id
RETURNING META().id;
```

Now is easy to update our "employees" documents by adding a job_history array:

```
UPDATE HR_DNORM emp
SET job_history=(
  SELECT RAW jobs
  FROM HR_DNORM jobh
  USE KEYS "job_history::" || SUBSTR(meta(emp).id, 11)
)[0]
WHERE
emp.type="employees"
RETURNING meta().id
```

This is how our employee document looks:

```
{
 "commission_pct": null,
 "department": {
   "department_id": 10,
   "manager_id": 200,
   "name": "Administration"
 },
 "email": "JWHALEN",
 "employee_id": 200,
 "first_name": "Jennifer",
 "hire_date": "2003-09-16T22:00:00Z",
 "job_history": [
   {
     "department_id": 80,
     "end_date": "2007-12-31T23:00:00Z",
     "job_id": "SA_REP",
     "start_date": "2006-12-31T23:00:00Z"
   },
   {
        "department_id": 90,
        "end_date": "2001-06-16T22:00:00Z",
        "job_id": "AD_ASST",
        "start_date": "1995-09-16T22:00:00Z"
      },
      {
        "department_id": 90,
        "end_date": "2006-12-30T23:00:00Z",
        "job_id": "AC_ACCOUNT",
        "start_date": "2002-06-30T22:00:00Z"
      }
    ],
    "job_id": "AD_ASST",
    "job_title": "Administration Assistant",
    "last_name": "Whalen",
    "manager_id": 101,
    "phone_number": "515.123.4444",
    "salary": 4400,
    "type": "employees"
  }
```

Note the job_history array of the previous positions.
We can now delete the temporary job_history documents:

```
DELETE FROM HR_DNORM emp
WHERE meta().id LIKE "job_history::%"
```

For our last step, we insert the original *jobs* documents:

```
INSERT INTO HR_DNORM (key _k, value _v)
SELECT meta().id _k, _v
FROM HR _v
WHERE _v.type="jobs"
```

We are done. This is a simple example, but it shows can powerful N1QL can be for data manipulation. Happy data migration!

Migrate From Any Relational Engine to Couchbase Using N1QL and SQSL

Author: Marco Greco

Yes, you have read right: I have not misspelled SQL, there is indeed an extra middle "S"!

A quarter of a century ago, when I was in charge of designing and developing the ERP system at my first employer, people would often come to me and ask to implement a simple report which would consist at most of a couple of SELECT statements, minimal glue code, and some formatting.

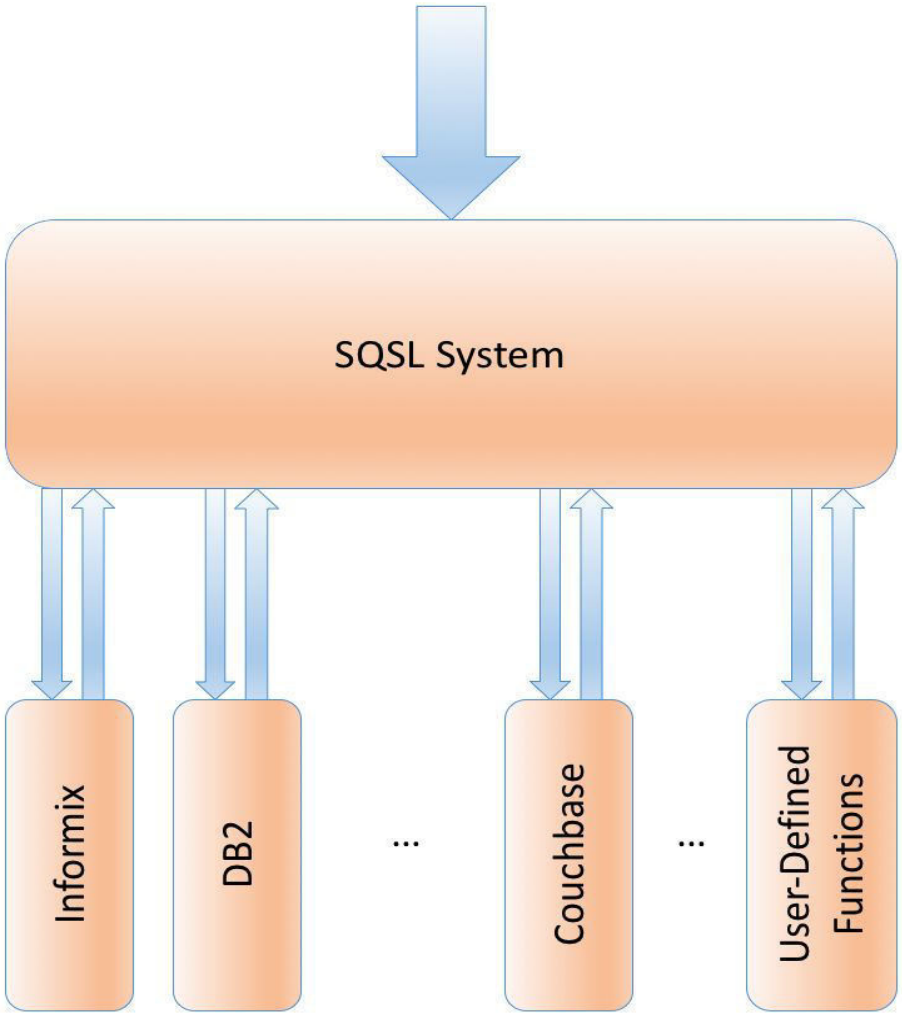
Simple task, you would think, except that the publishing cycle for the application, considering the powerful technological means at our disposal (our development machine at the time was an NCR Tower32 200 with a 5 ¼" 100MB hard disk, 4MB of RAM and AT&T SYSV R2 shipped on a DC600 cartridge[1]), would be at the very least 24 hours, if not 48.

Wouldn't it be nice, I said to myself (yes, being a geek can be a very lonely affair!), if I could just store the SQL and formatting somewhere in the database, and have a scripting language runner do the rest of the work for me, or, rather, my users?

A few late nights later a prototype implementation had been deployed, with minimal control syntax, an expansion facility, and formatting capabilities. The Structured Query Scripting Language was born.
People have a nasty habit of changing employers, and I was no different, so move forward a few years — I had switched country and industry.

SQSL, still without a name and part of a larger application framework, had very much fallen to the bottom of my to-do list.
Still, all was not lost: When my new employer was acquired by a larger corporation, in a spontaneous fight for survival, the language gained some new major features like database engine vendor independence (each database engine would get its own driver in the form of a shared library), concurrent engine access, data redirection, and User Defined Functions.

A few years later still, the language was finally separated from the application framework it had begun life conjoined to, found a windows port (in fact, two) and finally gained a name, albeit not a very imaginative one. Now SQSL is an ETL scripting environment that supports numerous databases, user-defined functions, and a rich set of control structures:



But it was in late 2015 that, having joined Couchbase, it occurred to me that the language could be a solid framework for ETL operations between SQL and NoSQL: N1QL fitted nicely with SQSL's syntax and all this latter needed was a driver for Couchbase Server and a few JSON UTFs.

This article shows how to put the two to good use.
1Some of my first employer's old machines have rightfully found their place at a local history of computer science museum:
https://www.flickr.com/photos/31231773@N02/albums/72157624950108347 and
https://www.flickr.com/photos/31231773@N02/sets/72157624949903613/

Language Overview

SQSL is a client-side SQL-like scripting language geared to developing sophisticated scripts running in a single place, or, rephrased, avoiding the ugly mess of mixing and matching various database shells and separate scripting languages in order to accomplish otherwise straightforward tasks.
It offers a wide range of features, sometimes borrowed from other languages, among which you find

- database independence
- control and simple statements
- simple control constructs for parallel operation (Perl, Bourne, and Korn shells)
- statement redirection
- aggregates, data-driven operation (awk)
- formatting (x4GL, Informix SQL, Oracle UFI / *SQL Plus...)
- expansion facility (Bourne and Korn shells)

The aim is simple — if you know SQL and server-side scripting languages on the one hand, and traditional scripting languages on the other, then an SQSL script should be immediately obvious in meaning to you, even if you had never before come across 'S', 'Q', 'S' and 'L', in that particular order.
By way of example, consider this rather contrived "hello world!" script which selects some data using a database shell and passes it awk to transform and format it.

```
dbaccess <<"EOF" 2>/dev/null | awk '($1=="hello") { $1 = $1 FS "world" } ($1 !=
"") { printf "%s!\n", $1 }'
database stores_demo;
output to pipe 'cat' without headings
 select 'hello' from systables where tabid=1
EOF
```

The end result is

`hello world!`

The script has to first pipe the data to *cat* to remove the projection list labels and then to *awk* in order to remove blank lines, conditionally alter text (append world when it sees hello), and format it (it just appends an exclamation mark). Compare it to this SQSL script, which does everything using one binary and within a single statement:

```
./cli - <<"EOF" 2>/dev/null
database stores_demo;
select "hello" from systables where tabid=1
 aggregate when column(1)="hello" then let column(1) ||= " world"
 format full "%s!"
EOF
```

Or if you prefer a more *awk* like syntax:

```
./cli - <<"EOF" 2>/dev/null
database stores_demo;
select "hello" from systables where tabid=1
 aggregate ($1="hello") $1 ||= " world"
 format full "%s!"
EOF
```

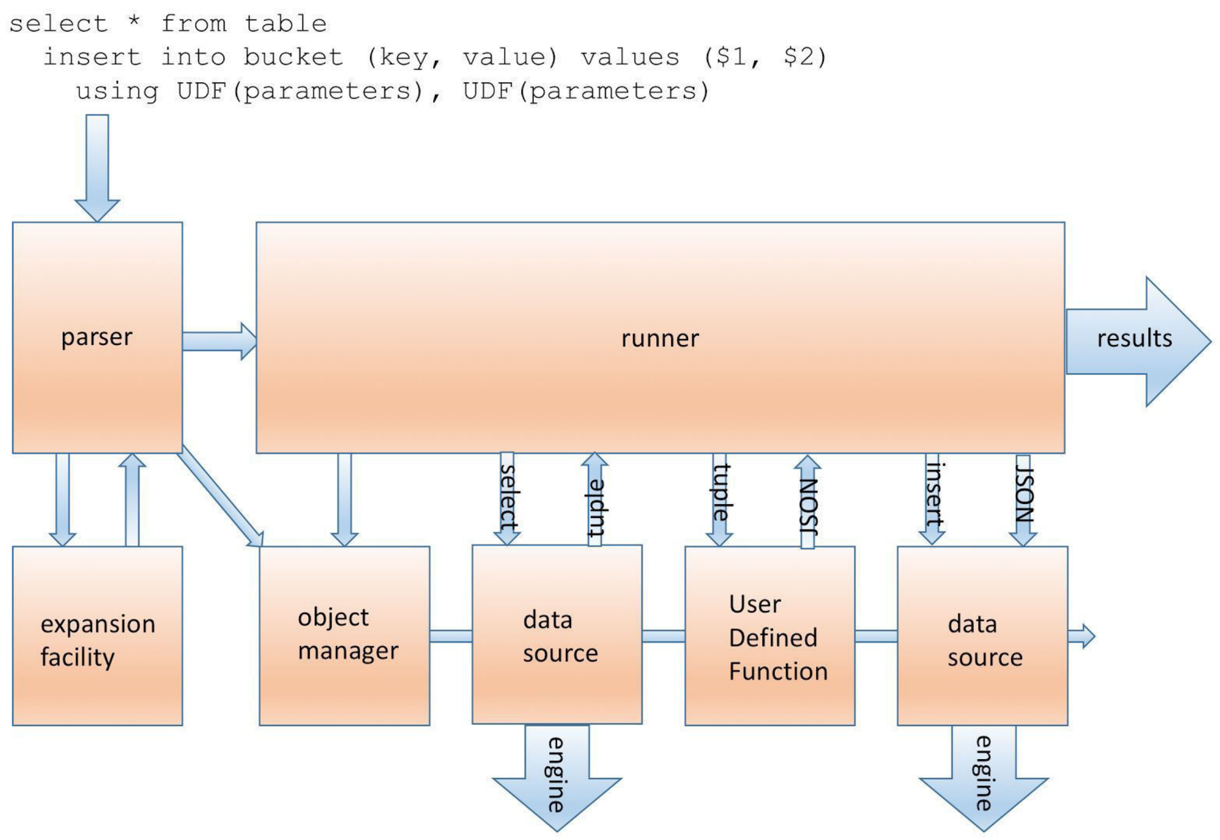
The difference is even more apparent when more than one database engine is involved in the script: say we have to take different actions depending on data coming from two sources: this

```
connect to "stores_demo" source ifmx;
connect to "sample" source db2cli;
```

```
select count(*) from systables connection "stores_demo" into ifx;
select count(*) from sysibm.tables connection "sample" into db2;
if (db2<ifx) then
   display db2 format full "db2 has %i tables";
else
   display ifx format full "informix has %i tables";
end if
```

Looks a damned sight more immediate than two database shells, the bourne shell and awk all chipping in.
I'm sidetracking.
The point of this article is not to dissect the syntax of SQSL, so I will not be providing a detailed feature discussion. If you are interested, you can find the whole syntax reference here.

Instead, we will be covering increasingly elaborate data movement examples, introducing the relevant language features as we encounter them.
For now, the only point I'll make is that SQSL is implemented as a precompiler.
If the engines you are connecting to understand anything that resembles SQL, then the SQSL parser is able to extract the relevant dialects from the actual script, pass the extracted statements to the engines, and make sense of the rest:



```
select * from table
  insert into bucket (key, value) values ($1, $2)
    using UDF(parameters), UDF(parameters)
```

This is precisely what we are doing with N1QL in this particular article.

Migrating Rows to Documents: A Simple Strategy

In general, terms migrating a database from a relational engine to a document store entails the following steps:

- scan the tables system catalog. For each table:
    - determine the primary key columns
    - describe each column (gather name and type)
    - scan the table. For each row
        - assemble the document key from the primary key column values and the table name
        - assemble the output document from each column name, type and value

        - insert into the target bucket using the generated key and document

Let's Get Down to It

Now that we have an idea of the end result, we'll meander through a series of examples, each demonstrating individual steps of the process, starting with…

Migrating a Single Table

```
let fromconn="stores_demo";
connect to fromconn source ifmx;
connect to "couchbase://192.168.1.104:8091" source cb;
select * from customer
 connection fromconn
 insert into default (key, value)
   values($1, $2)
   using json:key("::", columns),
         json:row2doc(displaylabels, columns);
```

That's all there is to it: three statements (well, OK, four, counting the convenience LET).
This example moves the contents of the 'customer' table in Informix's standard stores demo database to Couchbase's default bucket using:

- concurrent connections
- statement redirection
- placeholders and
- UDFs

The first three statements are pretty straightforward: they just connect to an Informix database engine on one side and a Couchbase cluster on the other.
All the action happens in the last statement. The SQSL parser splits the statement into several parts:

- a SELECT statement which it prepares against the connection specified in the connection clause (line 5)

- an INSERT statement (line 6), which, lacking in this example a connection clause, is prepared against the current connection (in this case, our Couchbase cluster)
- data transformation logic contained in the using clause (line 8)

Once the statement is parsed, the binary then proceeds to

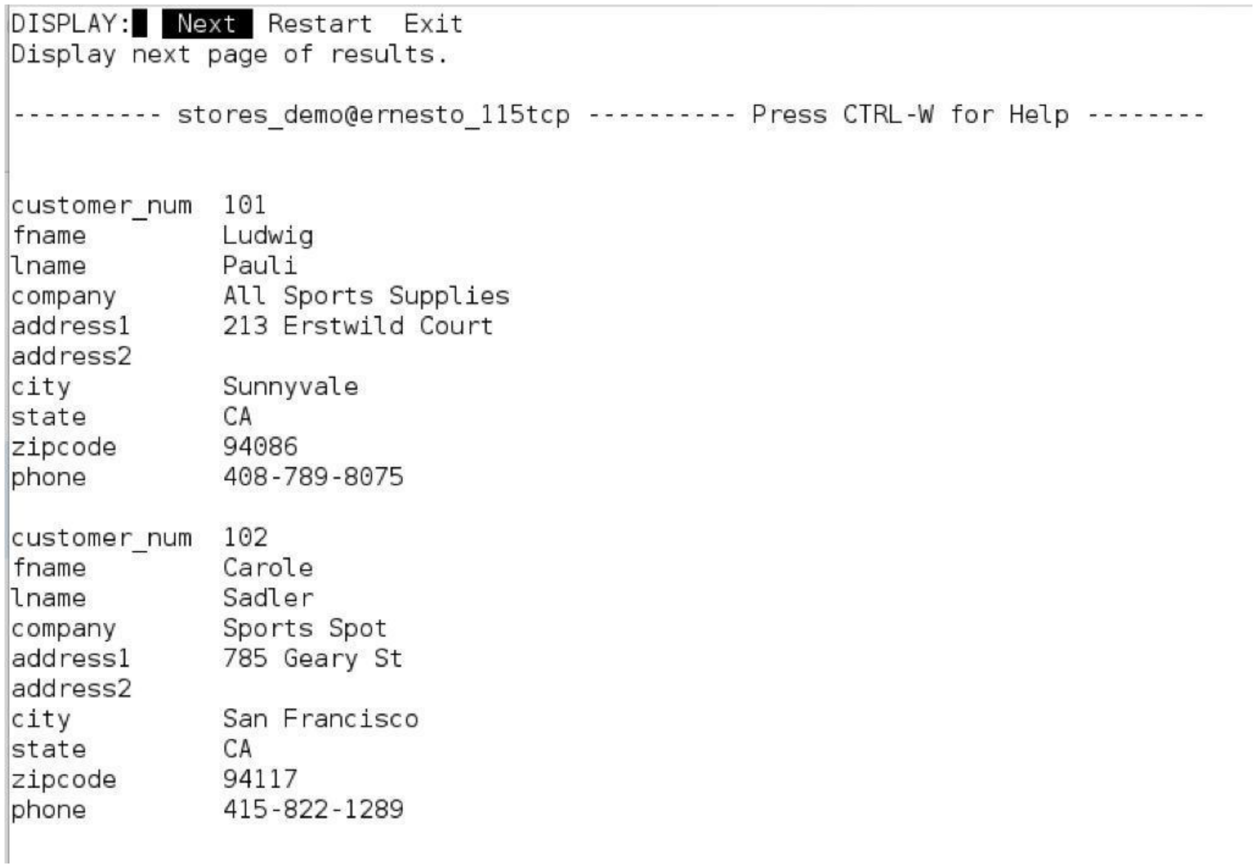
- retrieving rows from one connection using the implicitly prepared SELECT statement
- executing the logic in the using clause (line 8) to assemble a document key and a document from each row examined
- piping the transformed data to the N1QL INSERT statement which is being executed against the second connection

A few points are worth expanding:

- In general terms, in statement redirection, data flows from one side to the other without any need of syntactic sugar, if the output INSERT clause expects the same number of items provided by the input SELECT clause.
- In this example, this is clearly not the case - the SELECT clause is providing a not-yet-described tuple, while the INSERT clause expects a document key and a JSON document, which we have to construct from each incoming tuple. Thus data flow has to be explicitly defined by means of the using clause.
- For statement redirection to work, the input clause must be cursor based and the output clause must have placeholders. SQSL will enforce this.
- The using clause transforms the incoming row values using two User Defined Functions: json:key and json:row2doc (json is just the name of the containing library)
    - key() (line 8) in particular is a variadic function that generates a key using the first argument as a separator, and the other arguments as values.
        - the built-in function columns (line 8) returns a hash (AKA associative array) containing all the values for the current input row
    - row2doc() (line 9) is a variadic function that takes a list of field names and a list of values and constructs a JSON document out of them

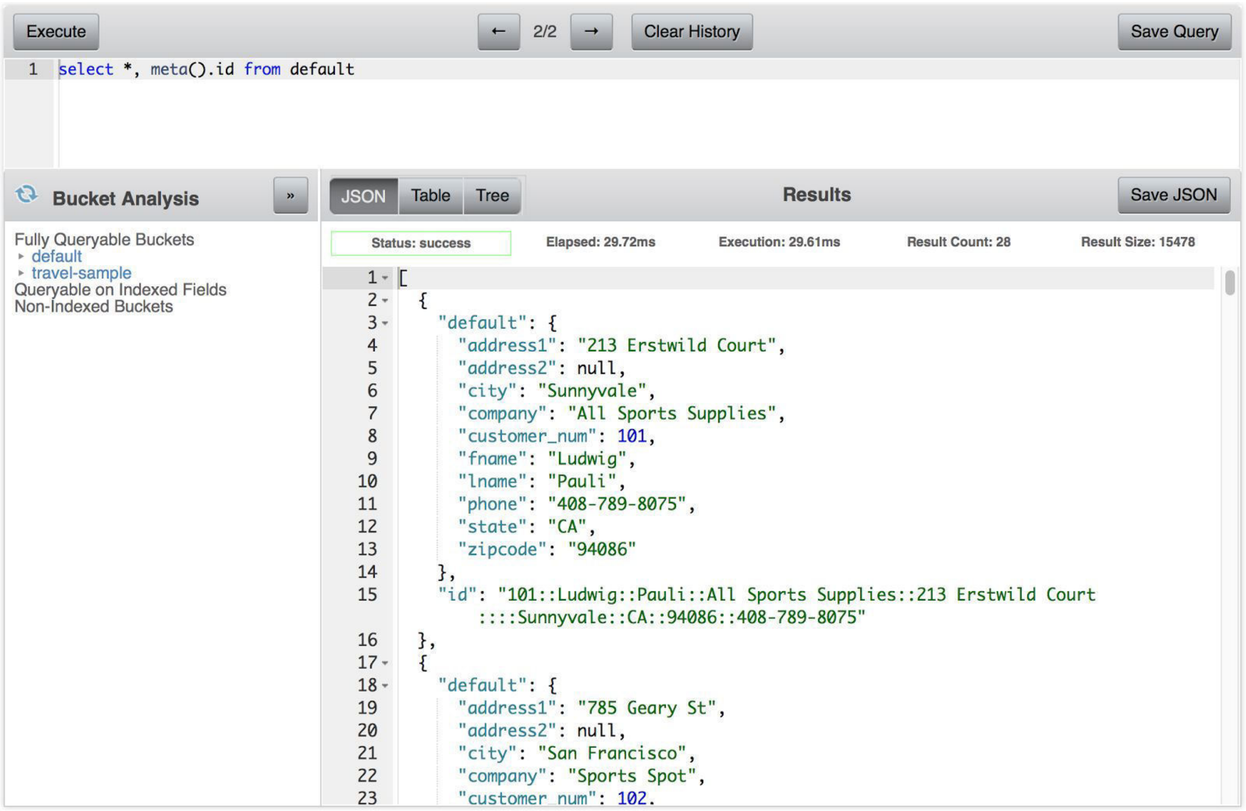
- the built-in function displaylabels (line 9) returns a hash containing all the names of the columns returned by the SELECT's projection list.

And here's the finished result: we migrated from



```
DISPLAY:█  Next  Restart  Exit
Display next page of results.

---------- stores_demo@ernesto_115tcp ---------- Press CTRL-W for Help --------


customer_num  101
fname         Ludwig
lname         Pauli
company       All Sports Supplies
address1      213 Erstwild Court
address2
city          Sunnyvale
state         CA
zipcode       94086
phone         408-789-8075

customer_num  102
fname         Carole
lname         Sadler
company       Sports Spot
address1      785 Geary St
address2
city          San Francisco
state         CA
zipcode       94117
phone         415-822-1289
```

to



```
select *, meta().id from default
```

Status: success   Elapsed: 29.72ms   Execution: 29.61ms   Result Count: 28   Result Size: 15478

```
[
  {
    "default": {
      "address1": "213 Erstwild Court",
      "address2": null,
      "city": "Sunnyvale",
      "company": "All Sports Supplies",
      "customer_num": 101,
      "fname": "Ludwig",
      "lname": "Pauli",
      "phone": "408-789-8075",
      "state": "CA",
      "zipcode": "94086"
    },
    "id": "101::Ludwig::Pauli::All Sports Supplies::213 Erstwild Court
        ::::Sunnyvale::CA::94086::408-789-8075"
  },
  {
    "default": {
      "address1": "785 Geary St",
      "address2": null,
      "city": "San Francisco",
      "company": "Sports Spot",
      "customer_num": 102,
```

And we even did all that without any prior knowledge of the source table schema! We did say that we would be creating document keys from each table's primary key columns, but if you pay close attention to the image above, clearly, we did not.

Bear with me — we'll get there!

Table Move, Take Two

```
let fromconn="stores_demo";
connect to fromconn source ifmx;
connect to "couchbase://192.168.1.104:8091" source cb;
let key=0;
select * from customer
 aggregate key+=1
 connection fromconn
 insert into default (key, value)
   values($1, $2)
   using key::string, json:row2doc(displaylabels, columns);
```

An alternative to using document keys generated from the individual columns values is to use surrogate keys, or, keys that have no relation to the data.
In this case, we are using an incrementing integer value held in the key variable.
The aggregate clause (line 6) can be used to take action on each incoming row, possibly based on some conditions (whether or not based on incoming data values).
In this case, we simply use it to increment the key variable at every incoming row, and then using it as a document key in the using clause (line 10).
Couchbase document keys can only be strings, hence the cast on the *key* variable.

Migrate a Database

```
let fromconn="stores_demo";
connect to fromconn source ifmx;
connect to "couchbase://192.168.1.104:8091" source cb;
foreach select tabname
 from systables
 where tabid>=100 and tabtype="T"
 connection fromconn
 into tab;
   display tab;
   select * from <+get tab+>
     connection fromconn
     insert into default (key, value)
       values($1, $2)
       using json:key("::", tab, columns),
             json:row2doc("tabtype", displaylabels, tab, columns);
end foreach;
```

We can do one better, and move an entire database in one go.
This can be achieved by augmenting the previous example: we just scan the tables system catalog and reuse the same strategy as before for each table found.
Enter the FOREACH loop at line 5: this prepares and executes SELECT statement against the source engine, and for each table name retrieved (stored in the tab

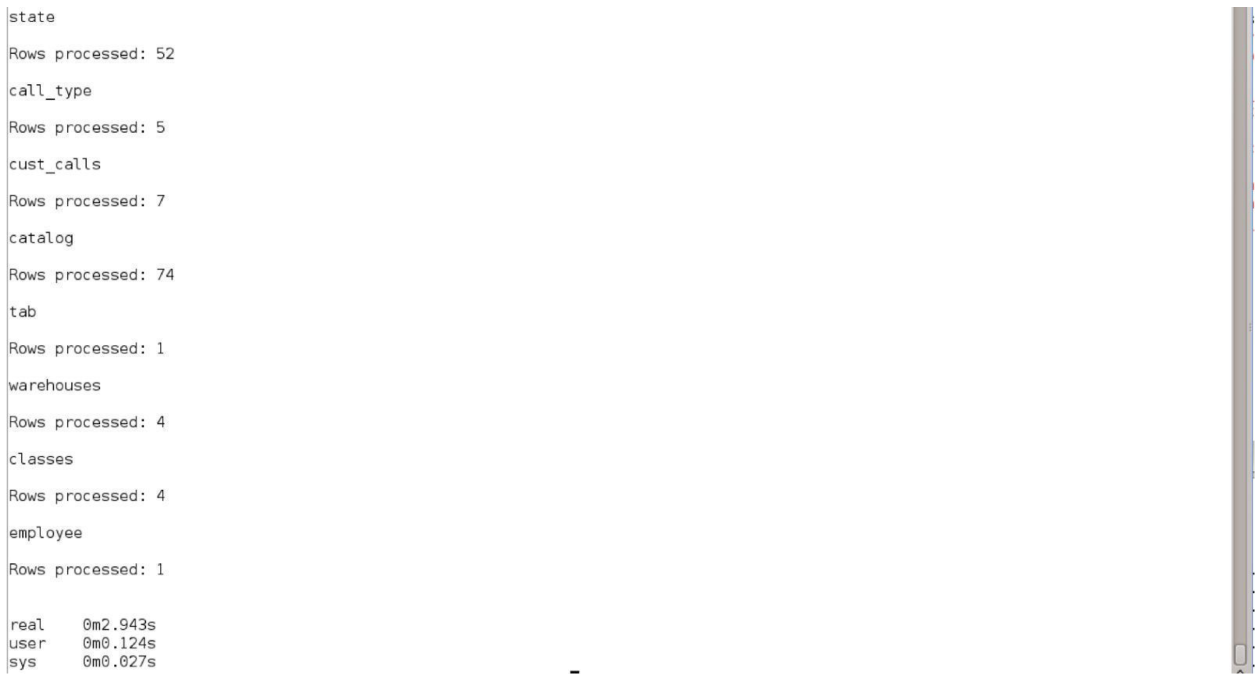
variable via the storage clause at line 9) it proceeds to execute the code in the FOREACH block.

The rest of the code looks pretty much the same as the previous example, with the notable exception of line 11: we need a mechanism to prepare the input SELECT statement using the correct table name, and this we do by means of the expansion facility (the bit in the <+ +> block in line 11).

This, among other things, evaluates the arguments within and injects the result in the statement text — in this case, it places the contents of the tab variable in the text of the prepared statement.

The last thing worthy of note is that now the using clause in line 15 is placing the table name in both the document key and document. This is because the contents of multiple source tables are being placed in the same bucket, and we need to differentiate in between document types, in order to be able to do meaningful queries.

Here's a sample output from the previous script.
At first sight, not very interesting, but I do have a point, I promise.



```
state
Rows processed: 52
call_type
Rows processed: 5
cust_calls
Rows processed: 7
catalog
Rows processed: 74
tab
Rows processed: 1
warehouses
Rows processed: 4
classes
Rows processed: 4
employee
Rows processed: 1

real    0m2.943s
user    0m0.124s
sys     0m0.027s
```

Database Move, Different Strategy

```
let fromconn="stores_demo";
connect to fromconn source ifmx;
connect to "couchbase://192.168.1.104:8091" source cb;
prepare insst from insert into default (key, value)
         values($1, $2)
         using json:key("::", tab, columns),
               json:row2doc("tabtype", displaylabels, tab, columns);
foreach select tabname
 from systables
 where tabid>=100 and tabtype="T"
 connection fromconn
 into tab;
   display tab;
   select * from <+get tab+>
     connection fromconn
     execute insst;
end foreach;
free insst;
```

The statement redirection clause can sport prepared statements, too.
In this example, we PREPARE an INSERT statement beforehand (line 5) and just EXECUTE it (line 18) as a target for the SELECT clause in line 16.
Clearly, at the time the INSERT is prepared, the tab variable is undefined and there even isn't a source statement to which the displaylabels and columns built-in functions (in lines 7 and 8, in the USING clause) could apply: as such you would rightfully expect the PREPARE statement to fail.

Lazy evaluation to the rescue!

The insst prepared statement will actually fail if executed on its own, but execute it as part of a redirected statement, and everything suddenly makes semantic sense: the using clause will use the current tab value and the values current for each individual row.

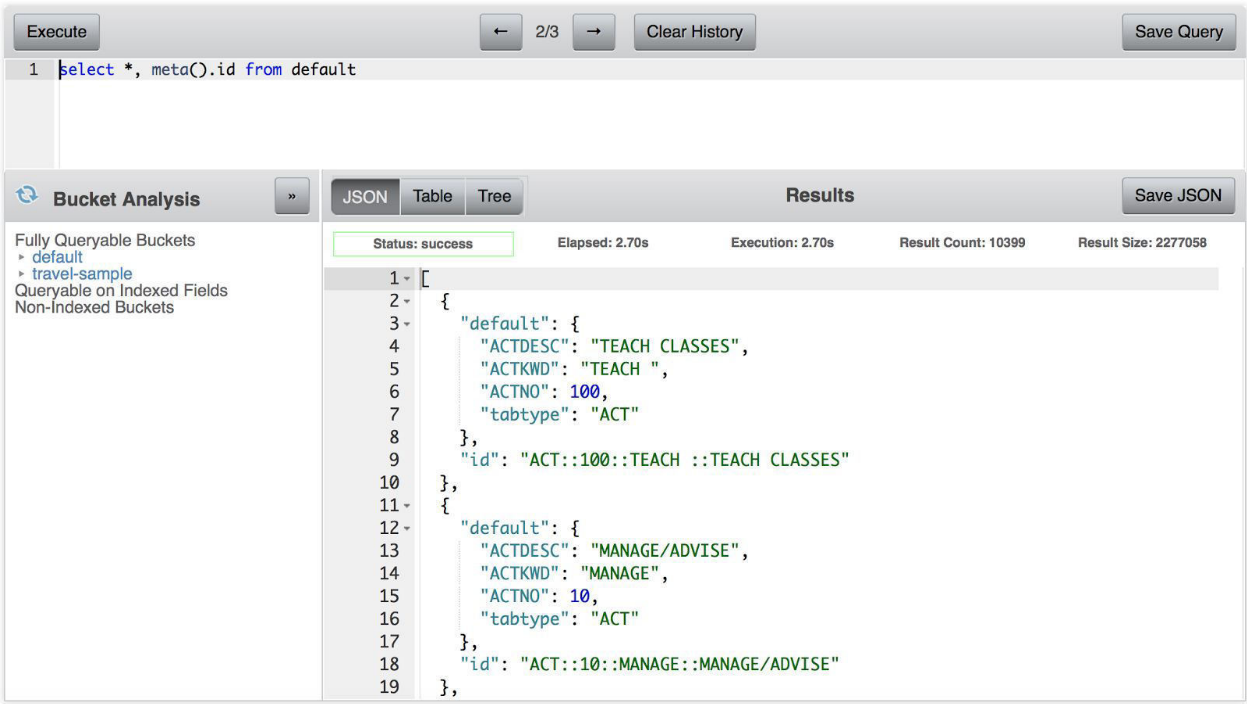
Explaining why using a prepared statement for the SELECT part of the redirected statement, although feasible, offers no advantage over the code above is left as an exercise for the reader.

Database Move, Different Source

```
let fromconn="sample";
let schema="DB2INST1";
connect to fromconn source db2cli;
connect to "couchbase://192.168.1.104:8091" source cb;
foreach select name
 from sysibm.systables
 where creator=? and type='T'
 using schema
 connection fromconn
 into tab;
   display tab;
   select * from <+get schema+>.<+get tab+>
     connection fromconn
     insert into default (key, value)
       values($1, $2)
       using json:key("::", tab, columns),
             json:row2doc("tabtype", displaylabels, tab, columns);
end foreach;
```

The previous example moved the whole Informix's stores demo database to Couchbase.

This one does the same with DB2's sample demo database.
Note that the process is exactly the same as before - aside from connecting to a different engine, the changes needed are a different SELECT statement (specific to the source database engine) for the FOREACH uses to find table names and, in this case, the use of schema names when selecting from source tables.
Note how DB2 system catalogs store column names in uppercase:



```
select *, meta().id from default
```

Status: success   Elapsed: 2.70s   Execution: 2.70s   Result Count: 10399   Result Size: 2277058

```
[
  {
    "default": {
      "ACTDESC": "TEACH CLASSES",
      "ACTKWD": "TEACH ",
      "ACTNO": 100,
      "tabtype": "ACT"
    },
    "id": "ACT::100::TEACH ::TEACH CLASSES"
  },
  {
    "default": {
      "ACTDESC": "MANAGE/ADVISE",
      "ACTKWD": "MANAGE",
      "ACTNO": 10,
      "tabtype": "ACT"
    },
    "id": "ACT::10::MANAGE::MANAGE/ADVISE"
  },
```

Parallel Migration

```
let fromconn="stores_demo";
connect to fromconn source ifmx;
let tabcount=0;
select tabname
 from systables
 where tabid>=100 and tabtype="T"
 aggregate tabcount+=1
 into tabs(tabcount);
clone tabcount into child, children(child);
   connect to fromconn source ifmx;
   connect to "couchbase://192.168.1.104:8091" source cb;
   select * from <+get tabs(child)+>
     connection fromconn
     insert into default (key, value)
       values($1, $2)
       using json:key("::", tabs(child), columns),
             json:row2doc("tabtype", displaylabels, tabs(child), columns);
 parent;
   display tabs(child) format full "moving %s";
   let pid2tab(children(child))=tabs(child);
end clone;
wait for children.* into p, r;
```

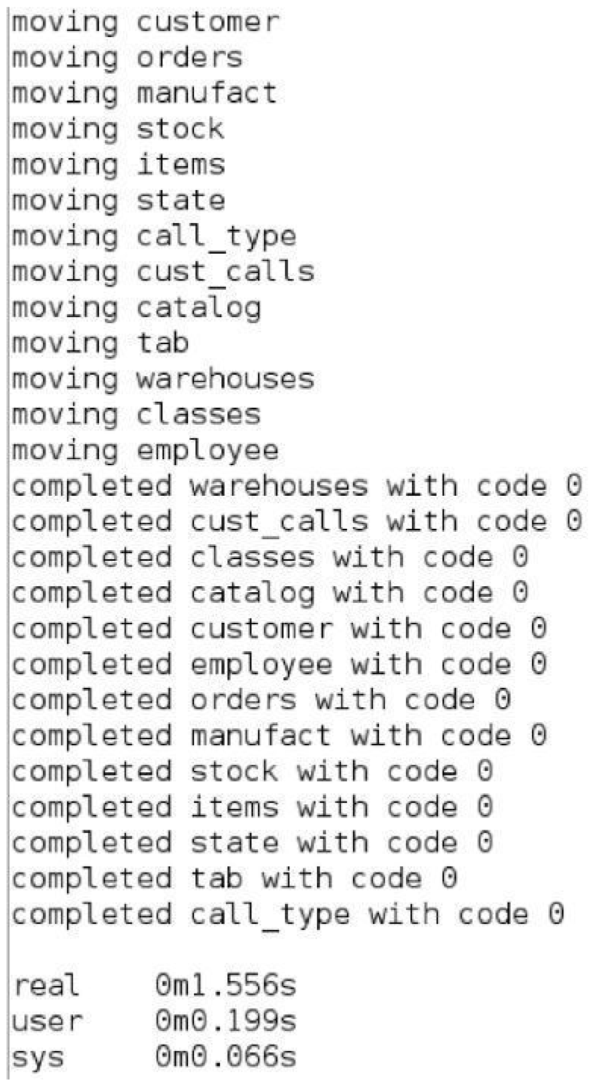
```
        display pid2tab(p), r format full "completed %s with code %d";
    end wait;
```

We have seen how to migrate whole databases from different engines without knowing the database schema, but we can do better still: we can migrate databases in parallel. The strategy used is simple — rather than looping through the tables handling each in turn, we'll fork a child for each table involved in the migration. This could be achieved using a FOREACH loop, as in the earlier examples, and using the fork function to start each child, but we use a more elegant method instead: a CLONE loop. This essentially forks the required number of children, each executing the code inside the block, or up to the PARENT clause, if present. The PARENT clause can be used by the forking process to take action after each child has been forked. In this case (line 20), all it does is just print the table being moved by each child.

After all children have been forked, the parent process just waits for them to complete by means of the WAIT loop at line 23.
Each child behaves as in each of the previous examples.
The only thing left to be explained is how each child determines which table to move: simples!

Note how we had earlier quickly built a list of tables in the tabs hash, using the SELECT statement at line 5 (a neat trick here was to move the target variable in the storage clause at line 9 by means of the aggregate clause — line 8 — thereby skipping a less efficient FOREACH loop). The tabs hash is indexed via an integer key. Coincidentally, each child is identified by an integer key stored in the child variable - so all each child has to do is to pick the table matching its child id, via the expansion at line 14.

As for the earlier examples, here's a sample output:



```
moving customer
moving orders
moving manufact
moving stock
moving items
moving state
moving call_type
moving cust_calls
moving catalog
moving tab
moving warehouses
moving classes
moving employee
completed warehouses with code 0
completed cust_calls with code 0
completed classes with code 0
completed catalog with code 0
completed customer with code 0
completed employee with code 0
completed orders with code 0
completed manufact with code 0
completed stock with code 0
completed items with code 0
completed state with code 0
completed tab with code 0
completed call_type with code 0

real    0m1.556s
user    0m0.199s
sys     0m0.066s
```

Remember the output from the earlier database migration example?
Here's the point that I was trying to make — I do realize we didn't really have a lot to migrate, but even with the limited amount of work involved, parallel operations do save time!

...And for My Last Trick: Parallel Migration, Indexes and All!

```
let fromconn="stores_demo";
connect to fromconn source ifmx;
{ gather table list }
let tabcount=0;
select tabname, tabid
 from systables
 where tabid>=100 and tabtype="T"
 aggregate tabcount+=1
 into tabs(tabcount), tabids(tabcount);
{ fork children }
clone tabcount into child, children(child);
   connect to fromconn source ifmx;
   { gather table primary key column numbers }
   let primcount=0;
   select ikeyextractcolno(indexkeys, idx)
     from sysindices, sysconstraints, table (list{0, 1, 2, 3, 4, 5, 6, 7})
colnos(idx)
     where sysconstraints.constrtype="P"
       and sysconstraints.idxname=sysindices.idxname
       and sysconstraints.tabid=?
       and ikeyextractcolno(indexkeys, idx)>0
       using tabids(child)
       aggregate primcount+=1
       into primkeys(primcount);
   { do the data migration }
   connect to "couchbase://192.168.1.104:8091" source cb;
   select * from <+get tabs(child)+> connection fromconn
     insert into default (key, value)
   values($1, $2)
   using json:key("::", tabs(child),
         { only use primary key columns }
         columns<+ get case when primcount>0
                            then ".(
                                 <+ separator "," get primkeys.(1 to primcount)
+>
                                 )"
                            else ""
                       end case +>),
         json:row2doc("tabtype", displaylabels, tabs(child), columns);
 parent;
   display tabs(child) format full "moving %s";
   let pid2tab(children(child))=tabs(child);
end clone;
{ wait for children }
wait for children.* into p, r;
   display pid2tab(p), r format full "completed %s with code %d";
end wait;
{ create indexes }
connect to "couchbase://192.168.1.104:8091" source cb as "cb";
```

```
set connection fromconn;
{ loop through tables }
for tabnum in 1 to tabcount do
   {loop through indexes, getting name and columns }
   foreach select idxname, part1, part2, part3, part4, part5, part6, part7,
part8,
                 part9, part10, part11, part12, part13, part14, part15, part16
     from sysindexes
     where tabid=?
     using tabids(tabnum)
     { make sure that the index name has no spaces }
     aggregate $1=replace($1, " ", "_")
     into idxname, ikeys.(1 to 16);
       display idxname, tabs(tabnum) format full "creating %s on %s";
       create index `<+get idxname+>`
         on default(<+silent multi quotes "``" separator ","
                <* get column names from catalog using column numbers *>
                select colname from syscolumns
                  where tabid=<+get tabids(tabnum)+>
                    and colno in (<+multi separator "," get ikeys.(1 to 16)+>)
                  order by field(colno, list{<+multi separator "," get ikeys.(1
to 16)+>})+>)
         where tabtype="<+get tabs(tabnum)+>"
         connection "cb";
   end foreach;
end for
```

The examples that we have considered so far have a major issue - you can't join different types of documents because the keys contained in individual documents do not match the document keys they should be referring to (remember we were creating document keys out of the contents of the entire document), and even if they did, we hadn't yet created the supporting indexes.
This is where we put things right. The first thing we have to do is determining which columns make up the primary key for each index, so that we generate the correct document keys. This is done by the weird looking SELECT at line 18 - which populates the primkeys hash with the list of column numbers of the primary key.
Having the column list, we have to extract from the column hash the relevant values.

For this, we use a combination of hash expansion and multi-value expansions.
A quick shorthand to generate a partial list of elements of a hash is with the following dot notation

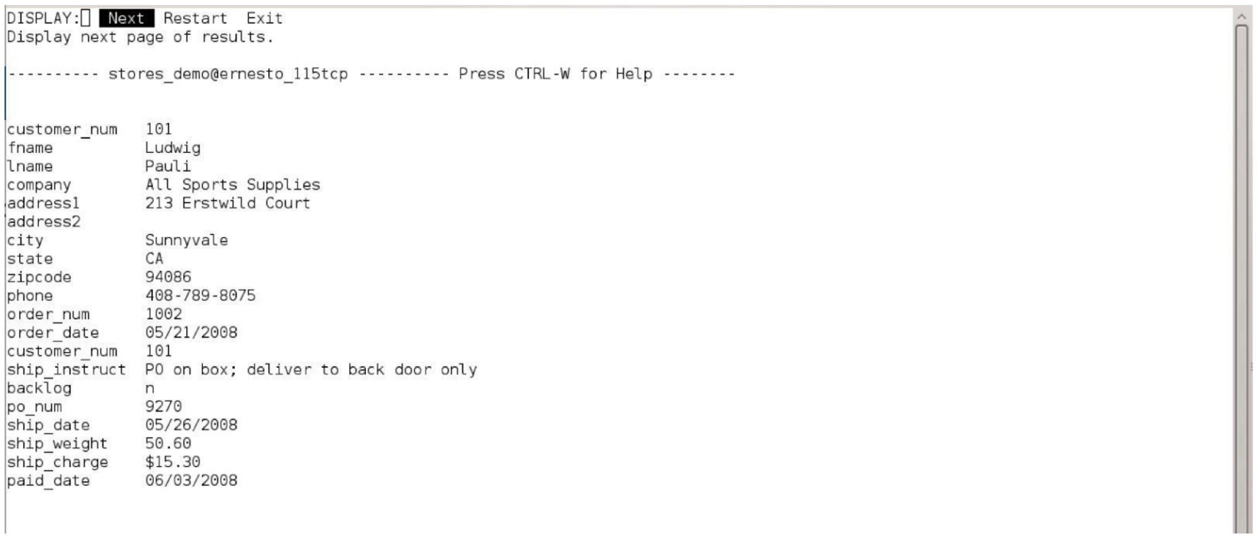
`identifier.(expression list)`

This will generate an expression list with just the hash elements whose keys are in the list following the identifier.
For instance, we can get the values of the first 16 columns with

`columns.(1 to 16)`

So the trick here is to expand the prinkeys hash to a list of its elements, and then applying that to the columns hash.
The expansion at line 38 does exactly that — generate a comma separated list of values, which is then applied to the columns hash by the expansion at line 36, if the table in question has a primary key.
We now move to index creation.
I could create the indexes relevant to each table in each child, however, this is not a strategy that pays if, like me, you are working on a single node - the sole indexing service would not be very happy being swamped with index creation requests.
For this reason, in this example, we wait for the children to finish loading and then loop through the tables (line 58), and for each table, we loop through the indexes (line 61).
In line 68 we use an aggregate clause to make sure that the index names we fetch don't have spaces. The indexing services doesn't like them.
After that, we just assemble a N1QL CREATE INDEX statement (line 71) with similar techniques to what we have already seen.
An interesting point to note is that we get index field at line 72 names through an expansion directly SELECTing a list of column names.
After the migration, we can easily port an original relational query

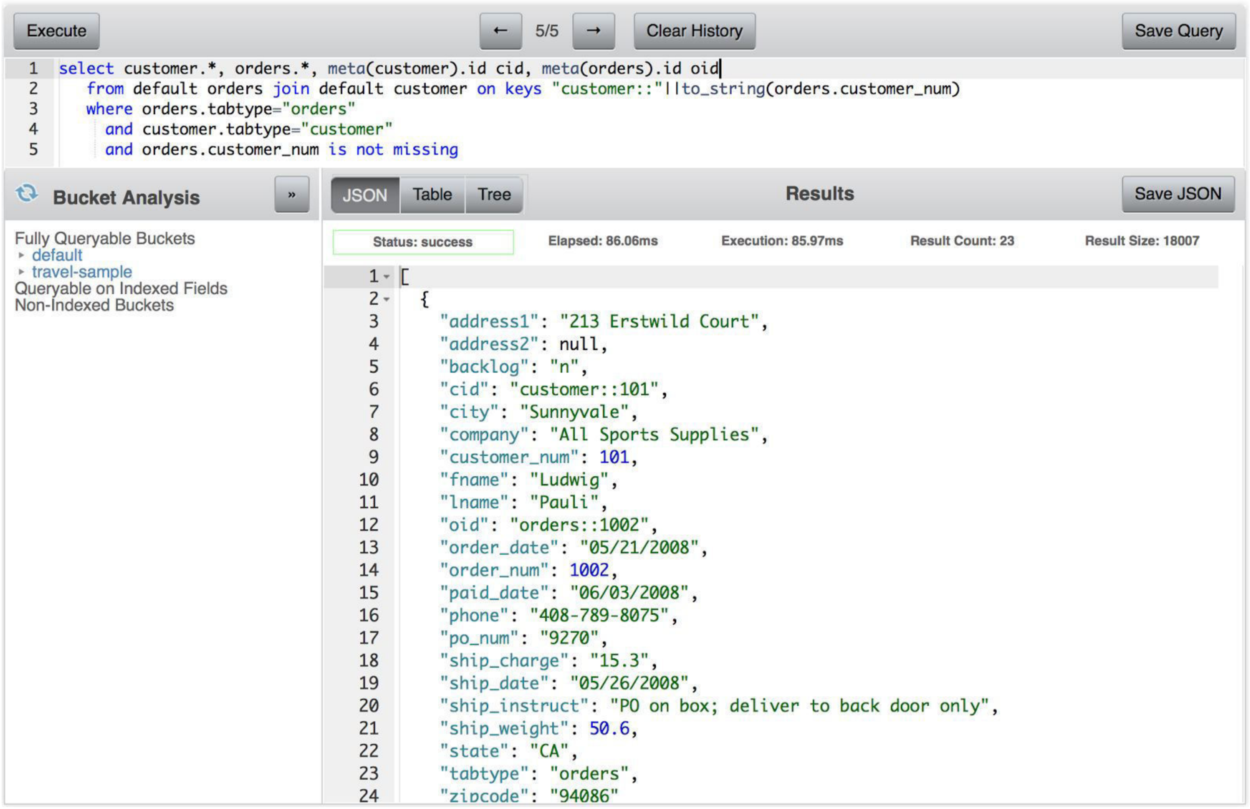
```
select customer.*, orders.*
 from customer, orders
 where customer.customer_num=orders.customer_num
```



```
DISPLAY:  Next  Restart  Exit
Display next page of results.

---------- stores_demo@ernesto_115tcp ---------- Press CTRL-W for Help --------


customer_num   101
fname          Ludwig
lname          Pauli
company        All Sports Supplies
address1       213 Erstwild Court
address2
city           Sunnyvale
state          CA
zipcode        94086
phone          408-789-8075
order_num      1002
order_date     05/21/2008
customer_num   101
ship_instruct  PO on box; deliver to back door only
backlog        n
po_num         9270
ship_date      05/26/2008
ship_weight    50.60
ship_charge    $15.30
paid_date      06/03/2008
```

to N1QL:

```
select customer.*, orders.*, meta(customer).id cid, meta(orders).id oid
  from default orders
    join default customer
      on keys "customer::"||to_string(orders.customer_num)
  where orders.tabtype="orders"
    and customer.tabtype="customer"
    and orders.customer_num is not missing
```

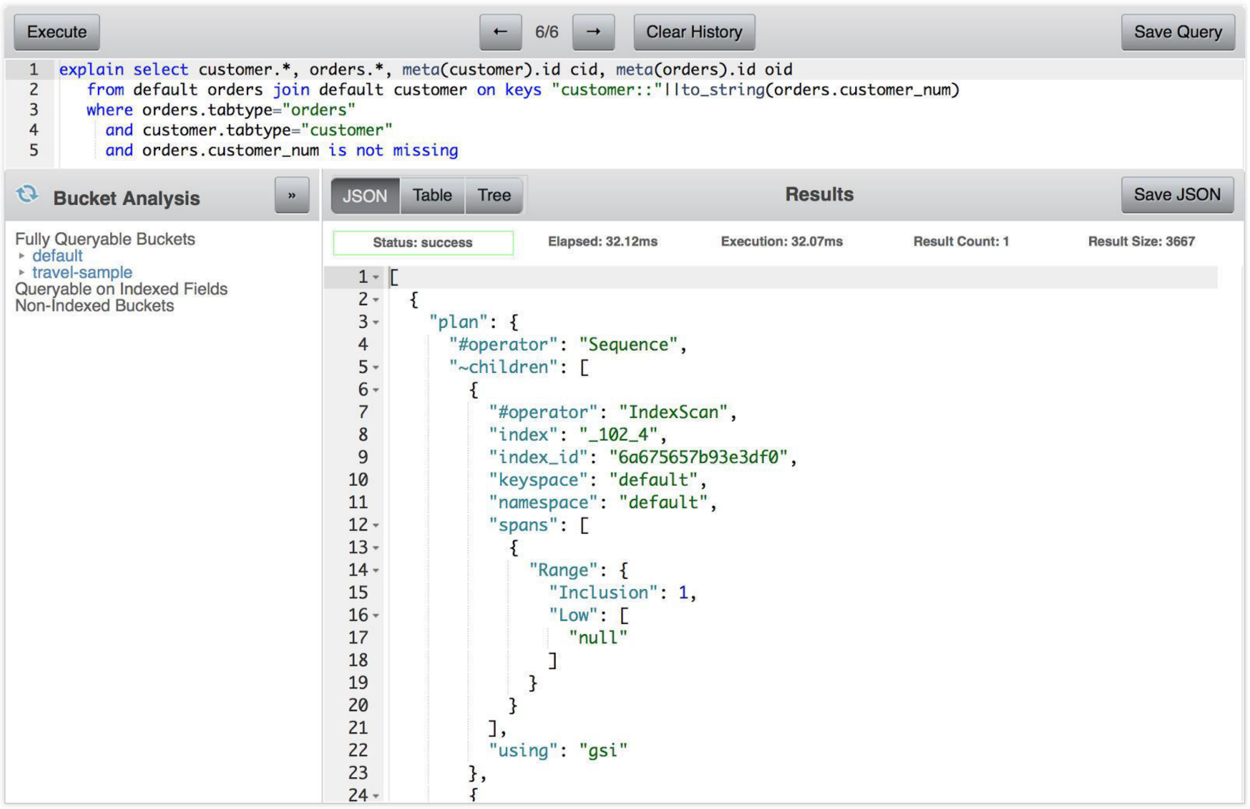


```
select customer.*, orders.*, meta(customer).id cid, meta(orders).id oid
  from default orders join default customer on keys "customer::"||to_string(orders.customer_num)
  where orders.tabtype="orders"
    and customer.tabtype="customer"
    and orders.customer_num is not missing
```

Status: success   Elapsed: 86.06ms   Execution: 85.97ms   Result Count: 23   Result Size: 18007

```
[
  {
    "address1": "213 Erstwild Court",
    "address2": null,
    "backlog": "n",
    "cid": "customer::101",
    "city": "Sunnyvale",
    "company": "All Sports Supplies",
    "customer_num": 101,
    "fname": "Ludwig",
    "lname": "Pauli",
    "oid": "orders::1002",
    "order_date": "05/21/2008",
    "order_num": 1002,
    "paid_date": "06/03/2008",
    "phone": "408-789-8075",
    "po_num": "9270",
    "ship_charge": "15.3",
    "ship_date": "05/26/2008",
    "ship_instruct": "PO on box; deliver to back door only",
    "ship_weight": 50.6,
    "state": "CA",
    "tabtype": "orders",
    "zipcode": "94086"
```

Which, just to prove the point, uses the correct index:



```
explain select customer.*, orders.*, meta(customer).id cid, meta(orders).id oid
  from default orders join default customer on keys "customer::"||to_string(orders.customer_num)
  where orders.tabtype="orders"
    and customer.tabtype="customer"
    and orders.customer_num is not missing
```

Status: success   Elapsed: 32.12ms   Execution: 32.07ms   Result Count: 1   Result Size: 3667

```
[
  {
    "plan": {
      "#operator": "Sequence",
      "~children": [
        {
          "#operator": "IndexScan",
          "index": "_102_4",
          "index_id": "6a675657b93e3df0",
          "keyspace": "default",
          "namespace": "default",
          "spans": [
            {
              "Range": {
                "Inclusion": 1,
                "Low": [
                  "null"
                ]
              }
            }
          ],
          "using": "gsi"
        },
        {
```

Conclusion

We have quickly gone through several features of a very old project of mine, geared to migrating between two relational databases, and we have used it to migrate databases from the relational world to Couchbase server, using N1QL. This we have done without having any knowledge of the schema of the source database.

Having covered Extract, enough Transform to create documents out of tuples, and Load, the next logical step is to explore more Transform, in order to move the data that we have just loaded to a denormalized model.

For this, could I point you to Manuel Hurtado's excellent ELT with Couchbase and N1QL article.

Author's note

You can find the complete stable source of the SQSL project at
http://www.sqsl.org/sqslsource.htm.
The current development tree can be found at
http://github.com/marcogrecopriolo/sqsl
SQSL comes in the form of libraries, since the original project aimed to embed the runner in larger applications.
The source shows how to put the libraries to good use with several sample applications, in the form, not very imaginatively, of command line shells: you can choose from command line interface to curses, to NW.js to NPAPI.
The examples above have been run using the CLI shell.
The runner connects to individual database engines using ad hoc data sources, in the form of shared libraries.
The number of available data sources is currently fairly limited.
The current implementation relies on Informix ESQL/c for decimal, date, datetime, and interval arithmetics. I have to write my own library.
The Couchbase data source and the JSON UTFs are alpha at best. The Couchbase data source does not yet parse incoming JSON documents.
There isn't an OSX port yet — that's just about the only port missing.

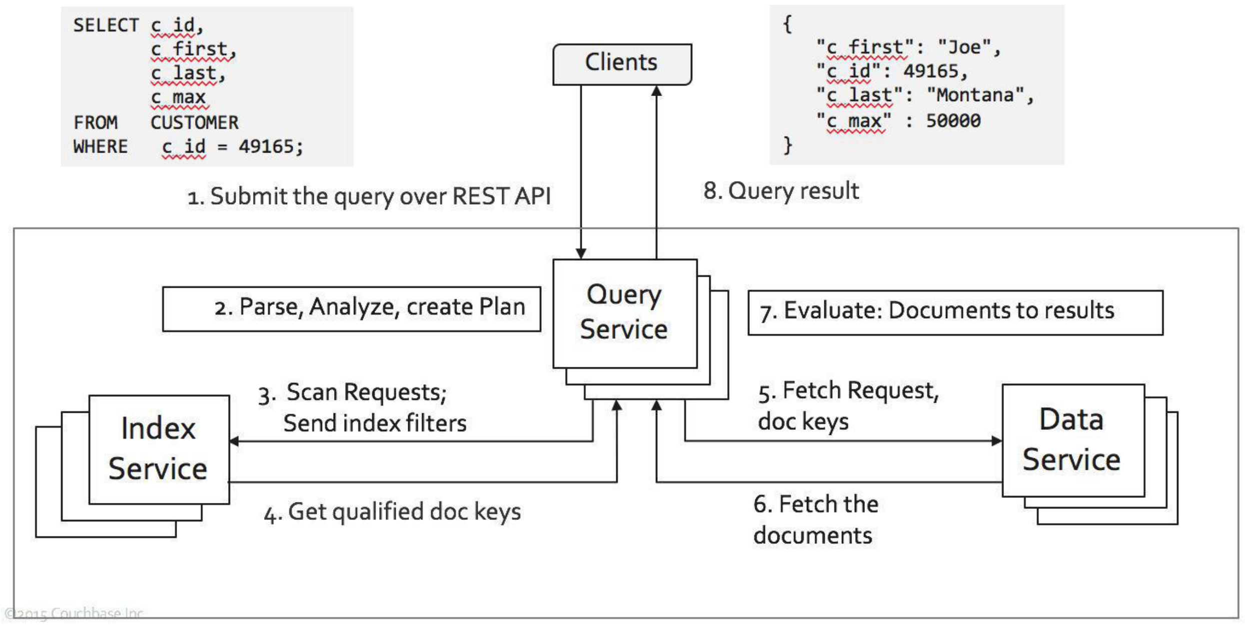
How Couchbase Won YCSB

Author: Keshav Murthy

We chose to run YCSB and do the other performance work in Couchbase 4.5. Not because it's easy, but because it's hard; because that goal served us to organize and measure our effort and features and hold them to a high standard.

Like all the good benchmarks, YCSB is simple to support and run. Most NoSQL can run YCSB benchmarks. The difficulty is in getting those numbers high. Just like the TPC wars in 90s, the NoSQL market is going through its YCSB wars to prove the performance and scalability.

Cihan Biyikoglu has discussed the combination of technologies within Couchbase helping customers scale and perform. In this article, we'll focus on Workload-E in YCSB — How Couchbase is architected and enhanced in Couchbase 4.5 to win this round of YCSB against MongoDB.



```
SELECT c_id,
       c_first,
       c_last,
       c_max
FROM   CUSTOMER
WHERE   c_id = 49165;
```

```
{
   "c_first": "Joe",
   "c_id": 49165,
   "c_last": "Montana",
   "c_max" : 50000
}
```

Clients
1. Submit the query over REST API
8. Query result
Query Service
2. Parse, Analyze, create Plan
7. Evaluate: Documents to results
3. Scan Requests; Send index filters
Index Service
4. Get qualified doc keys
5. Fetch Request, doc keys
Data Service
6. Fetch the documents
©2015 Couchbase Inc.

Figure 1 Couchbase Architecture.

To put it simply, Couchbase is a distributed database. Each of the critical services — Data, Query, Index — can be scaled up and scaled out. How you scale is your choice.

We used workload-A and workload-E to measure the performance. In this article, we'll focus on workload-E. The workload-E simulates queries for threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread ID). In all, 95% of the operations are scans and 5% of the operations are inserts to the database.
The short scan operations can be written in SQL with the following query:

```
SELECT field0, field1, field2, field3,
     field4, field5, field6, field7,
     Field8, field9
FROM  ycsb
WHERE primary_key > "xyz1028"
ORDER BY primary_key
LIMIT 50;
```

In Couchbase, the YCSB data is represented as the following:

```
DocumentKey:  "user4052466453699787802"
{
"field0":
"(W342\u003e?5f.Q;6S{+;t30h#Vc:_.7\u003e5!~3Q)0?h#\"r77j+H%89:%\u003cb\"Ss:56\"G=94\u003e(Q'!Yi57f+@k;H-%+v4J;.:4U\
u00261 *1\u003c=",
"field1": "8/t:3|0,d\"@i6^+452\u0026Ly3108,|+2v?D%,2$%_5$Uc\u0026=l:Zm)Hk3*02Ak*$|
;\"-8r+2*\u003cA?#N#!84%\u0026~6X656!Ms?+`;/`'Cg6",
"field2":
"+J;5Nc\u003e5h\"/f1+l?.v.\u003e\"-4p!Rq\u003eY53;\u003e4Pg9We*!6;]}7A)8\u0026`\u003eWg+\\1-Nq?Wk:,\u003e$Ae3\u003c
4\u003c7~+\u0026,/ d*Y!5?(=@k\u00269|'F53D!$",
"field3": ",\u003c`'-\u003e;2*1Du#A5\u003cA36/d6%:*!\u003c -*?;4!%
)\\o'\"j2,28\u003ez1Ze;Ve:Ci\"+,*'.4^+,9f+Rk7Vo/Y'5h*7n2L398p\u003c=l'+|3Ru7",
"field4": "36t?Po!5.%Jk6\"l':8=48$P5*H/4F/#'4 %0,T}'9z##j6'
-\\!!2n958\u003c7d0%r*44\u003e(f/[a?#69]w((f.No+U)0M52+\u003c-D/*W30",
"field5": ",Fu+182 40?$?$:)2:))6$Gg
Yk6',\u003e,\"5Cw%9\u0026,'p.^#%7p4Ps/Vy-[;$Ak9!l\u003e%8\u003eI)46l/]g2Go(-\u003e= x;]!\u003c2~\"B!0Bi)\u003c.4",
"field6": "+2h;@m,)0,7n-Iw/W/79b7Y{*% 8\u00268\u003e:r'X/,#08\":7Fm72\u003c/Dw7Ja %l',\"7I+\u003cEe?;4% v%\u0026,$-
;$\u0026,%86 ~3J7\u0026?n*O#2S?8",
"field7": ",/\u003e4 (,L}\u00263h5/j2(,1Q+1A)4$\u003e\u003c+.,;d6@;*Yi;Ou-\u00260%?\"3Ua;\\k8P'$\"d#B7$\"*-S1(8
!Vy*Kg;^e2N);?. Ke1Eg\u003eVo\"R=(",
"field8": "52p\";:\u003eBu\u0026D/0O/ ?0='d3%\"'G+\u003c z.Gg\u003eIy!Xe?D-6$0%Za
1:%(01.6%F/#\u003e\u003c(6\u00265\\)8_i:(\u003c5Oo+?p7+l?Ym6\"|)\u0026\u003c\u003c(\u003e4,,-",
"field9": "#G=($:6X%%Z14,v#6r;#l/9x04*8 h,N3-Za4Da##`')t.\u003e(-(p#C:L%= d(S2H/\"P}/%0/1j/=h0Q1
)2\u003c12(_y#P!$82(\\=!"
}
```

The Couchbase port for YCSB is available at:
https://github.com/brianfrankcooper/YCSB/tree/master/couchbase2
Below is the actual query in Couchbase. The meta().id expression in the query refers to the document key of the document. In Couchbase, each document in a bucket will have a unique key, referred to as document key.

```
SELECT field0, field1, field2, field3,
     field4, field5, field6, field7,
     Field8, field9xz
FROM  ycsb
WHERE meta().id >= "xyz1028"
ORDER BY meta().id
LIMIT 50;
```

Once you load the data, you simply create the following index. You're ready to run the YCSB workload E.

```
CREATE INDEX meta_i1 ON ycsb(meta().id);
```

When we ran this on Couchbase 4.0, we knew we had our work cut out. In this article, we'll walk through the optimizations done in 4.5 to improve this query. What we've done for this query applies to your index design, resource usage, and performance of your application queries.

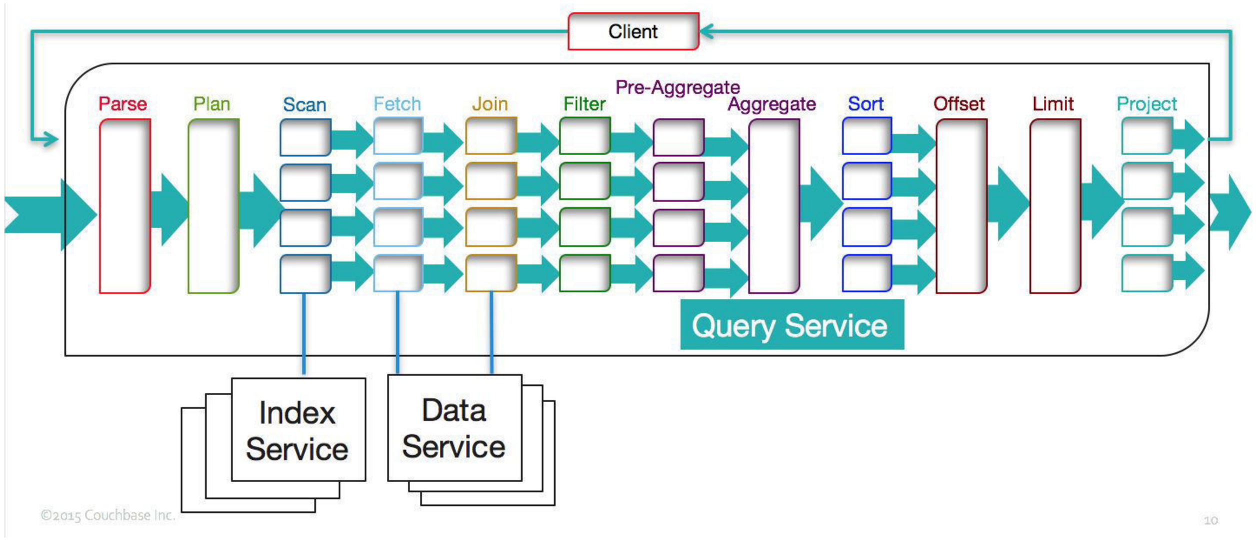
In 4.5, when you execute this query, here is the plan used:

```
[
{
  "plan": {
    "#operator": "Sequence",
    "~children": [
      {
        "#operator": "Sequence",
        "~children": [
          {
            "#operator": "IndexScan",
            "index": "meta_i1",
            "index_id": "bb26497ef8cf5fef",
            "keyspace": "ycsb",
            "limit": "50",
            "namespace": "default",
            "spans": [
              {
                "Range": {
                  "Inclusion": 1,
                  "Low": [
                    "\"xyz1028\""
                  ]
                }
              }
            ],
            "using": "gsi"
```

```
},
{
  "#operator": "Parallel",
  "maxParallelism": 1,
  "~child": {
    "#operator": "Sequence",
    "~children": [
      {
        "#operator": "Fetch",
        "keyspace": "ycsb",
        "namespace": "default"
      },
      {
        "#operator": "Filter",
        "condition": "(\"xyz1028\" <= (meta(`ycsb`).`id`))"
      },
      {
        "#operator": "InitialProject",
        "result_terms": [
          {
            "expr": "(`ycsb`.`field0`)"
          },
          {
            "expr": "(`ycsb`.`field1`)"
          },
          {
            "expr": "(`ycsb`.`field2`)"
          },
          {
            "expr": "(`ycsb`.`field3`)"
          },
          {
            "expr": "(`ycsb`.`field4`)"
```

```
                  },
                  {
                    "expr": "(`ycsb`.`field5`)"
                  },
                  {
                    "expr": "(`ycsb`.`field6`)"
                  },
                  {
                    "expr": "(`ycsb`.`field7`)"
                  },
                  {
                    "expr": "(`ycsb`.`Field8`)"
                  },
                  {
                    "expr": "(`ycsb`.`field9`)"
                  }
                ]
              }
            ]
          }
        }
      ]
    },
    {
      "#operator": "Limit",
      "expr": "50"
    },
    {
      "#operator": "FinalProject"
    }
  ]
},
   "text": "SELECT field0, field1, field2, field3, \n       field4, field5,
field6, field7,\n       Field8, field9\nFROM  ycsb\nWHERE meta().id >=
\"xyz1028\" \nORDER BY meta().id\nLIMIT 50;"
}
]
```

Each query goes thru multiple layers of execution within the query engine. The query engine orchestrates query execution among multiple indexes and data services, channeling the data through multiple operators.



Let's take this and explain the performance optimizations, indexing features implemented, exploited to get the best performance

1. Index Scan

Here's the explain snippet on index scan. In this case, we use the index meta_i1, with a predicate specified by the spans and the index scan is expected to return the first 50 qualified keys. Let's examine each of those decisions in in subsequent sections.

```
                "#operator": "IndexScan",
                 "index": "meta_i1",
                 "index_id": "bb26497ef8cf5fef",
                 "keyspace": "ycsb",
                 "limit": "50",
                 "namespace": "default",
                 "spans": [
                   {
                     "Range": {
                       "Inclusion": 1,
                       "Low": [
                         "\"xyz1028\""
                       ]
                     }
                   }
                 ],
                 "using": "gsi"
               },
```

2. Index Selection

Index selection in Couchbase is based on the predicates (filters) in the WHERE clause of the statement only. The index selection is made solely based on predicates and not any references to any other clauses like projection, grouping, ordering, etc.

In this statement, the WHERE clause is: meta().id >= "xyz1028". In this case, the match is quite straightforward. The index meta_i1 is chosen.

This is the very basic case. There are many subtle things about creating the right index and improving the throughput of the index scans. Let's discuss one by one.

In Couchbase, you can create many types of indices. We now have a secondary index meta_i1 on the document key (meta().id). In couchbase, you can create multiple indexes with same set of keys, but with different name.

```
CREATE INDEX meta_i2 ON ycsb(meta().id);
CREATE INDEX meta_i3 ON ycsb(meta().id);
CREATE INDEX meta_i4 ON ycsb(meta().id);
```

With these indices, the plan would choose the primary index, saving memory and CPU resources.

```
{
   "#operator": "Sequence",
   "~children": [
     {
       "#operator": "IndexScan",
       "index": "primary_i1",
       "index_id": "23db838eab4b16ab",
       "keyspace": "ycsb",
       "limit": "50",
       "namespace": "default",
       "spans": [
         {
           "Range": {
             "Inclusion": 1,
             "Low": [
               "\"xyz1028\""
             ]
           }
         }
       ],
       "using": "gsi"
     },
```

3. LIMIT Pushdown

In pagination queries, it's typical to limit the results what the screen can show. OFFSET and LIMIT keywords in the query help you to paginate through the resultset. From the application point of view, OFFSET and LIMIT is on the resultset of the whole query after all the select-join-sort-project operations are done.

However, if the index scan is being used for filtering the data, data is already ordered on the index keys. Then, if the ORDER BY is on the index keys, ascending and in the same order as the index keys, we can exploit the index ordering to provide the results in the expected order. This is significant, without the pushdown, we need to retrieve all of the qualified documents, sort them and then choose the specific window of results to return.

The primary index matches the predicate in the query. The ORDER BY is ascending by default. So, the predicate and the LIMIT is pushed down to the index scan. The plan includes the limit field with the number pushed down as well. When you want a high performing pagination queries, the LIMIT and OFFSET should be pushed down to the index scan.

```
SELECT field0, field1, field2, field3,
     field4, field5, field6, field7,
     Field8, field9xz
FROM  ycsb
```

```
WHERE meta().id >= "xyz1028"
ORDER BY meta().id
LIMIT 50;

     {
        "#operator": "Sequence",
        "~children": [
          {
            "#operator": "IndexScan",
            "index": "primary_i1",
            "index_id": "23db838eab4b16ab",
            "keyspace": "ycsb",
            "limit": "50",
            "namespace": "default",
            "spans": [
              {
                "Range": {
                  "Inclusion": 1,
                  "Low": [
                    "\"xyz1028\""
                  ]
                }
              }
            ],
            "using": "gsi"
          },
```

Another thing to notice is that, because the LIMIT is pushed down to index scan, the order, offset, limit operators become unnecessary and are removed from the plan. The best optimizations simply avoid the work altogether. This is one of them.

If we didn't have this optimization in place, or when the ORDER BY does not follow the index key order, you'd see something like this in the query plan.

```
          {
            "#operator": "Order",
            "limit": "50",
            "offset": "2500",
            "sort_terms": [
              {
                "expr": "(meta(`ycsb`).`id`)"
              }
            ]
          },
          {
            "#operator": "Offset",
            "expr": "2500"
          },
          {
            "#operator": "Limit",
            "expr": "50"
          },
          {
            "#operator": "FinalProject"
          }
        ]
```

If the query has both OFFSET and LIMIT, the sum of both are pushed down and the query engine simply skips over the keys required by the OFFSET.

```
explain SELECT field0, field1, field2, field3,
     field4, field5, field6, field7,
     Field8, field9
FROM  ycsb
WHERE meta().id >= "xyz1028"
```

```
ORDER BY meta().id
OFFSET 2500
LIMIT 50;
            "#operator": "IndexScan",
            "index": "primary_i1",
            "index_id": "23db838eab4b16ab",
            "keyspace": "ycsb",
            "limit": "(2500 + 50)",
            "namespace": "default",
```

3. Index Scan Range

The index scan range specified the by the spans. For this query, the range is (meta().id >= "xyz1028") and is specified by the following.

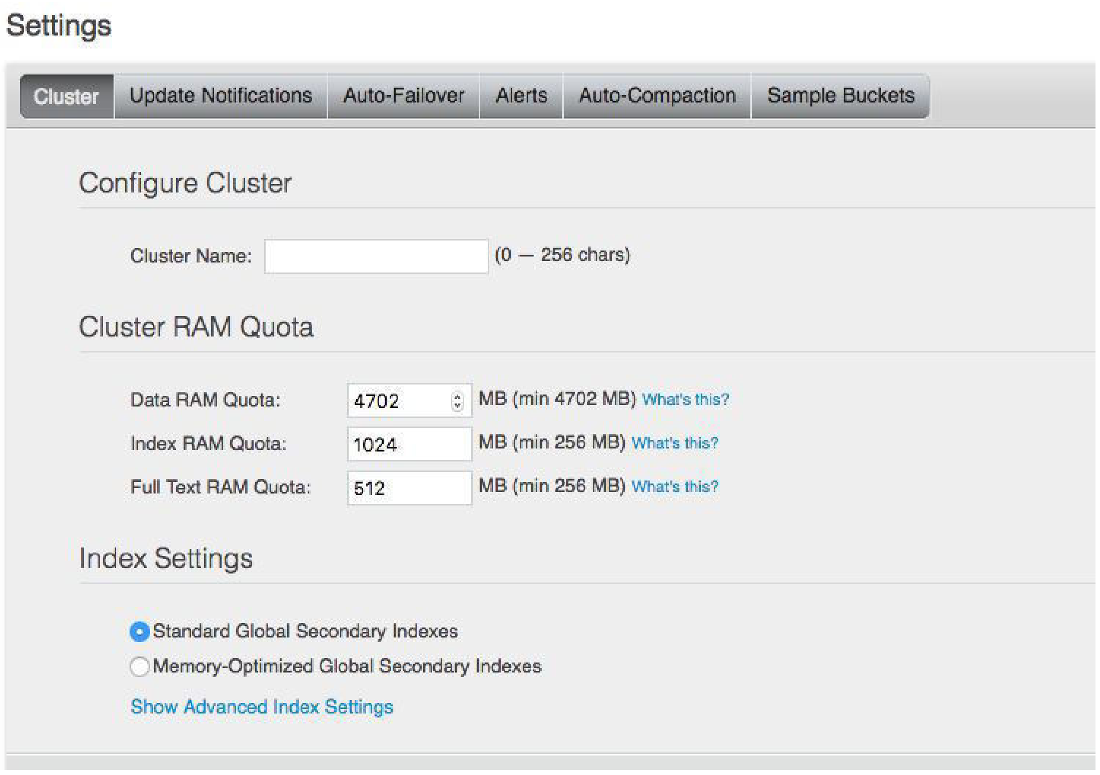
```
"spans": [
              {
                "Range": {
                  "Inclusion": 1,
                  "Low": [
                    "\"xyz1028\""
                  ]
                }
              }
            ],
```

4. Type of the Index

In my cluster, I have the default standard global secondary index. In the plan, we can see this in the field {"using": "gsi"}.

Couchbase 4.5 introduces memory optimized index. A memory-optimized index uses a novel lock-free skiplist to maintain the index and keeps 100% of the index data in memory. A memory-optimized index has better latency for index scans and can also process the mutations of the data much faster.

When you install Couchbase cluster, you'd need to select the type indexes you want to create in that cluster: Standard Global Secondary indexes, which uses the forestdb storage engine underneath and Memory-Optimized Global Secondary Index which keeps the full index in memory. More information at: http://developer.couchbase.com/documentation/server/4.5-dp/in-memory-indexes.html



Configure Cluster

Cluster Name: (0 — 256 chars)

Cluster RAM Quota

Data RAM Quota:	4702	MB (min 4702 MB)
Index RAM Quota:	1024	MB (min 256 MB)
Full Text RAM Quota:	512	MB (min 256 MB)

Index Settings

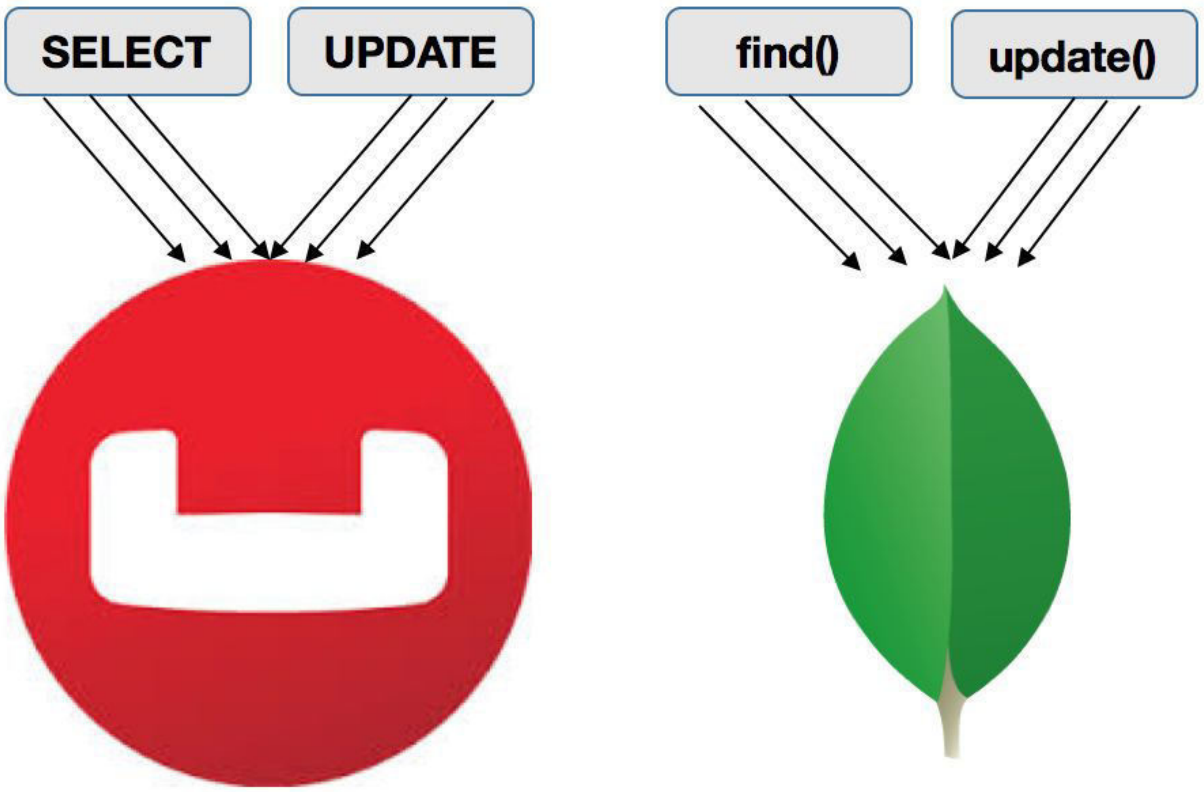
[✓ Standard Global Secondary Indexes]
Memory-Optimized Global Secondary Indexes

Summary

The duplicate indices, push down of the LIMIT, avoiding fetching extra keys from the index, avoiding sort — all done in a generalized way —helps optimize this YCSB scan query. This will help general queries with your application queries as well.

Concurrency Behavior: Couchbase vs. MongoDB

Author: Keshav Murthy



Multi-User Testing

David Glasser of Meteor wrote a blog on an MongoDB query missing matching documents issue he ran into. It is straightforward to reproduce the issue on both MongoDB MMAPv1 and the MongoDB WiredTiger engine. Here are his conclusions from his article (emphasis is mine)

Long story short…

This issue doesn't affect queries that don't use an index, such as queries that just look up a document by ID. It doesn't affect queries which explicitly do a single value equality match on all fields used in the index key. It doesn't affect queries that use indexes whose fields are never modified after the document is originally inserted. But any other kind of MongoDB query can fail to include all the matching documents!
Here's another way to look at it. In MongoDB, if the query can retrieve two documents using a secondary index (index on something other than _id) when concurrent operations are going on, the results could be wrong. This is a common scenario in many database applications.

Here is the test:

1. Create a container: bucket, table, or collection.
2. Load the data with a small dataset, say 300,000 documents.
3. Create an index on the field you want to filter on (predicates).
4. In one session, update the indexed field in one session and query on the other.

MongoDB Testing

Steps to reproduce the issue on MongoDB:

1. Install MongoDB 3.2.
2. Bring up MongoDB with either MMAPv1 or WiredTiger.
3. Load data using tpcc.py.
4. python tpcc.py --warehouses 1 --no-execute mongodb.
5. Get the count.
6. `> use tpcc`
7. `> db.ORDER_LINE.find().count();`
8. `299890`

9. db.ORDER_LINE.ensureIndex({state:1});

MongoDB Experiment 1: Update to a Higher Value

Set up the state field with the value aaaaaa and then concurrently update this value to zzzzzz and query for total number of documents with the two values ['aaaaaa','zzzzzz'] matching the field. When the value of the indexed field moves from lower (aaaaaa) to higher (zzzzzz) value, these entries are moving from one side of the B-tree to the other. Now, we're trying to see if the scan returns duplicate value, translated to higher count() value.

```
> db.ORDER_LINE.update({OL_DIST_INFO:{$gt:""}}, {$set: {state:"aaaaaa"}}, {multi:true});
WriteResult({ "nMatched" : 299890, "nUpserted" : 0, "nModified" : 299890 })
> db.ORDER_LINE.find({state:{$in:['aaaaaa','zzzzzz']}}).count();
299890
> db.ORDER_LINE.find({state:{$in:['aaaaaa','zzzzzz']}}).explain();
```

Following is the explain for this find() query. It's using the index to evaluate the IN predicate.

```
{
  "queryPlanner" : {
      "plannerVersion" : 1,
      "namespace" : "tpcc.ORDER_LINE",
      "indexFilterSet" : false,
      "parsedQuery" : {
          "state" : {
              "$in" : [
                  "aaaaaa",
                  "zzzzzz"
              ]
```

```
                }
            },
            "winningPlan" : {
                "stage" : "FETCH",
                "inputStage" : {
                    "stage" : "IXSCAN",
                    "keyPattern" : {
                        "state" : 1
                    },
                    "indexName" : "state_1",
                    "isMultiKey" : false,
                    "direction" : "forward",
                    "indexBounds" : {
                        "state" : [
                            "[\"aaaaaa\", \"aaaaaa\"]",
                            "[\"zzzzzz\", \"zzzzzz\"]"
                        ]
                    }
                }
            },
            "rejectedPlans" : [ ]
        },
        "serverInfo" : {
            "host" : "Keshavs-MacBook-Pro.local",
            "port" : 27017,
            "version" : "3.0.2",
            "gitVersion" : "6201872043ecbbc0a4cc169b5482dcf385fc464f"
        },
        "ok" : 1
    }
```

1. Update statement 1: Update all documents to set state = “zzzzzz”.

```
db.ORDER_LINE.update({OL_DIST_INFO:{$gt:""}},
                    {$set: {state: "zzzzzz"}}, {multi:true});
```

4.
5. Update statement 2: Update all documents to set state = “aaaaaa”.

```
db.ORDER_LINE.update({OL_DIST_INFO:{$gt:""}},
                     {$set: {state: "aaaaaa"}}, {multi:true});
```

3. Count statement: Count documents:(state in [“aaaaaa”, “zzzzzz”])

   db.ORDER_LINE.find({state:{$in:['aaaaaa','zzzzzz']}}).count();

Time	Session 1: Issue Update Statement1 (update state = “zzzzzz”)	Session 2: Issue Count Statement continuously.
T0	Update Statement starts	Count = 299890
T1	Update Statement Continues	Count = 312736
T2	Update Statement Continues	Count = 312730
T3	Update Statement Continues	Count = 312778
T4	Update Statement Continues	Count = 312656
T4	Update Statement Continues	Count = 313514
T4	Update Statement Continues	Count = 303116
T4	Update Statement Done	Count = 299890

Result: In this scenario, the index double counts many of the documents and reports more than it actually has.

Cause: Data in the leaf level of B-Tree is sorted. As the update B-Tree gets updated from aaaaaa to zzzzzz, the keys in the lower end are moved to the upper end. The concurrent scans are unaware of this move. MongoDB does not implement a stable scan and counts the entries as they come. So, in a production system with many updates going on, it could count the same document twice, thrice, or more. It just depends on the concurrent operations.

MongoDB Experiment 2: Update to a Lower Value

Let’s do the reverse operation to update the data from ‘zzzzzz’ to ‘aaaaaa’. In this case, the index entries are moving from a higher value to a lower value, thus causing the scan to miss some of the qualified documents, shown to be undercounting.

Time	Session 1: Issue Update Statement2 (update state = “aaaaaa”)	Session 2: Issue Count Statement continuously.
T0	Update Statement starts	Count = 299890
T1	Update Statement Continues	Count = 299728
T2	Update Statement Continues	Count = 299750
T3	Update Statement Continues	Count = 299780
T4	Update Statement Continues	Count = 299761
T4	Update Statement Continues	Count = 299777
T4	Update Statement Continues	Count = 299815
T4	Update Statement Done	Count = 299890

Result: In this scenario, the index misses many documents and reports fewer documents than it actually has.
Cause: This exposes the reverse effect. When the keys with value zzzzzz is modified to aaaaaa, items go from the higher to the lower end of the B-Tree. Again, since there is no stability in scans, it would miss the keys that moved from the higher end to the lower end.

MongoDB Experiment 3: Concurrent UPDATES

Two sessions update the indexed field concurrently and continuously. In this case, based on the the prior observations, each of the sessions run into both overcount and undercount issue. The nModified result varies because MongoDB reports only updates that changed the value.
But the total number of modified documents is never more than 299980. So, MongoDB does avoid updating the same document twice, thus handling the classic Halloween problem. Because they don't have a stable scan, I presume they handle this by maintaining lists of objectIDs updated during this multi-update statement and avoiding the update if the same object comes up as a qualified document.

Session 1

```
>  db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"aaaaaa"}}, {multi:true});
WriteResult({ "nMatched" : 299890, "nUpserted" : 0, "nModified" : 299890 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"zzzzzz"}}, {multi:true});
WriteResult({ "nMatched" : 303648, "nUpserted" : 0, "nModified" : 12026 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"aaaaaa"}}, {multi:true});
WriteResult({ "nMatched" : 194732, "nUpserted" : 0, "nModified" : 138784 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"zzzzzz"}}, {multi:true});
WriteResult({ "nMatched" : 334134, "nUpserted" : 0, "nModified" : 153625 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"aaaaaa"}}, {multi:true});
WriteResult({ "nMatched" : 184379, "nUpserted" : 0, "nModified" : 146318 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"zzzzzz"}}, {multi:true});
WriteResult({ "nMatched" : 335613, "nUpserted" : 0, "nModified" : 153403 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"aaaaaa"}}, {multi:true});
WriteResult({ "nMatched" : 183559, "nUpserted" : 0, "nModified" : 146026 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"zzzzzz"}}, {multi:true});
WriteResult({ "nMatched" : 335238, "nUpserted" : 0, "nModified" : 149337 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"aaaaaa"}}, {multi:true});
WriteResult({ "nMatched" : 187815, "nUpserted" : 0, "nModified" : 150696 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"zzzzzz"}}, {multi:true});
WriteResult({ "nMatched" : 335394, "nUpserted" : 0, "nModified" : 154057 })
```

```
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"aaaaaa"}}, {multi:true});
WriteResult({ "nMatched" : 188774, "nUpserted" : 0, "nModified" : 153279 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"zzzzzz"}}, {multi:true});
WriteResult({ "nMatched" : 334408, "nUpserted" : 0, "nModified" : 155970 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"zzzzzz"}}, {multi:true});
WriteResult({ "nMatched" : 299890, "nUpserted" : 0, "nModified" : 0 })
>
```

Session 2

```
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"zzzzzz"}}, {multi:true});
WriteResult({ "nMatched" : 302715, "nUpserted" : 0, "nModified" : 287864 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"aaaaaa"}}, {multi:true});
WriteResult({ "nMatched" : 195248, "nUpserted" : 0, "nModified" : 161106 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"zzzzzz"}}, {multi:true});
WriteResult({ "nMatched" : 335526, "nUpserted" : 0, "nModified" : 146265 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"aaaaaa"}}, {multi:true});
WriteResult({ "nMatched" : 190448, "nUpserted" : 0, "nModified" : 153572 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"zzzzzz"}}, {multi:true});
WriteResult({ "nMatched" : 336734, "nUpserted" : 0, "nModified" : 146487 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"aaaaaa"}}, {multi:true});
WriteResult({ "nMatched" : 189321, "nUpserted" : 0, "nModified" : 153864 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"zzzzzz"}}, {multi:true});
WriteResult({ "nMatched" : 334793, "nUpserted" : 0, "nModified" : 150553 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"aaaaaa"}}, {multi:true});
WriteResult({ "nMatched" : 186274, "nUpserted" : 0, "nModified" : 149194 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"zzzzzz"}}, {multi:true});
WriteResult({ "nMatched" : 336576, "nUpserted" : 0, "nModified" : 145833 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"aaaaaa"}}, {multi:true});
WriteResult({ "nMatched" : 183635, "nUpserted" : 0, "nModified" : 146611 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"zzzzzz"}}, {multi:true});
WriteResult({ "nMatched" : 336904, "nUpserted" : 0, "nModified" : 143920 })
>
```

Couchbase Testing

1. Install Couchbase 4.5.
2. Load data using tpcc.py.
3. python tpcc.py --warehouses 1 --no-execute n1ql.
4. Get the count.
5. `> SELECT COUNT(*) FROM ORDER_LINE;`
6. `300023`
7. CREATE INDEX i1 ON ORDER_LINE(state);
8. UPDATE ORDER_LINE SET state = ‘aaaaaa’;

Couchbase Experiment 1: Update to a Higher Value

Do the initial setup using the following.

```
> UPDATE ORDER_LINE SET state = ‘aaaaaa’ WHERE OL_DIST_INFO > “”;
Verify the Count.  state field(attribute) with values “aaaaaa” is 300023.
> select count(1) a_cnt FROM ORDER_LINE where state =  'aaaaaa'
UNION ALL
select count(1) z_cnt FROM ORDER_LINE where state =  'zzzzzz';
 "results": [
     {
         "a_cnt": 300023
     },
     {
         "z_cnt": 0
     }
 ],
```

Let’s ensure the index scan happens on the query:

```
EXPLAIN SELECT COUNT(1) AS totalcnt
FROM ORDER_LINE
WHERE state = 'aaaaaa' or state = 'zzzzzz';
```

```
"~children": [
      {
          "#operator": "DistinctScan",
          "scan": {
              "#operator": "IndexScan",
              "covers": [
                  "cover ((`ORDER_LINE`.`state`))",
                  "cover ((meta(`ORDER_LINE`).`id`))"
              ],
              "index": "i2",
              "index_id": "665b11a6c36d4136",
              "keyspace": "ORDER_LINE",
              "namespace": "default",
              "spans": [
                  {
                      "Range": {
                          "High": [
                              "\"aaaaaa\""
                          ],
                          "Inclusion": 3,
                          "Low": [
                              "\"aaaaaa\""
                          ]
                      }
                  },
                  {
                      "Range": {
                          "High": [
                              "\"zzzzzz\""
                          ],
                          "Inclusion": 3,
                          "Low": [
                              "\"zzzzzz\""
```

```
                                            ]
                                        }
                                    }
                                ],
                                "using": "gsi"
                            }
                        },
```

We can also use UNION ALL of two separate predicates (state = ‘aaaaaa’) and (state = ‘zzzzzz’) to have the index count efficiently:

```
cbq> explain select count(1) a_cnt
FROM ORDER_LINE
where state =  'aaaaaa'
UNION ALL
select count(1) z_cnt
FROM ORDER_LINE
where state =  'zzzzzz';
{
 "requestID": "ef99e374-48f5-435c-8d54-63d1acb9ad22",
 "signature": "json",
 "results": [
     {
         "plan": {
             "#operator": "UnionAll",
             "children": [
                 {
                     "#operator": "Sequence",
                     "~children": [
                         {
                             "#operator": "IndexCountScan",
                             "covers": [
                                 "cover ((`ORDER_LINE`.`state`))",
                                 "cover ((meta(`ORDER_LINE`).`id`))"
```

```
                    ],
                    "index": "i2",
                    "index_id": "665b11a6c36d4136",
                    "keyspace": "ORDER_LINE",
                    "namespace": "default",
                    "spans": [
                        {
                            "Range": {
                                "High": [
                                    "\"aaaaaa\""
                                ],
                                "Inclusion": 3,
                                "Low": [
                                    "\"aaaaaa\""
                                ]
                            }
                        }
                    ],
                    "using": "gsi"
                },
                {
                    "#operator": "IndexCountProject",
                    "result_terms": [
                        {
                            "as": "a_cnt",
                            "expr": "count(1)"
                        }
                    ]
                }
            ]
        },
        {
            "#operator": "Sequence",
"~children": [
    {
        "#operator": "IndexCountScan",
        "covers": [
            "cover ((`ORDER_LINE`.`state`))",
            "cover ((meta(`ORDER_LINE`).`id`))"
        ],
        "index": "i2",
        "index_id": "665b11a6c36d4136",
        "keyspace": "ORDER_LINE",
        "namespace": "default",
        "spans": [
            {
                "Range": {
                    "High": [
                        "\"zzzzzz\""
                    ],
                    "Inclusion": 3,
                    "Low": [
                        "\"zzzzzz\""
                    ]
                }
            }
        ],
        "using": "gsi"
    },
    {
        "#operator": "IndexCountProject",
        "result_terms": [
            {
                "as": "z_cnt",
                "expr": "count(1)"
            }
```

```
                                                  ]
                                              }
                                          ]
                                      }
                                  ]
                              },
                              "text": "select count(1) a_cnt FROM ORDER_LINE where state =  'aaaaaa'
                    UNION ALL select count(1) z_cnt FROM ORDER_LINE where state =  'zzzzzz'"
                          }
                      ],
                      "status": "success",
                      "metrics": {
                          "elapsedTime": "2.62144ms",
                          "executionTime": "2.597189ms",
                          "resultCount": 1,
                          "resultSize": 3902
                      }
                  }
```

Set up the state field with the value aaaaaa. Then update this value to zzzzzz and concurrently query for total number of documents with the either of the two values.

Session 1

Update state field to value zzzzzz:

```
UPDATE ORDER_LINE SET state = ‘zzzzzz’ WHERE OL_DIST_INFO > “”;
{ "utationCount": 300023 }
```

Session 2

Query the count of ‘aaaaaa’ and ‘zzzzzz’ from ORDER_LINE.

Time	Session 1: Issue Update Statement1 (update state = “zzzzzz”)	a_cnt	z_cnt	Total
T0	Update Statement starts	300023	0	300023

T1	Update Statement Continues	288480	11543	300023
T2	Update Statement Continues	259157	40866	300023
T3	Update Statement Continues	197167	102856	300023
T4	Update Statement Continues	165449	134574	300023
T5	Update Statement Continues	135765	164258	300023
T6	Update Statement Continues	86584	213439	300023
T7	Update Statement Done	0	300023	300023

Result: Index updates happen as the data gets updated. As the values migrate from ‘aaaaaa’ to ‘zzzzzz,’ they’re not double counted.

Couchbase indexes provide stable scans by snapshotting the index at regular frequency. While this is a considerable engineering effort, as we’ve seen from this issue, it provides stability of answers even under extreme concurrency situations. The data that the index scan retrieves will be from the point the scan starts. Subsequent updates coming in concurrently, will not be returned. This provides another level of stability as well.

It’s important to note that stability of scans is provided for each index scan. Indices take snapshots every 200 milliseconds. When you have a long running query with multiple index scans on the same or multiple indices, the query could use multiple snapshots. So, different index scans in a long running query will each return different results. This is an use case we’ll be improving in a future release to use same snapshot across scans of a single query.

Couchbase Experiment 2: Update to a Lower Value

Let’s do the reverse operation to update the data from ‘zzzzzz’ back to ‘aaaaaa’.

Session 1

Update state field to value aaaaaa

```
UPDATE ORDER_LINE SET state = ‘aaaaaa’ WHERE OL_DIST_INFO > “”;
{ "mutationCount": 300023 }
```

Session 2

Query the count of ‘aaaaaa’ and ‘zzzzzz’ from ORDER_LINE:

Time	Session 1: Issue Update Statement1 (update state = “aaaaaa”)	a_cnt	z_cnt	Total
T0	Update Statement starts	0	300023	**300023**
T1	Update Statement Continues	28486	271537	**300023**
T2	Update Statement Continues	87919	212104	**300023**
T3	Update Statement Continues	150630	149393	**300023**
T4	Update Statement Continues	272358	27665	**300023**
T5	Update Statement Continues	299737	286	**300023**
T6	Update Statement Done	0	300023	**300023**

Couchbase Experiment 3: Concurrent Updates

Two sessions update the indexed field concurrently and continuously. The stable scans of the index alway returns the full list of qualified documents: 30023 and Couchbase updates all of the documents and reports the mutation count as 30023. An update is an update regardless of whether the old value is the same as the new value or not.

Session 1

```
    update ORDER_LINE set state = 'aaaaaa' where state > "";
    {     "mutationCount": 300023 }
    update ORDER_LINE set state =  ‘zzzzzz'where state > "";
```

```
{     "mutationCount": 300023 }
update ORDER_LINE set state = 'aaaaaa' where state > "";
{     "mutationCount": 300023 }
update ORDER_LINE set state =  ‘zzzzzz'where state > "";
{     "mutationCount": 300023 }
update ORDER_LINE set state = 'aaaaaa' where state > "";
{     "mutationCount": 300023 }
update ORDER_LINE set state =  ‘zzzzzz'where state > "";
{     "mutationCount": 300023 }
update ORDER_LINE set state = 'aaaaaa' where state > "";
{     "mutationCount": 300023 }
update ORDER_LINE set state =  ‘zzzzzz'where state > "";
{     "mutationCount": 300023 }
```

Session 2

```
update ORDER_LINE set state =  ‘zzzzzz'where state > "";
{     "mutationCount": 300023 }
update ORDER_LINE set state = 'aaaaaa' where state > "";
{     "mutationCount": 300023 }
update ORDER_LINE set state =  ‘zzzzzz'where state > "";
{     "mutationCount": 300023 }
update ORDER_LINE set state = 'aaaaaa' where state > "";
{     "mutationCount": 300023 }
update ORDER_LINE set state =  ‘zzzzzz'where state > "";
{     "mutationCount": 300023 }
update ORDER_LINE set state = 'aaaaaa' where state > "";
{     "mutationCount": 300023 }
update ORDER_LINE set state =  ‘zzzzzz'where state > "";
{     "mutationCount": 300023 }
```

Conclusions

MongoDB:

1. MongoDB queries will miss documents if there are concurrent updates that move the data from one portion of the index to another portion of the index (higher to lower).
2. MongoDB queries will return the same document multiple times if there are concurrent updates that move the data from one portion of the index to another portion of the index (lower to higher).
3. Concurrent multi-updates on MongoDB will run into the same issue and will miss updating all the required documents, but they do avoid updating the same document twice in a single statement.
4. When developing applications that use MongoDB, you must design a data model so you select and update only one document for each query. Avoid multi-document queries in MongoDB since it will return incorrect results when there are concurrent updates.

Couchbase:

1. Couchbase returns the expected number of qualifying documents even when there are concurrent updates. Stable index scans in Couchbase provide the protection to the application that MongoDB does not.
2. Concurrent updates benefit from stable index scans and process all of the qualified documents when the application statement is issued.

Acknowledgements

Thanks to Sriram Melkote and Deepkaran Salooja from Couchbase Indexing team for their review and valuable input to the effort.

References

1. MongoDB Strong Consistency: https://www.mongodb.com/nosql-explained
2. MongoDB Concurrency: https://docs.mongodb.com/manual/faq/concurrency/
3. MongoDB queries don't always return all matching documents!: https://engineering.meteor.com/mongodb-queries-dont-always-return-all-matching-documents-654b6594a827#.s9y0yheuv
4. N1QL: http://query.couchbase.com

N1QL

Journey Continues

http://query.couchbase.com