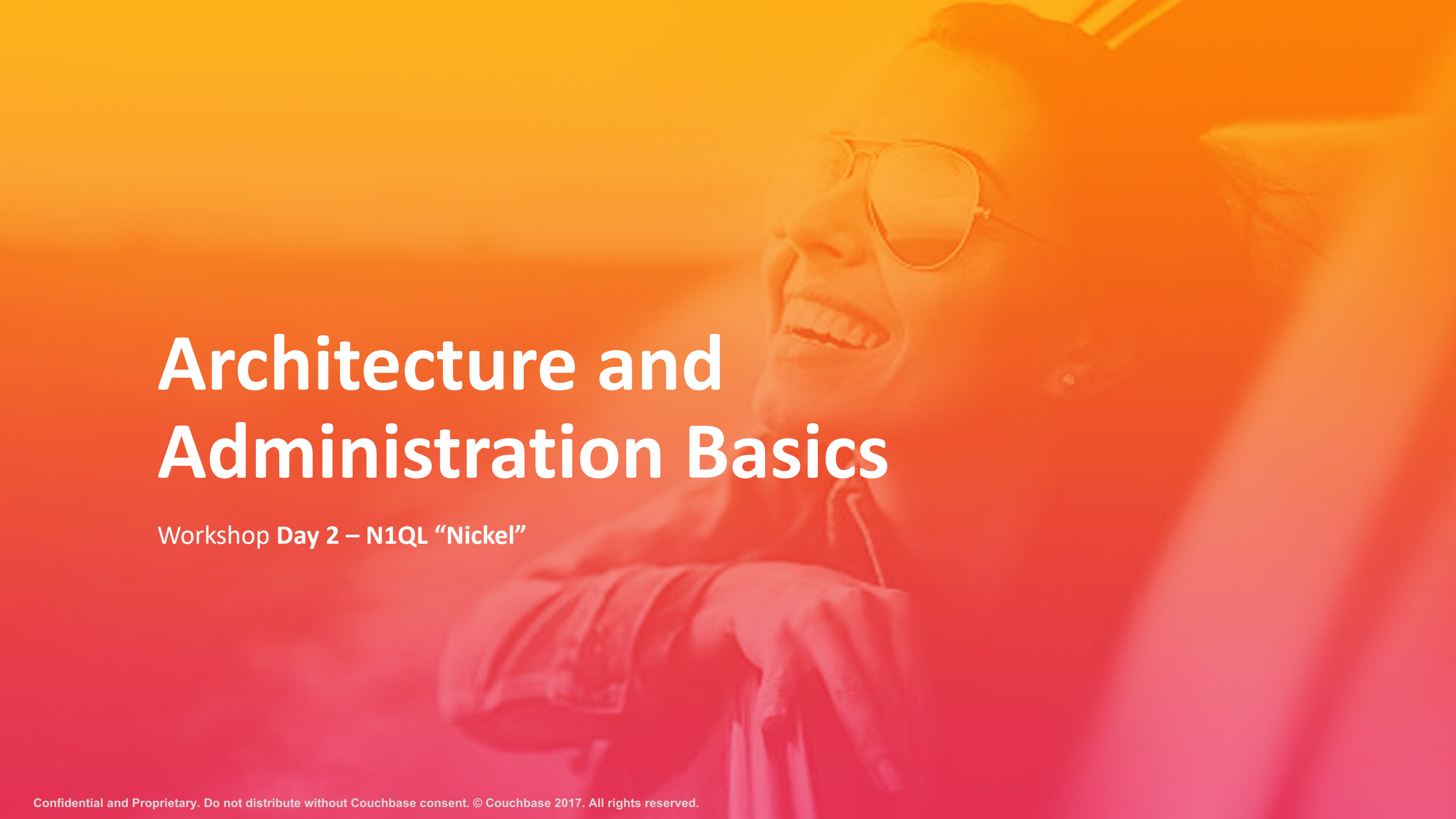


Architecture and Administration Basics

Workshop **Day 2 – N1QL “Nickel”**

1 | Motivation for N1QL

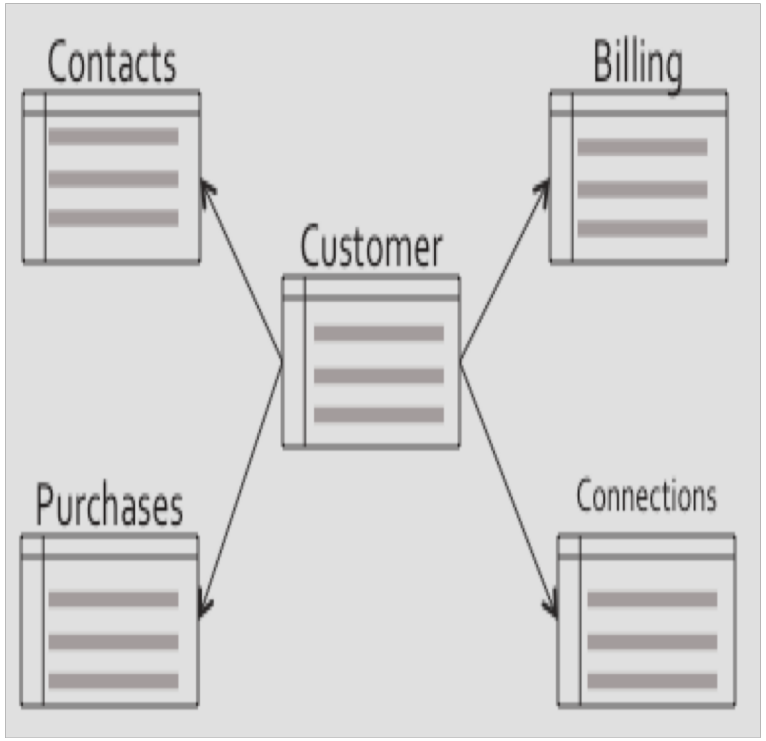
What is N1QL?

- **Non-first (N1)** Normal Form **Query Language (QL)**
  - It is based on ANSI 92 SQL
  - Its query engine is optimized for modern, highly parallel multi-core execution
- **SQL-like Query Language**
  - Expressive, familiar, and feature-rich language for querying, transforming, and manipulating JSON data
- N1QL extends SQL to handle data that is:
  - **Nested**: Contains nested objects, arrays
  - **Heterogeneous**: Schema-optional, non-uniform
  - **Distributed**: Partitioned across a cluster

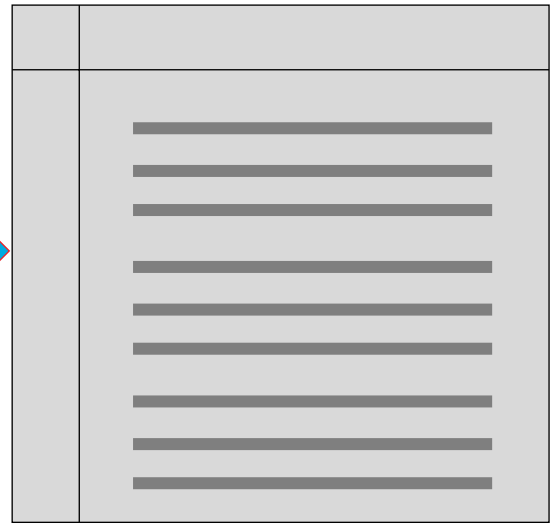
Power of SQL

Flexibility of JSON

Relations/Tuples



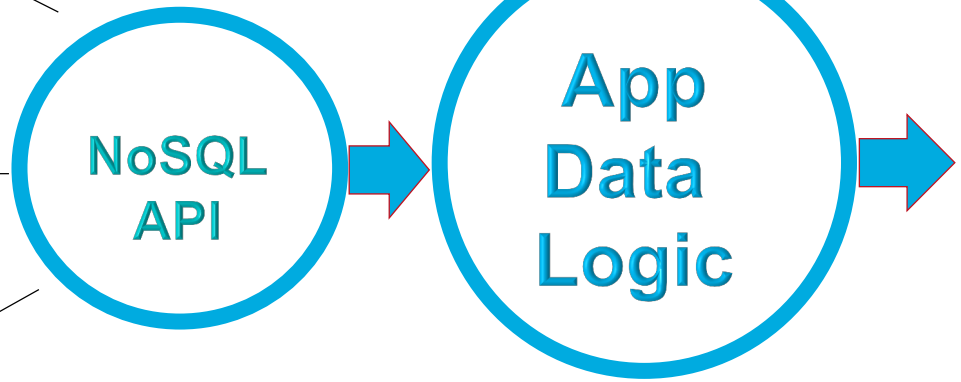
ResultSet



LoyaltyInfo

CUSTOMER

Orders

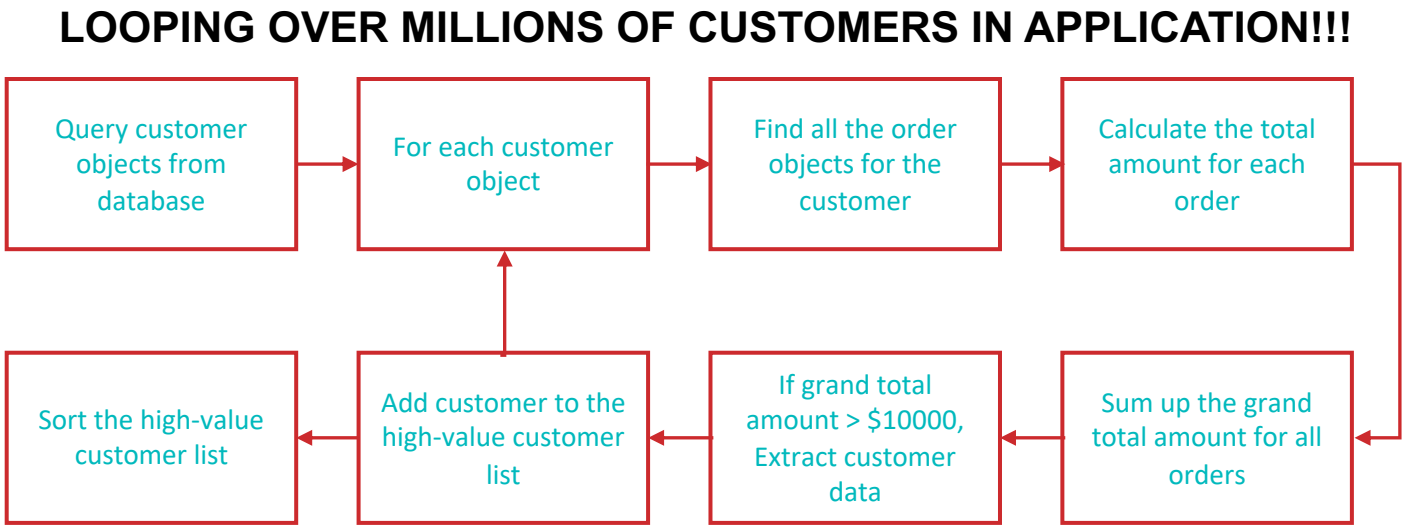


- NoSQL systems provide specialized APIs
- Key-Value get and set
- Each task requires custom built program
- Should test & maintain it

Results

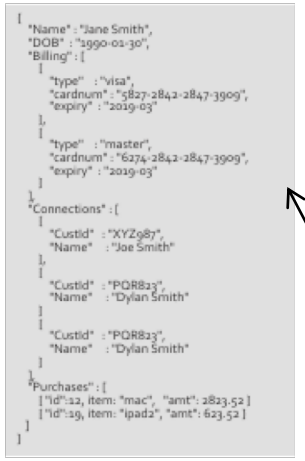
```
{
   "Name" : "Jane Smith",
   "DOB"  : "1990-01-30",
   "Billing" : [
      {
         "type"    : "visa",
         "cardnum" : "5827-2842-2847-3909",
         "expiry"  : "2019-03"
      },
      {
         "type"    : "master",
         "cardnum" : "6274-2842-2847-3909",
         "expiry"  : "2019-03"
      }
   ],
   "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 }
   ]
}
```

Find High-Value Customers with Orders > $10000



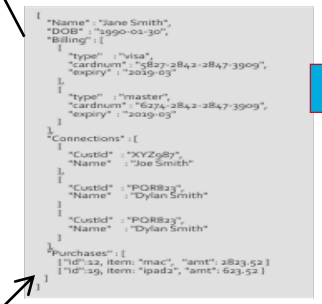
- Complex codes and logic
- Inefficient processing on client side

LoyaltyInfo



Orders

CUSTOMER



{N1QL}

ResultDocuments


```
{
   "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 }
   ]
}
```

N1QL = SQL + JSON

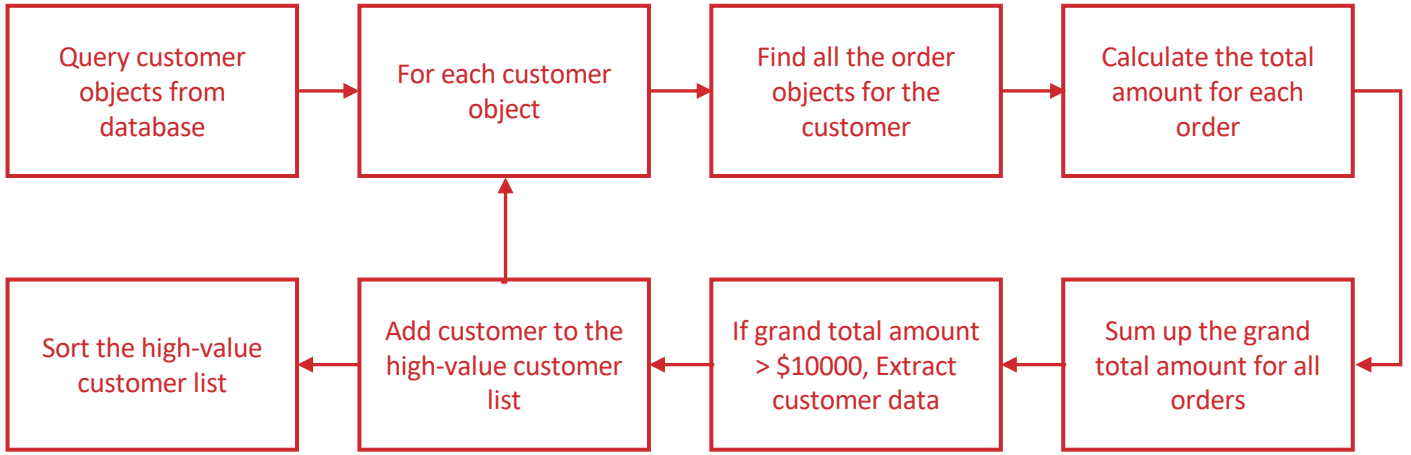
Give developers and enterprises an expressive, powerful, and complete language for querying, transforming, and manipulating JSON data.

Why SQL for NoSQL?

Find High-Value Customers with Orders > $10000

API QUERY

LOOPING OVER MILLIONS OF CUSTOMERS IN APPLICATION!!!



- Complex codes and logic
- Inefficient processing on client side

VS.

SQL for JSON

```
SELECT   customers.id, customers.NAME.lastname,
         customers.NAME.firstname, sum(orderline.amount)
FROM     orders UNNEST orders.lineitems AS orderline
         INNER JOIN customers ON KEYS orders.custid
WHERE customers.state = 'NY'
GROUP BY customers.id, customers.NAME.lastname,
         customers.NAME.firstname
HAVING   sum(orderline.amount) > 10000
ORDER BY sum(orderline.amount) DESC
```

- Proven and expressive query language
- Leverage SQL skills and ecosystem
- Extended for JSON

N1QL = SQL + JSON

```
SELECT    customers.id,
          customers.NAME.lastname,
          customers.NAME.firstname
          Sum(orderline.amount)
FROM      orders UNNEST orders.lineitems AS orderline
          INNER JOIN customers ON KEYS orders.custid
WHERE customers.state = 'NY'
GROUP BY customers.id,
          customers.NAME.lastname,
               customers.NAME.firstname
HAVING    sum(orderline.amount) > 10000
ORDER BY sum(orderline.amount) DESC
```

- Dotted sub-document reference
- Names are CASE-SENSITIVE

UNNEST to flatten the arrays

JOINS with Document KEY of customers

2 | N1QL Language

N1QL: Data Manipulation Statements

- SELECT ...
- UPDATE … SET … WHERE …
- DELETE FROM … WHERE …
- INSERT INTO … (KEY, VALUE) VALUES …
- INSERT INTO … (KEY …, VALUE …) SELECT …
- MERGE INTO … USING … ON …

  WHEN [NOT] MATCHED THEN …
- EXPLAIN ...

Note: Couchbase provides per-document atomicity.

N1QL: Data Definition Statements

Creating and deleting indexes:

- **CREATE INDEX … ON …**
- **DROP INDEX …**
- **BUILD INDEX …**

Granting/revoking user roles

- **GRANT … ON … TO …**
- **REVOKE … ON … FROM …**

Data Modeling with JSON: Travel Sample

Route Entity

Key: route_10000
Value:

```
{
  "id": 10000,
  "type": "route",
  "airline": "AF",
  "airlineid": "airline_137",
  "sourceairport": "TLV",
  "destinationairport": "MRS",
  "distance": 2881.617376098415,
  "equipment": "320",
  "schedule": [
     { "day": 0, "flight": "AF198", "utc": "10:13:00" },
     { "day": 1, "flight": "AF356", "utc": "12:40:00" },
     ...
   ]
}
```

Airline Entity

Key: airline_137
Value:

```
{
  "id": 137,
  "type": "airline",
  "callsign": "AIRFRANS",
  "country": "France",
  "iata": "AF",
  "icao": "AFR",
  "name": "Air France"
}
```

Data Modeling with JSON: Tables

Route Entity

Key: route_10000
Value:

```
{
  "id": 10000,
  "type": "route",
  "airline": "AF",
  "airlineid": "airline_137",
  "sourceairport": "TLV",
  "destinationairport": "MRS",
  "distance": 2881.617376098415,
  "equipment": "320",
  "schedule": [
     { "day": 0, "flight": "AF198", "utc": "10:13:00" },
     { "day": 1, "flight": "AF356", "utc": "12:40:00" },
     ...
   ]
}
```

Airline Entity

Key: airline_137
Value:

```
{
  "id": 137,
  "type": "airline",
  "callsign": "AIRFRANS",
  "country": "France",
  "iata": "AF",
  "icao": "AFR",
  "name": "Air France"
}
```

There is no explicit notion of table, so entity type is identified by an attribute or by the prefix of a key

Data Modeling with JSON: Relationships

Relationships are expressed by referencing the key or a unique attribute or another document

Route Entity

Key: route_10000
Value:

```
{
  "id": 10000,
  "type": "route",
  "airline": "AF",
  "airlineid": "airline_137",
  "sourceairport": "TLV",
  "destinationairport": "MRS",
  "distance": 2881.617376098415,
  "equipment": "320",
  "schedule": [
    { "day": 0, "flight": "AF198", "utc": "10:13:00" },
    { "day": 1, "flight": "AF356", "utc": "12:40:00" },
    ...
  ]
}
```

Airline Entity

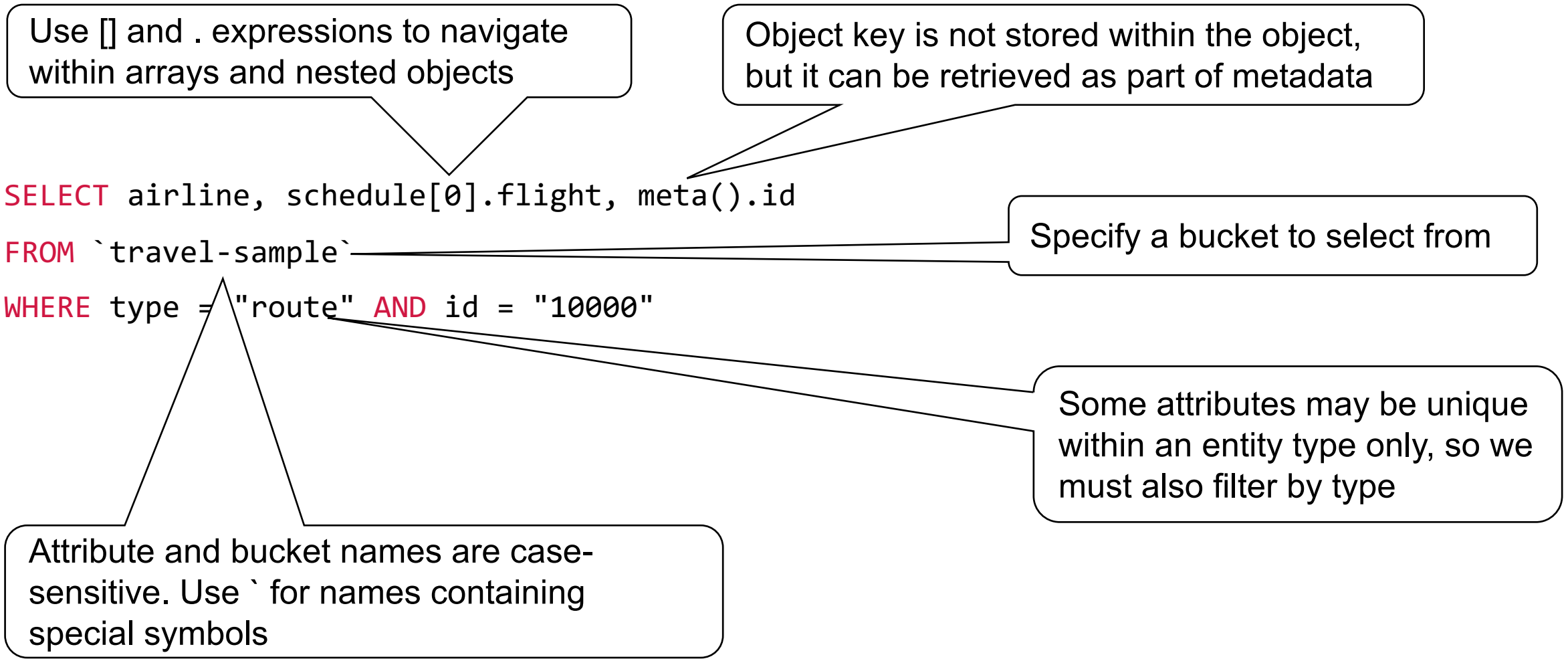
Key: airline_137
Value:

```
{
  "id": 137,
  "type": "airline",
  "callsign": "AIRFRANS",
  "country": "France",
  "iata": "AF",
  "icao": "AFR",
  "name": "Air France"
}
```

Small dependent entities are simply embedded hierarchically as subobjects or elements of an array

Many-to-many relationships can be expressed by arrays of references, e.g. “flight“ could be reference to a further document

SELECT Statement: Basics



Indexes: Basics

```
SELECT airline, schedule[0].flight, meta().id
FROM `travel-sample`
WHERE type = "route" AND id = "10000"
```

- A query can be executed only if a suitable index is available, e.g.

```
CREATE INDEX idx_type_id ON `travel-sample` (type, id);
```

- Primary index enables full bucket scan and thus execution of arbitrary queries

```
CREATE PRIMARY INDEX idx_primary ON `travel-sample`;
```

- Primary index is good for prototyping but should not be used in production

USE KEYS

```
SELECT airline, schedule[0].flight, meta().id
FROM `travel-sample`
USE KEYS ["route_10000", "route_10001"]
```

- USE KEYS filters objects by their keys
- Direct primary key lookup by passing index scans
- Ideal for hash-distributed datastore
- Available in SELECT, UPDATE, DELETE

JOIN ON KEYS: Referential Join

```
SELECT rt.sourceairport, rt.destinationairport, al.name
FROM `travel-sample` rt JOIN `travel-sample` al ON KEYS rt.airlineid
WHERE rt.type = "route" LIMIT 10;
```

Route Entity

Key: route_10000
Value:

```
{
  "id": 10000,
  "type": "route",
  "airline": "AF",
  "airlineid": "airline_137",
  "sourceairport": "TLV",
  "destinationairport": "MRS",
  ...
}
```

Airline Entity

Key: airline_137
Value:

```
{
  "id": 137,
  "type": "airline",
  "callsign": "AIRFRANS",
  "country": "France",
  "iata": "AF",
  "icao": "AFR",
  "name": "Air France"
}
```

ANSI Joins

```
SELECT rt.sourceairport, rt.destinationairport, al.name
FROM `travel-sample` rt JOIN `travel-sample` al
  ON rt.airline = al.iata AND al.type = "airline"
WHERE rt.type = "route" LIMIT 10;
```

Route Entity

Key: route_10000
Value:

```
{
  "id": 10000,
  "type": "route",
  "airline": "AF",
  "airlineid": "airline_137",
  "sourceairport": "TLV",
  "destinationairport": "MRS",
  ...
}
```

Airline Entity

Key: airline_137
Value:

```
{
  "id": 137,
  "type": "airline",
  "callsign": "AIRFRANS",
  "country": "France",
  "iata": "AF",
  "icao": "AFR",
  "name": "Air France"
}
```

https://blog.couchbase.com/ansi-join-support-n1ql/

UNNEST: Join with Embedded Arrays

```
SELECT rt.sourceairport, rt.destinationairport,
       sc.flight
FROM `travel-sample` rt UNNEST schedule sc
WHERE rt.type = "route" LIMIT 10;
```

- Unnest flattens an array withing an object
- Effectively, it performs a join of top level objects with the nested ones

Route Entity

Key: route_10000
Value:

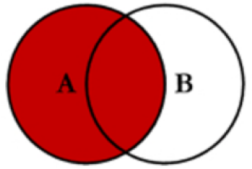
```
{
  "id": 10000,
  "type": "route",
  "airline": "AF",
  "airlineid": "airline_137",
  "sourceairport": "TLV",
  "destinationairport": "MRS",
  "distance": 2881.617376098415,
  "equipment": "320",
  "schedule": [
    { "day": 0, "flight": "AF198", "utc": "10:13:00" },
    { "day": 1, "flight": "AF356", "utc": "12:40:00" },
    ...
  ]
}
```

Join Types

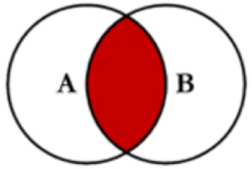
- Inner, Left, Right, Outer, Exclude,

```
SELECT <select_list>
FROM bucket A
JOIN bucket B
ON KEYS <keys-clause(A)>
```

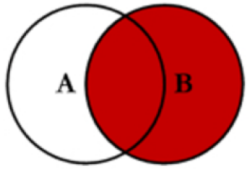
N1QL JOINs



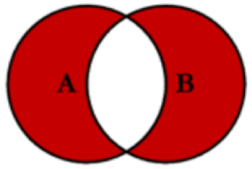
SELECT <select_list>
FROM Table_A A
LEFT JOIN Table_B B
ON KEYS <keys-clause(A)>



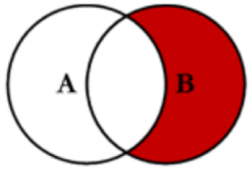
SELECT <select_list>
FROM Table_A A
INNER JOIN Table_B B
ON KEYS <keys-clause(A)>



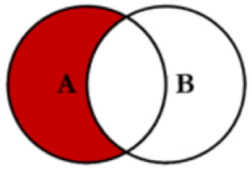
SELECT <select_list>
FROM Table_B B
LEFT JOIN Table_A A
ON KEYS <keys-clause(B)>



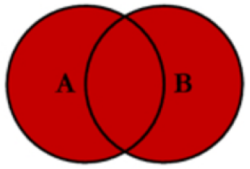
SELECT <select_list>
FROM Table_B B
LEFT JOIN Table_A A
ON KEYS <keys-clause(B)>
WHERE META(A**)**.id **IS MISSING**
UNION ALL
SELECT <select_list>
FROM Table_A A
LEFT JOIN Table_B B
ON KEYS <keys-clause(A)>
WHERE META(B**)**.id **IS MISSING**



SELECT <select_list>
FROM Table_B B
LEFT JOIN Table_A A
ON KEYS <keys-clause(B)>
WHERE META(B**)**.id **IS MISSING**



SELECT <select_list>
FROM Table_A A
LEFT JOIN Table_B B
ON KEYS <keys-clause(A)>
WHERE META(B**)**.id **IS MISSING**



SELECT <select_list>
FROM Table_B B
LEFT JOIN Table_A A
ON KEYS <keys-clause(B)>
UNION ALL
SELECT <select_list>
FROM Table_A A
LEFT JOIN Table_B B
ON KEYS <keys-clause(A)>
WHERE META(B**)**.id **IS MISSING**

@atom_yang, 2017

SELECT: Aggregation, Sorting

```
SELECT airlineid, COUNT(1) AS num_routes,
       SUM(distance) AS total_dist
FROM `travel-sample`
WHERE type = "route"
GROUP BY airlineid
HAVING count(1) > 10
ORDER BY num_routes DESC;
```

- Aggregation and sorting is expressed in a similar way like in SQL
- The typical aggregation functions supported: COUNT, SUM, AVG, MIN, MAX
- Efficient execution of aggregation and sorting requires appropriate secondary indexes

SELECT: Pagination

- Use OFFSET + LIMIT to paginate query results

```
SELECT sourceairport, destinationairport
FROM `travel-sample`
WHERE type = "route" AND airlineid = "airline_137"
LIMIT 10;
```

```
SELECT sourceairport, destinationairport
FROM `travel-sample`
WHERE type = "route" AND airlineid = "airline_137"
OFFSET 10 LIMIT 10;
```

SELECT: Combining Results, Query Nesting

- Results can be combine using various set operators: UNION, UNION ALL, INTERSECT, EXCEPT

```
SELECT DISTINCT sourceairport AS airport FROM `travel-sample` WHERE type = "route"
UNION
SELECT DISTINCT destinationairport AS airport FROM `travel-sample` WHERE type = "route"
```

- A subquery can be used wherever an array expression is expected

```
SELECT a.airport
FROM (SELECT DISTINCT sourceairport AS airport FROM `travel-sample` WHERE type = "route"
      UNION
      SELECT DISTINCT destinationairport AS airport FROM `travel-sample` WHERE type = "route") a
ORDER BY a.airport ASC
```

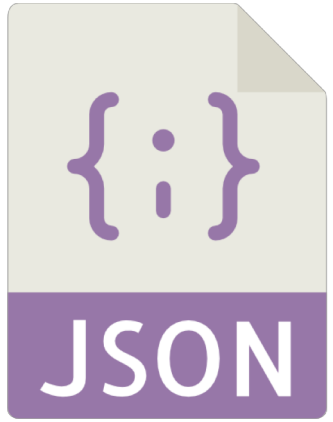
Data Modification Statements

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

INSERT INTO `airports` (KEY UUID(), VALUE ts)
    SELECT ts FROM `travel-sample` ts
      WHERE type = "airport";

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
```

JSON literals can be used in any expression

N1QL : Data Types from JSON

Data Type	Example
Numbers	`{ "id": 5, "balance":2942.59 }`
Strings	`{ "name": "Joe", "city": "Morrisville" }`
Boolean	`{ "premium": true, "balance pending": false}`
Null	`{ "last_address": Null }`
Array	`{ "hobbies": ["tennis", "skiing", "lego"]}`
Object	`{ "address": {"street": "1, Main street", "city": Morrisville, "state":"CA", "zip":"94824"}}`
MISSING	
Arrays of objects of arrays	`[ { "type": "visa", "cardnum": "5827-2842-2847-3909", "expiry": "2019-03" }, { "type": "master", "cardnum": "6274-2542-5847-3949", "expiry": "2018-12" } ]`



N1QL Data Type Handling

Non-JSON data types

- MISSING
- Binary

Data type handling

- Date functions for string and numeric encodings
- Total ordering across all data types
  - Well defined semantics for ORDER BY and comparison operators
- Defined expression semantics for all input data types
  - No type mismatch errors

N1QL Expressions for JSON

Ranging over collections	• WHERE **ANY** c **IN** children **SATISFIES** c.age > 10 **END** • WHERE **EVERY** r **IN** ratings **SATISFIES** r > 3 **END**
Mapping with filtering	• **ARRAY** c.name **FOR** c **IN** children **WHEN** c.age > 10 **END**
Deep traversal, SET, and UNSET	• WHERE **ANY** node **WITHIN** request **SATISFIES** node.type = “xyz” **END** • UPDATE doc **UNSET** c.field1 **FOR** c **WITHIN** doc **END**
Dynamic Construction	• SELECT **{ “a”: expr1, “b”: expr2 }** AS obj1, name FROM … // Dynamic object • SELECT **[a, b]** FROM … // Dynamic array
Nested traversal	• SELECT **x.y.z, a[0]** FROM **a.b.c** …
IS [NOT] MISSING	• WHERE name **IS MISSING**

N1QL Functions

Aggregate functions
ARRAY_AGG()
ARRAY_AGG(DISTINCT)
AVG()
AVG(DISTINCT)
COUNT()
COUNT(DISTINCT)
MAX()
MIN()
SUM()
SUM(DISTINCT)

Object functions
OBJECT_LENGTH()
OBJECT_NAMES()
OBJECT_PAIRS()
OBJECT_INNER_PAIRS()
OBJECT_VALUES()
OBJECT_INNER_VALUES()
OBJECT_ADD()
OBJECT_PUT()
OBJECT_REMOVE()
OBJECT_UNWRAP()

Array functions
ARRAY_APPEND()
ARRAY_AVG()
ARRAY_CONCAT()
ARRAY_CONTAINS()
ARRAY_COUNT()
ARRAY_DISTINCT()
ARRAY_IFNULL()
ARRAY_LENGTH()
ARRAY_MAX()
ARRAY_MIN()
ARRAY_POSITION()
ARRAY_PREPEND()
ARRAY_PUT()
ARRAY_RANGE()
ARRAY_REMOVE()
ARRAY_REPEAT()
ARRAY_REPLACE()
ARRAY_REVERSE()
ARRAY_SORT()
ARRAY_SUM()

Comparison functions
GREATEST()
LEAST()

Conditional functions - unknowns
IFMISSING()
IFMISSINGORNULL()
IFNULL()
MISSINGIF()
NULLIF()

Conditional functions - numbers
IFINF()
IFNAN()
IFNANORINF()
NANIF()
NEGINFIF()
POSINFIF()

Number functions
ABS()
ACOS()
ASIN()
ATAN()
ATAN2()
CEIL()
COS()
DEGREES()
E()
EXP()
LN()
LOG()
FLOOR()
PI()
POWER()
RADIANS()
RANDOM()
ROUND()
SIGN()
SIN()
SQRT()
TAN()
TRUNC()

N1QL Functions

Date functions
CLOCK_MILLIS()
CLOCK_STR()
DATE_ADD_MILLIS()
DATE_ADD_STR()
DATE_DIFF_MILLIS()
DATE_DIFF_STR()
DATE_PART_MILLIS()
DATE_PART_STR()
DATE_TRUNC_MILLIS()
DATE_TRUNC_STR()
STR_TO_MILLIS()
MILLIS_TO_STR()
MILLIS_TO_UTC()
MILLIS_TO_ZONE_NAME()
NOW_MILLIS()
NOW_STR()
STR_TO_MILLIS()
STR_TO_UTC()
STR_TO_ZONE_NAME()

JSON functions
DECODE_JSON()
ENCODE_JSON()
ENCODED_SIZE()
POLY_LENGTH()

Meta and UUID functions
BASE64()
BASE64_ENCODE()
BASE64_DECODE()
META()
UUID()

Pattern-matching functions
REG_CONTAINS()
REG_LIKE()
REG_POSITION()
REG_REPLACE()

String functions
CONTAINS()
INITCAP()
LENGTH()
LOWER()
LTRIM()
POSITION()
REPEAT()
REPLACE()
RTRIM()
SPLIT()
SUBSTR()
TITLE()
TRIM()
UPPER()

Type-Checking Functions
ISARRAY()
ISATOM()
ISBOOLEAN()
ISNUMBER()
ISOBJECT()
ISSTRING()
TYPE()

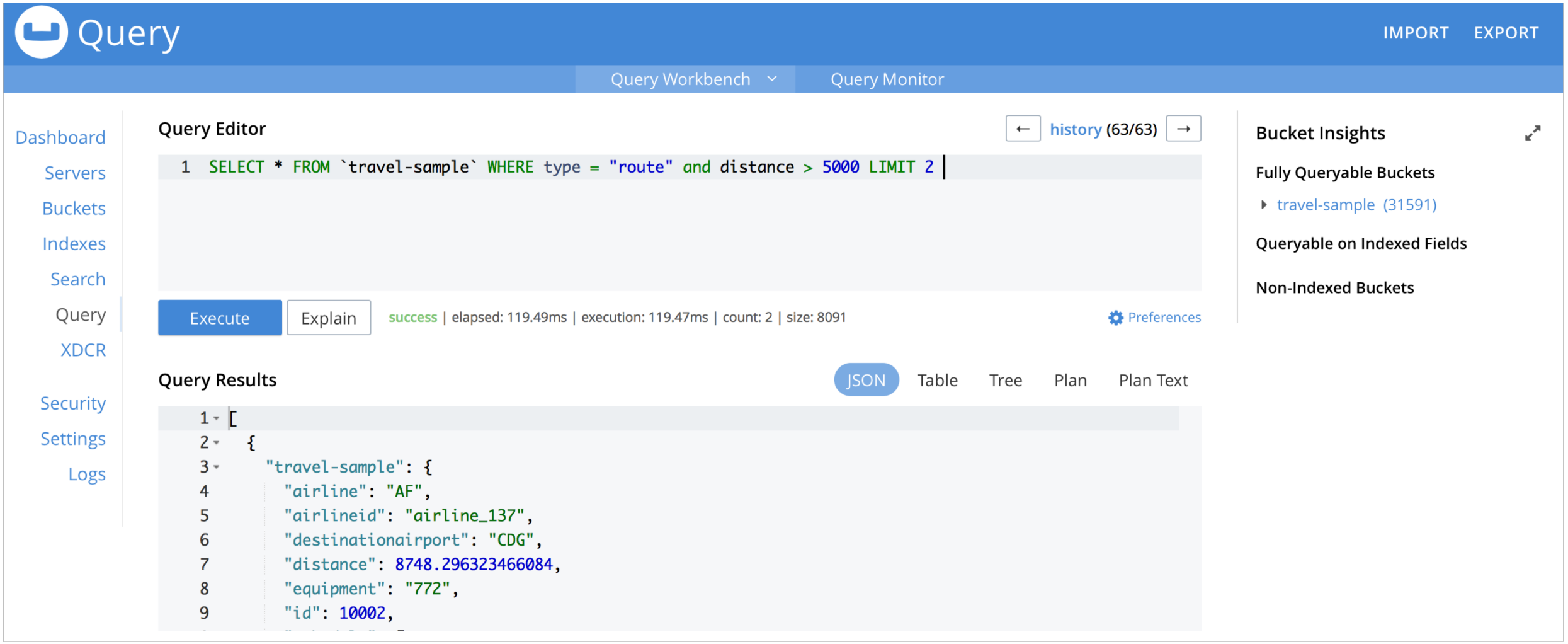
Type-Conversion Functions
TOARRAY()
TOATOM()
TOBOOLEAN()
TONUMBER()
TOOBJECT()
TOSTRING()

Summary: SQL & N1QL

Query Features	SQL	N1QL
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 ▪ Conversion Functions
Query Processing	▪ INPUT: Sets of Tuples ▪ OUPUT: Set of Tuples	▪ INPUT: Sets of JSON ▪ OUTPUT: Set of JSON

3 | Query Execution

Querying over the UI



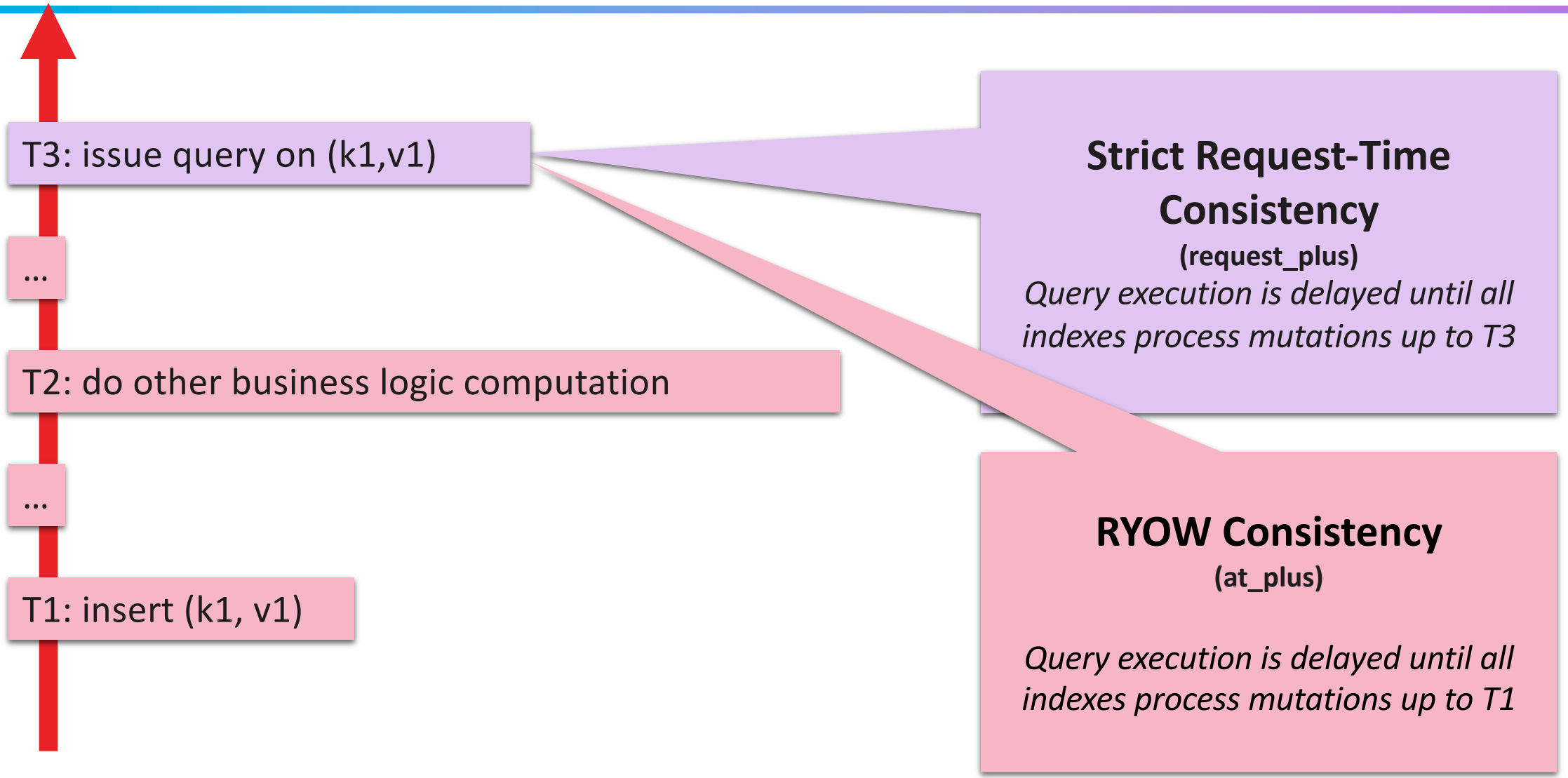
Querying from the Command Line: cbq

```
# cbq -e http://localhost:8093 -u Administrator -p password

cbq> SELECT * FROM `travel-sample` LIMIT 1;
{
    "requestID": "f699aa25-d477-4648-9bf5-4b3b20e37cd5",
    "signature": {
        "*": "*"
    },
    "results": [
        {
            "travel-sample": {
                "callsign": "MILE-AIR",
                "country": "United States",
                ...
cbq> \quit
```

Querying over REST API

```
# curl http://localhost:8093/query/service -u Administrator:password
  -d 'statement=SELECT * FROM `travel-sample` LIMIT 1'

{
"requestID": "4127aafd-32ef-455c-a7e0-4db5e1862df4",
"signature": {"*":"*"},
"results": [
{"travel-sample":{"callsign":"MILE-AIR","country":"United
States","iata":"Q5","icao":"MLA","id":10,"name":"40-Mile Air","type":"airline"}}
],
"status": "success",
"metrics": {"elapsedTime": "11.504349ms","executionTime": "11.462028ms","resultCount":
1,"resultSize": 138}
}
```

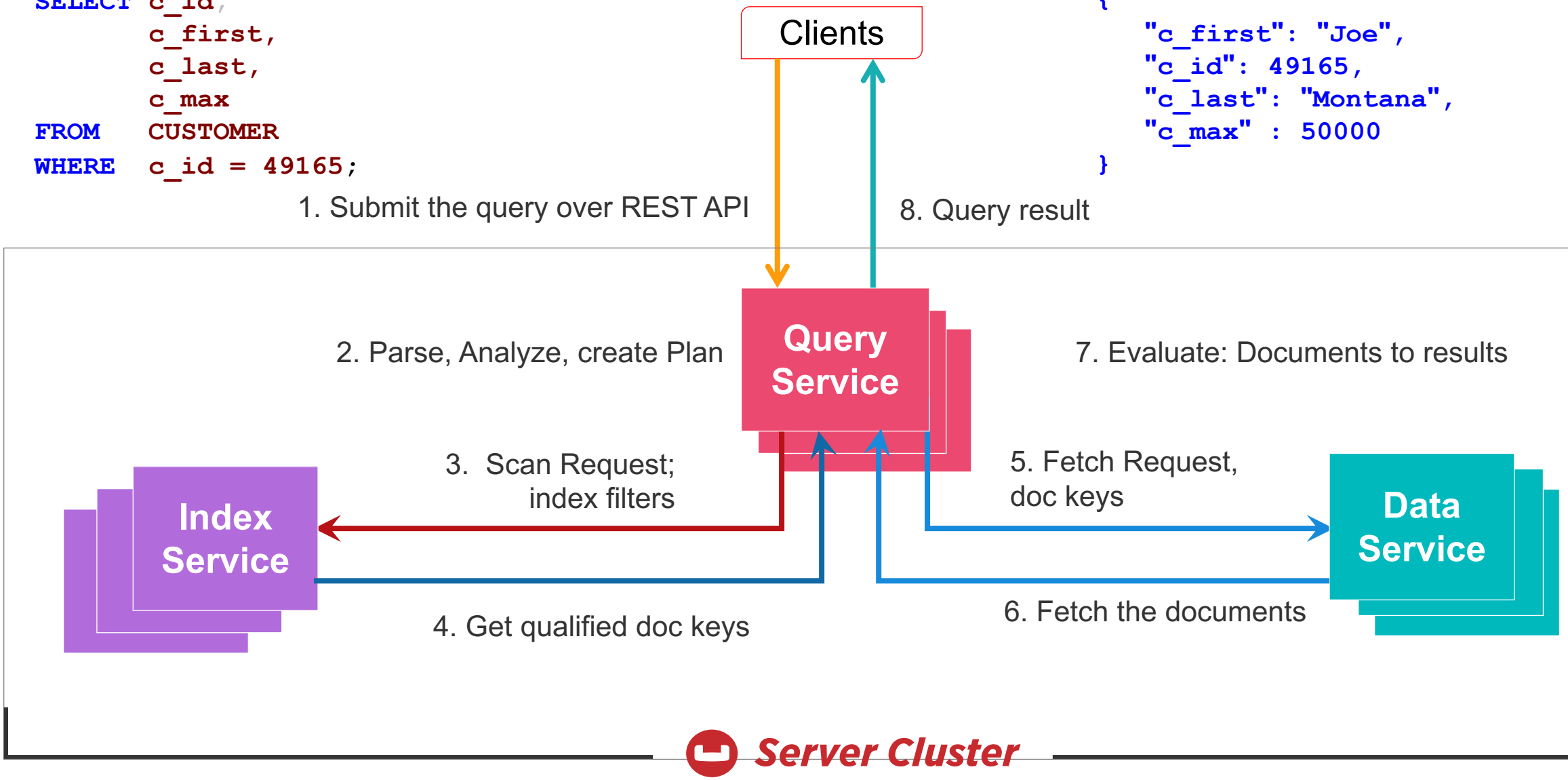
Query Consistency



N1QL: Query Execution Flow

```
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
}
```



4 | Index Types

Index Options

	Index Type	Description
1	Primary Index	Index on the document key on the whole bucket
2	Named Primary Index	Give name for the primary index. Allows multiple primary indexes in the cluster
3	Secondary Index	Index on the key-value or document-key
4	Secondary Composite Index	Index on more than one key-value
5	Functional Index	Index on function or expression on key-values
6	Array Index	Index individual elements of the arrays
7	Partial Index	Index subset of items in the bucket
8	Covering Index	Query able to answer using the the data from the index and skips retrieving the item.
9	Adaptive Index	Adaptive Indexes are a special type of GSI array index that can index all or specified fields of a document. Such an index is generic in nature, and it can efficiently index and lookup any of the index-key values

Primary Index

- The primary index is the index on the document key on every document in a keyspace
- The primary index is used for:
  - Complete keyspace scan – equivalent of table scan
  - Query does not have any predicates (filters)
  - Query has predicate on document key
  - No other index or access path can be used
- Primary index is optional

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


SELECT *
FROM `travel-sample`;

SELECT *
FROM `travel-sample`
WHERE META().id LIKE "user::%";
```

- **We *do not* recommend deploying a primary in index in a production environment**

Simple Secondary Index

- Index on any attribute or document-key
- Can use attribute within the document:
  - scalar, object, or array or expression
- Query WHERE clause must use index keys
  - Leading keys

```
CREATE INDEX ts_name ON `travel-sample`(name);
CREATE INDEX ts_city ON `travel-sample`(city);

SELECT *
FROM `travel-sample`
WHERE name = "United Airlines";

SELECT *
FROM `travel-sample`
WHERE city = "San Francisco";
```

Secondary Composite Index

- Index more than one expression
- Efficiently support queries with multiple predicates

```
CREATE INDEX ts_info ON `travel-sample` (type, id, name);

SELECT *
FROM `travel-sample`
WHERE type = "airline" AND id BETWEEN 0 AND 1000 AND name IS NOT NULL;

SELECT *
FROM `travel-sample`
WHERE type = "airline" AND id  < 100;

SELECT *
FROM `travel-sample`
WHERE type = "airline";
```

Functional Index

- Index any function or expression on the data, in addition to attributes
- Cannot contain aggregate expressions

```
CREATE INDEX ts_func ON `travel-sample`(UPPER(name), LENGTH(description));

SELECT *
FROM `travel-sample`
WHERE UPPER(name) = "MARRIOTT";

SELECT *
FROM `travel-sample`
WHERE UPPER(name) = "MARRIOTT" AND LENGTH(description) > 256;
```

Partial Index

- Index only a subset of the documents in a keyspace
- Create smaller, focused indexes
- Useful for keyspaces with multiple types of documents

```
CREATE INDEX ts_airline ON `travel-sample`(name) WHERE type="airline";
CREATE INDEX ts_hotels ON `travel-sample`(name) WHERE type = "hotel" AND country =
'United States';

SELECT *
FROM `travel-sample`
WHERE type = "airline" AND name = "United Airlines";

SELECT *
FROM `travel-sample`
WHERE type = "hotel" AND country = "United States” AND name LIKE "M%";
```

Partial Index

- Complex predicates in the WHERE clause of the index
  - Separate index for each state
  - Separate index for each year

```
CREATE INDEX ts_year ON `travel-sample`(orderid) WHERE SUBSTR(orderdate,0,4) = "2016";

SELECT *
FROM `travel-sample`
WHERE SUBSTR(orderdate,0,4) = "2016" AND orderid IS NOT NULL;
```

Partial Index

- Techniques
  - Partitioning into multiple indexes using ranges or hashing
  - Partitioning into multiple indexes onto distinct indexer nodes
  - Partitioning based on a list of values, e.g. an index for each state

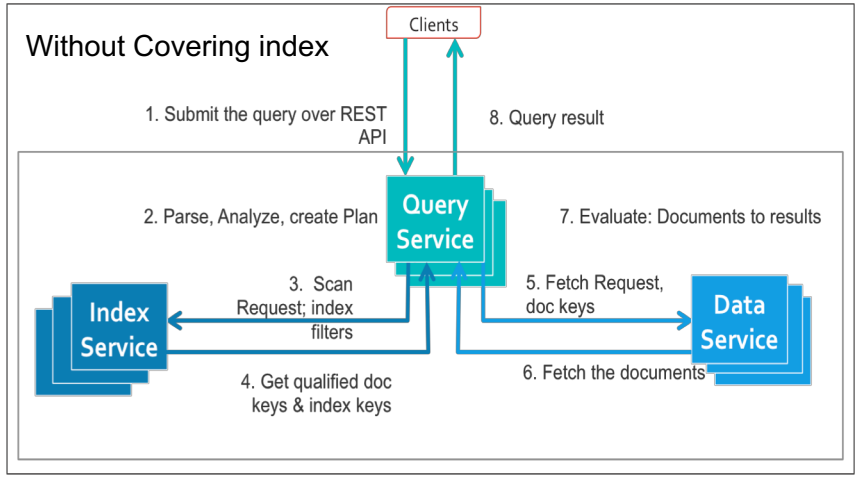
```
CREATE INDEX ts_airline ON `travel-sample`(name, id, icao, iata) WHERE type="airline”
AND icao like "A%";

SELECT *
FROM `travel-sample`
WHERE type = "airline" AND icao like "A%" AND name = "American" ;
```

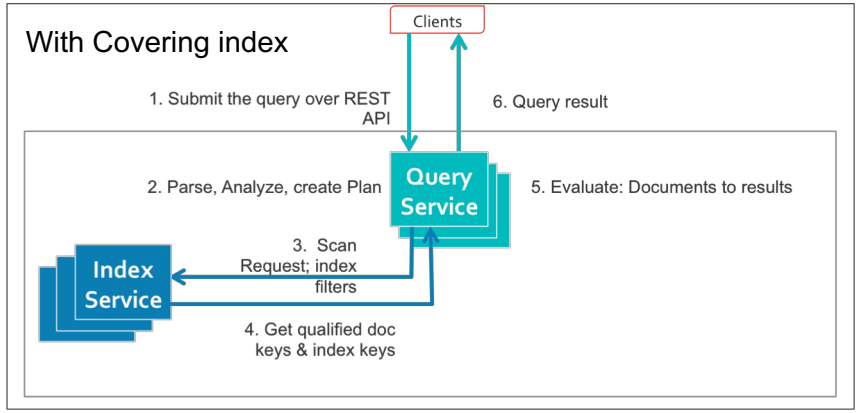
```
CREATE INDEX ts_year ON `travel-sample`(orderid) WHERE SUBSTR(orderdate,0,4) BETWEEN
2015 AND 2020;

SELECT *
FROM `travel-sample`
WHERE SUBSTR(orderdate,0,4) BETWEEN "2015" AND "2020" AND orderid IS NOT NULL;
```

Covering Index

- Index selection for a query solely depends on the query predicates
- Index keys cover predicates and all attribute references
- Avoids fetching the whole document
- Performance

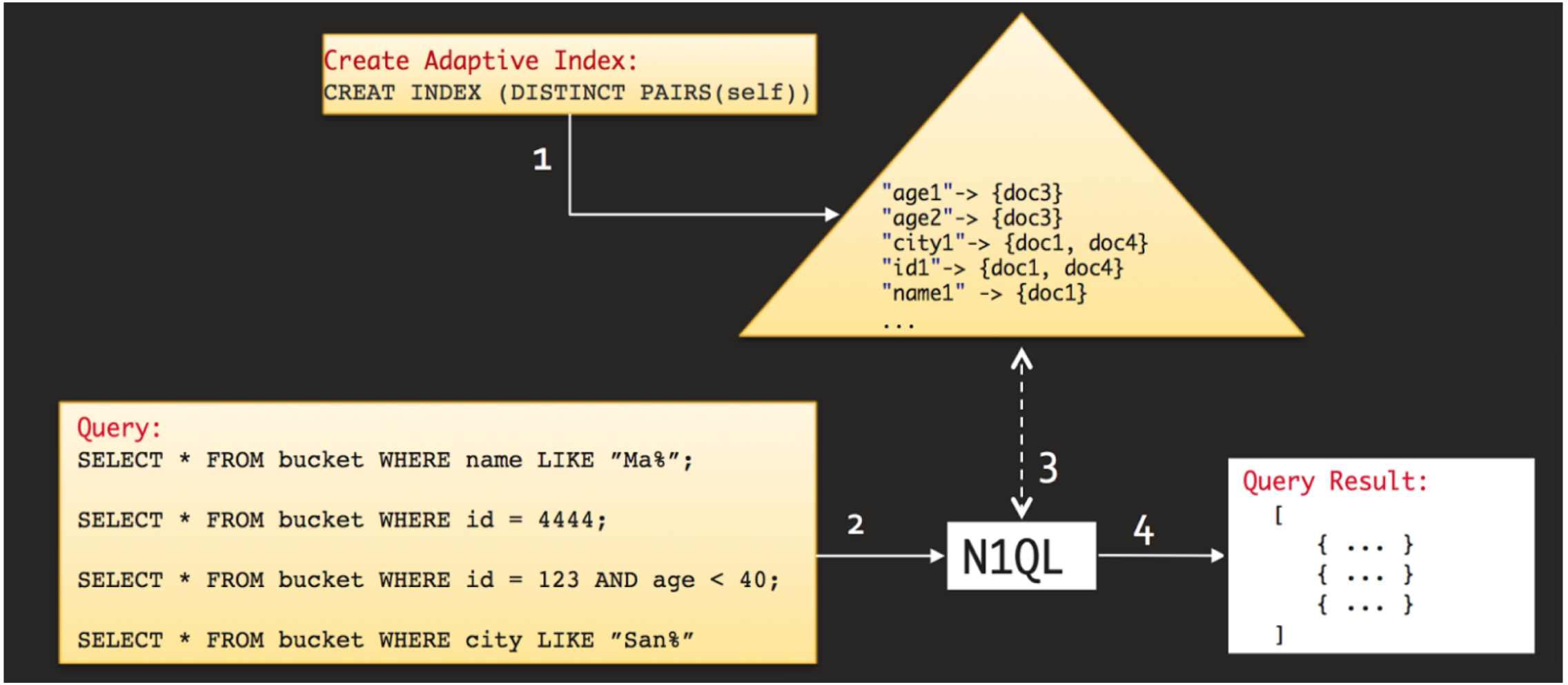
```
CREATE INDEX ts_airline ON `travel-sample`(name, id)
WHERE type = "airline";

SELECT name, id
FROM `travel-sample`
WHERE type = "airline" AND name = "United Airlines”;
```





Adaptive Index

Adaptive index covers all possible combination of fields, also the ones not known at design time



Using N1QL: Index Scan

- CREATE INDEX `idx_id` ON `travel-sample`(`id`);
- EXPLAIN SELECT meta().id FROM `travel-sample` WHERE id = 10;
- EXPLAIN SELECT meta().id FROM `travel-sample` WHERE id >=10 AND id < 25;
- EXPLAIN SELECT meta().id FROM `travel-sample` WHERE id IN [10, 20];

Predicate	Span Low	Span High	Inclusion
id = 10	10	10	3 (Both)
id >= 10	10	None	0 (Neither)
id <= 10	Null	10	2 (High)

5 | Quering Arrays

Querying Arrays

- Array Access
  - Expressions
  - Functions
  - Aggregates
- Statements
- Array Clauses

Array access : Expressions, Functions and Aggregates.

- **EXPRESSIONS**
- ARRAY
- ANY
- EVERY
- ANY AND EVERY
- IN
- WITHIN
- Construct [elem, elem]
- Slice [start:end]
- Elem subscript [#pos]

- **FUNCTIONS**
- ISARRAY
- TYPE
- ARRAY_APPEND
- ARRAY_CONCAT
- ARRAY_CONTAINS
- ARRAY_DISTINCT
- ARRAY_IFNULL
- ARRAY_FLATTEN
- ARRAY_INSERT
- ARRAY_INTERSECT
- ARRAY_LENGTH
- ARRAY_POSITION

- **FUNCTIONS**
- ARRAY_PREPEND
- ARRAY_PUT
- ARRAY_RANGE
- ARRAY_REMOVE
- ARRAY_REPEAT
- ARRAY_REPLACE
- ARRAY_REVERSE
- ARRAY_SORT
- ARRAY_STAR

- **AGGREGATES**
- ARRAY_AVG
- ARRAY_COUNT
- ARRAY_MIN
- ARRAY_MAX
- ARRAY_SUM

Array access

```
{
    "Name": "Jane Smith",
    "DOB" : "1990-01-30",
    "phones" : [
    "+1 510-523-3529", "+1 650-392-4923"
    ],
    "Billing": [
      {
        "type": "visa",
        "cardnum": "5827-2842-2847-3909",
        "expiry": "2019-03"
      },
      {
        "type": "master",
        "cardnum": "6274-2542-5847-3949",
        "expiry": "2018-12"
      }
    ]
}
```

```
SELECT phones from t;
[
  {
    "phones": [
      "+1 510-523-3529",
      "+1 650-392-4923"
    ]
  }
]
```

```
SELECT phones[1] from t;
[
  {
    "$1": "+1 650-392-4923"
  }
]
```

```
SELECT phones[0:1] from t;
[
  {
    "$1": [
      "+1 510-523-3529"
    ]
  }
]
```

Array access : Expressions and functions

```
{
    "Name": "Jane Smith",
    "DOB" : "1990-01-30",
    "phones" : [
    "+1 510-523-3529", "+1 650-392-4923"
    ],
    "Billing": [
      {
        "type": "visa",
        "cardnum": "5827-2842-2847-3909",
        "expiry": "2019-03"
      },
      {
        "type": "master",
        "cardnum": "6274-2542-5847-3949",
        "expiry": "2018-12"
      }
    ]
}
```

```
SELECT Billing[0].cardnum from t;
[
  {
    "cardnum": "5827-2842-2847-3909"
  }
]
```

```
SELECT Billing[*].cardnum from t;
[
  {
    "cardnum": [
      "5827-2842-2847-3909",
      "6274-2542-5847-3949"
    ]
  }
]
```

```
SELECT ISARRAY(Name) name, ISARRAY(phones)
phones from t;
[
  {
    "name": false,
    "phones": true
  }
]
```

Array access : Functions

```
{
    "Name": "Jane Smith",
    "DOB" : "1990-01-30",
    "phones" : [
    "+1 510-523-3529", "+1 650-392-4923"
    ],
    "Billing": [
      {
        "type": "visa",
        "cardnum": "5827-2842-2847-3909",
        "expiry": "2019-03"
      },
      {
        "type": "master",
        "cardnum": "6274-2542-5847-3949",
        "expiry": "2018-12"
      }
    ]
}
```

```
SELECT ARRAY_CONCAT(phones, ["+1 408-284-
2921"]) from t;
[
  {
    "$1": [
      "+1 510-523-3529",
      "+1 650-392-4923",
      "+1 408-284-2921"
    ]
  }
]
```

```
SELECT ARRAY_COUNT(Billing) billing,
       ARRAY_COUNT(phones) phones from t;
[
  {
    "billing": 2,
    "phones": 2
  }
]
```

Array access : Functions

```
SELECT phones, ARRAY_REVERSE(phones)
reverse from t;

  {
    "phones": [
      "+1 510-523-3529",
      "+1 650-392-4923"
    ],
    "reverse": [
      "+1 650-392-4923",
      "+1 510-523-3529"
    ]
  }
]
```

```
SELECT phones, ARRAY_COUNT(phones, 0, "+1 415-
439-4923") newlist from t;
[
  {
    "billing": 2,
    "phones": 2
  }
]
```

```
SELECT phones, ARRAY_INSERT(phones, 0, "+1 415-
439-4923") newlist from t;
[
  {
    "newlist": [
      "+1 415-439-4923",
      "+1 510-523-3529",
      "+1 650-392-4923"
    ],
    "phones": [
      "+1 510-523-3529",
      "+1 650-392-4923"
    ]
  }
]
```

Array access : Aggregates

```
SELECT ARRAY_MIN(Billing) AS minbill FROM
t;
[
  {
    "minbill": {
      "cardnum": "5827-2842-2847-3909",
      "expiry": "2019-03",
      "type": "visa"
    }
  }
]
```

```
SELECT name,
ARRAY_AVG(reviews[*].ratings[*].Overall) AS
avghotelrating
FROM `travel-sample`
WHERE type = 'hotel'
ORDER BY avghotelrating desc
LIMIT 3;

[
  {
    "avghotelrating": 5,
    "name": "Culloden House Hotel"
  },
  {
    "avghotelrating": 5,
    "name": "The Bulls Head"
  },
  {
    "avghotelrating": 5,
    "name": "La Pradella"
  }
]
```

Array predicates

- ANY
- EVERY
- ANY AND EVERY
- SATISFIES
- IN
- WITHIN
- WHEN

Array predicates

- **Arrays and Objects:** Arrays are compared element-wise. Objects are first compared by length; objects of equal length are compared pairwise, with the pairs sorted by name.
- **IN** clause: Use this when you want to evaluate based on specific field.
- **WITHIN** clause: Use this when you don't know which field contains the value you're looking for. The WITHIN operator evaluates to TRUE if the right-side value contains the left-side value as a child or descendant. The NOT WITHIN operator evaluates to TRUE if the right-side value does not contain the left-side value as a child or descendant.
- **EVERY**: EVERY is a range predicate that tests a Boolean condition over the elements or attributes of a collection, object, or objects. It uses the IN and WITHIN operators to range through the collection.

```
SELECT *
FROM `travel-sample`
WHERE type = 'hotel'
AND ANY r IN reviews
        SATISFIES r.ratings.`Value` >= 3
    END;
```

```
SELECT *
FROM `travel-sample`
WHERE type = 'hotel'
AND ANY r WITHIN reviews
        SATISFIES r LIKE '%Ozella%'
    END;
```

```
SELECT *
FROM `travel-sample`
WHERE type = 'hotel'
AND EVERY r IN reviews
     SATISFIES r.ratings.Cleanliness >= 4
    END;
```

Array predicates

- ARRAY_CONTAINS
  - Returns true if the array contains value.

```
SELECT name, t.public_likes
FROM `travel-sample` t
WHERE type="hotel" AND
ARRAY_CONTAINS(t.public_likes,
               "Vallie Ryan") = true;
[
  {
    "name": "Medway Youth
Hostel",
    "public_likes": [
      "Julius Tromp I",
      "Corrine Hilll",
      "Jaeden McKenzie",
      "Vallie Ryan",
      "Brian Kilback",
      "Lilian McLaughlin",
      "Ms. Moses Feeney",
```

Querying Arrays: UNNEST

- **UNNEST** : If a document or object contains an array, UNNEST performs a join of the nested array with its parent document. Each resulting joined object becomes an input to the query. UNNEST, JOINs can be chained.

```
SELECT r.author, COUNT(r.author) AS authcount
FROM `travel-sample` t UNNEST reviews r
WHERE t.type="hotel"
GROUP BY r.author
ORDER BY COUNT(r.author) DESC
LIMIT 5;
[
  {
    "authcount": 2,
    "author": "Anita Baumbach"
  },
  {
    "authcount": 2,
    "author": "Uriah Gutmann"
  },
  {
    "authcount": 2,
    "author": "Ashlee Champlin"
  },
  {
    "authcount": 2,
    "author": "Cassie O'Hara"
  },
  {
    "authcount": 1,
    "author": "Zoe Kshlerin"
  }
]
```

Querying Arrays: NEST

- **NEST** is the inverse of UNNEST.
- Nesting is conceptually the inverse of unnesting. Nesting performs a join across two keyspaces. But instead of producing a cross-product of the left and right inputs, a single result is produced for each left input, while the corresponding right inputs are collected into an array and nested as a single array-valued field in the result object.

```
SELECT *
FROM `travel-sample` route
NEST `travel-sample` airline
     ON KEYS route.airlineid
WHERE route.type = ‘airline' LIMIT 1;
```


```
[
  {
    "airline": [
      {
        "callsign": "AIRFRANS",
        "country": "France",
        "iata": "AF",
        "icao": "AFR",
        "id": 137,
        "name": "Air France",
        "type": "airline"
      }
    ],
    "route": {
      "airline": "AF",
      "airlineid": "airline_137",
      "destinationairport": "MRS",
      "distance": 2881.617376098415,
      "equipment": "320",
      "id": 10000,
      "schedule": [
        {
          "day": 0,
          "flight": "AF198",
          "utc": "10:13:00"
        },
        {
          "day": 0,
          "flight": "AF547",
          "utc": "19:14:00"
        },
        {
          "day": 0,
```

Array Index

- Index individual elements of an array
- Stored array or computed array (functional)
- Supports ANY, ANY AND EVERY, UNNEST queries

```
CREATE INDEX ts_sched ON `travel-sample`
         (DISTINCT ARRAY v.day FOR v IN schedule END);

SELECT *
FROM `travel-sample`
WHERE ANY v IN schedule SATISFIES v.day <= 2 END;

SELECT *
FROM `travel-sample`
WHERE ANY v IN schedule SATISFIES v.day IN [0,1] END;
```

Array Index

```
CREATE INDEX ts_sched ON `travel-sample`

         (DISTINCT ARRAY v.day FOR v IN schedule END);



SELECT *

FROM `travel-sample`

WHERE ANY AND EVERY v IN schedule SATISFIES v.day <= 2 END;



SELECT *

FROM `travel-sample`
```

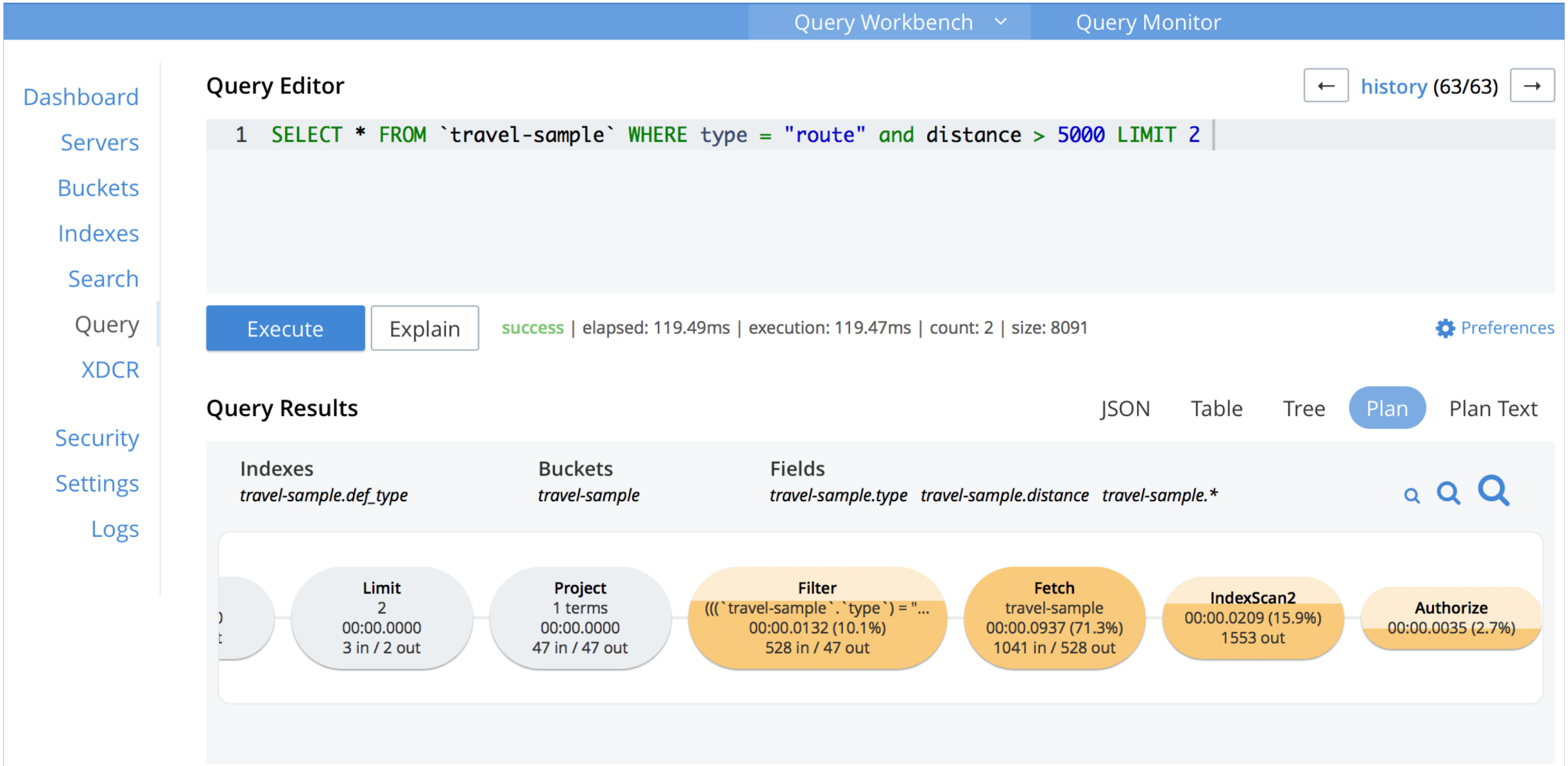
Array Index

- UNNEST using array indexing

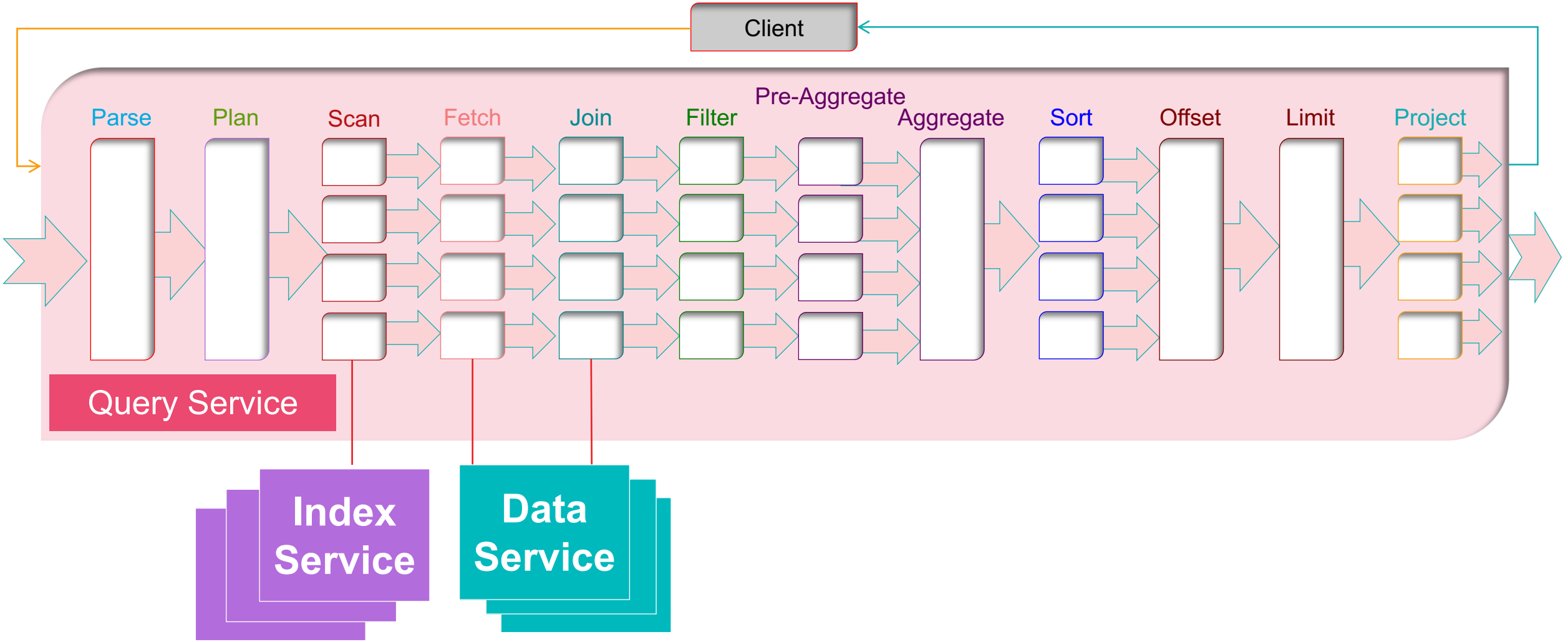
```
CREATE INDEX ts_sched ON `travel-sample`
         (DISTINCT ARRAY v.day FOR v IN schedule END);

SELECT *
FROM `travel-sample` UNNEST schedule AS v
WHERE v.day <= 2;
```

6 | Query Tuning

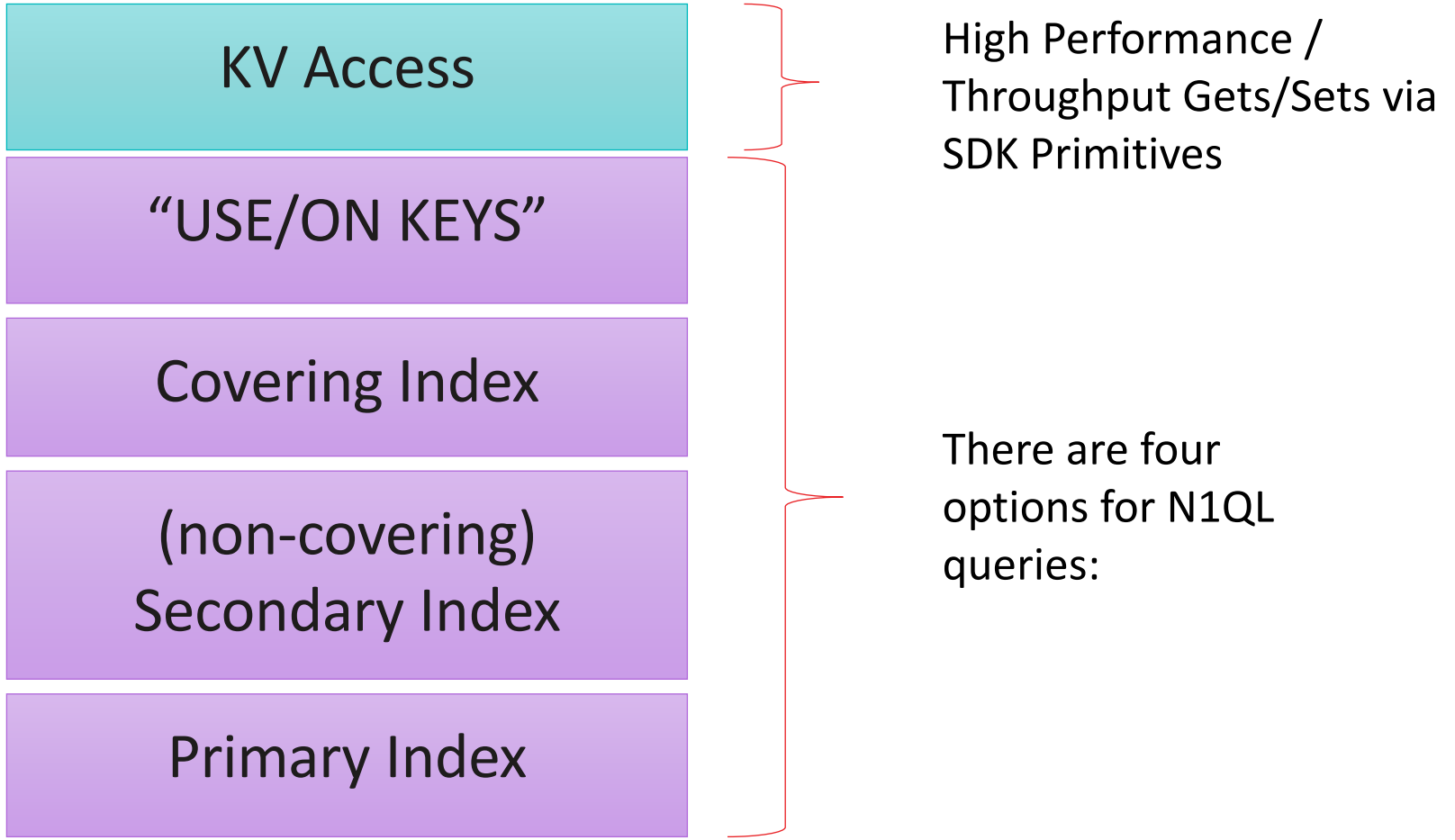
Exploring Query Plan



Inside the Query Service



Optimizing Query Speed



Query Tuning

- Rule based optimization
- Optimal indexes for optimal performance
- EXPLAIN to understand and tune
- Examine configuration for Data, Indexer, Query

Pattern “USE KEYS”

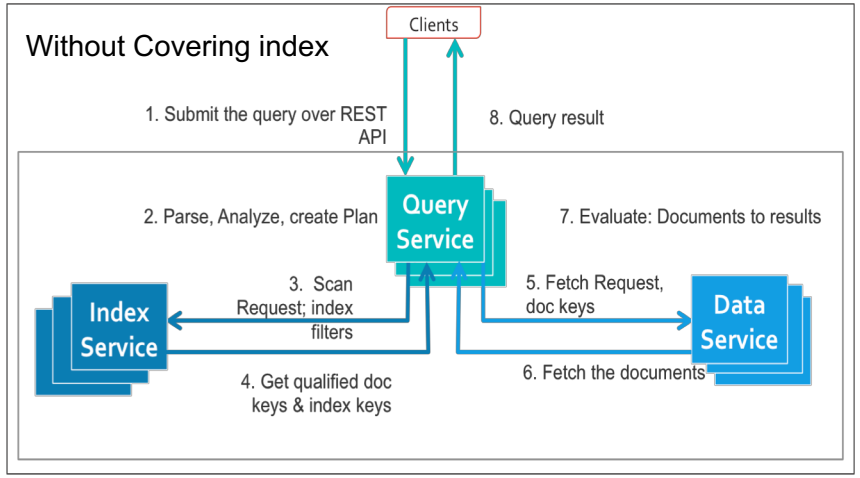
- USE KEYS provides facility Query Service to access Data Service directly
- No index required
- Scales independent of bucket size

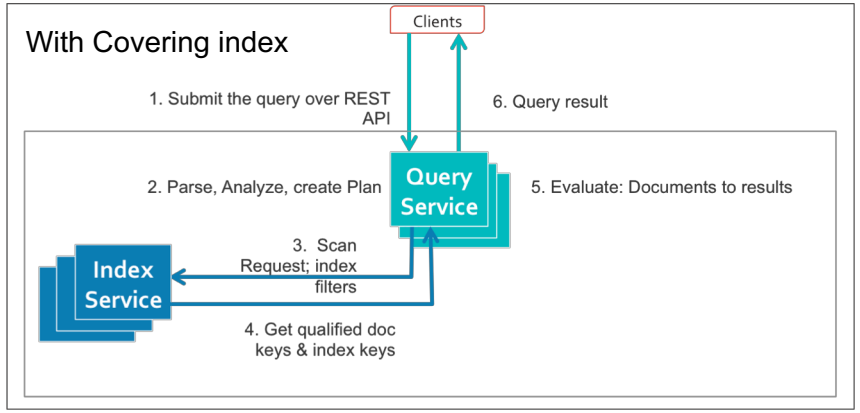
```
SELECT b.destinationairport, b.sourceairport,
  (SELECT c.name
    FROM `travel-sample` c
     USE KEYS b.airlineid
     LEFT UNNEST c.name) as theAirline
FROM `travel-sample` b
USE KEYS "route_5966"
```

Pattern “Covering Index”

- Index selection for a query solely depends on the query predicates
- Index keys cover predicates and all attribute references
- Avoids fetching the whole document
- Performance

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

SELECT name, id
FROM `travel-sample`
WHERE type = "airline" AND name = "United Airlines”;
```





Query Tuning Checklist

- Query should have predicates to avoid primary index scan.
- Explore all index options (composite secondary index, partial composite index, covered partial composite index ,…)
- Include as many predicate keys as possible in leading index keys
    - Query processing is bit more efficient when there is equality predicates on leading index keys
- Explore avoiding Intersect scan. If required provide hint with USE INDEX
- For ANY predicate clause, use ARRAY index keys.
- Explore using index key order and pushing limit, offset to indexer
- Rewrite query if required
- Use array fetch when possible
- Execute query and look the monitoring stats for each phase of query (ex: system:completed_requests) and tune it.
- Check the final query plan through the explain
- Set pretty=false in Query Service or query parameter
- Increase parallel processing via max_parallelism
- Fetching large set of data for non covered index increase pipeline-cap, pipeline-batch in Query-Service
- Duplicate indexes and memory optimized indexes
- Add more Query nodes or Indexer nodes

7 | Monitoring

Query Service REST API Endpoints

- Endpoints:
  - http://<NodeIP>:8093/admin/vitals : Overall heath picture of the query service
  - http://<NodeIP>:8093/admin/active_requests
  - http://<NodeIP>:8093/admin/completed_requests
  - http://<NodeIP>:8093/admin/prepareds

Query Monitoring: Active requests

- List / Delete requests currently being run by the query service
- Through N1QL
  - SELECT * FROM system:active_requests
  - DELETE FROM system:active_requests WHERE…
- Through REST
  - GET http://localhost:8093/admin/active_requests
  - GET http://localhost:8093/admin/active_requests/<request_id>
  - DELETE http://localhost:8093/admin/active_requests/<request_id>

Query Monitoring: Prepared Statements

- List / Delete requests prepared on the query node
- Through N1QL
  - SELECT * FROM system:prepareds
  - DELETE FROM system:prepareds WHERE…
- Through REST
  - GET http://localhost:8093/admin/prepareds
  - GET http://localhost:8093/admin/prepareds/<request_id>
  - DELETE http://localhost:8093/admin/prepareds/<request_id>

Query Monitoring: Completed Requests

- List / Delete completed requests deemed to be of high cost
- Through N1QL
  - SELECT * FROM system:completed_requests
  - DELETE FROM system:completed_requests where…
- Through REST
  - GET http://localhost:8093/admin/completed_requests
  - GET http://localhost:8093/admin/completed_requests/<request_id>
  - DELETE http://localhost:8093/admin/completed_requests/<request_id>

Completed Requests: Cache Configuration

- Change completed requests settings, per node
  - POST http://localhost:8093/admin/settings

  Settings passed as data, in json format
- Requires authorization
- Available completed requests settings
  - “completed-limit”, cache size, integer, from zero upwards
  - “completed-threshold”, minimum time duration in milliseconds, -1 (disabled) upwards
- Example
  - `Curl –X POST http://localhost:8093/admin/prepareds/<statement id> -u Administrator:password –d ‘{ “completed-limit”: 10000 }’`

Query Analysis Using N1QL

- N1QL can be by itself used as a powerful tool to analyze the completed requests
- e.g., run the following query to get 10 longest running queries

```
SELECT elapsedTime, statement
FROM system:completed_requests
ORDER BY str_to_duration(elapsedTime)
DESC LIMIT 10;
```

Plans using suboptimal indexes

- `system:active_requests` and `system:completed_requests` report among other things processed documents per phase
- This can be used to determine
  - Requests using primary scans

```
select * from system:completed_requests where phaseCounts.`primaryScan` is not missing
```

  - Requests not using a covering index

```
select * from system:completed_requests where phaseCounts.`indexScan` is not missing and phaseCounts.`fetch` is not missing
```

  - Requests using an index which is not very selective

```
select * from system:completed_requests where phaseCounts.`indexScan` is not missing and phaseCounts.`fetch` / resultCount > 10
```

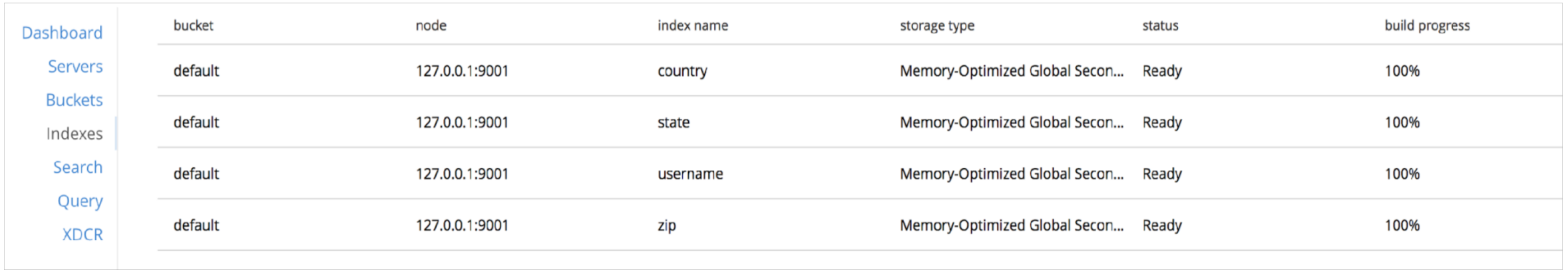
  - Etc!
- Order results by `phaseCounts.`fetch`` or `resultCount`
- Same queries can be done against `system:active_requests`

Cancelling unruly requests

- `DELETE FROM system:active_requests WHERE RequestId='…'`
- Must be run on the node the request is running on

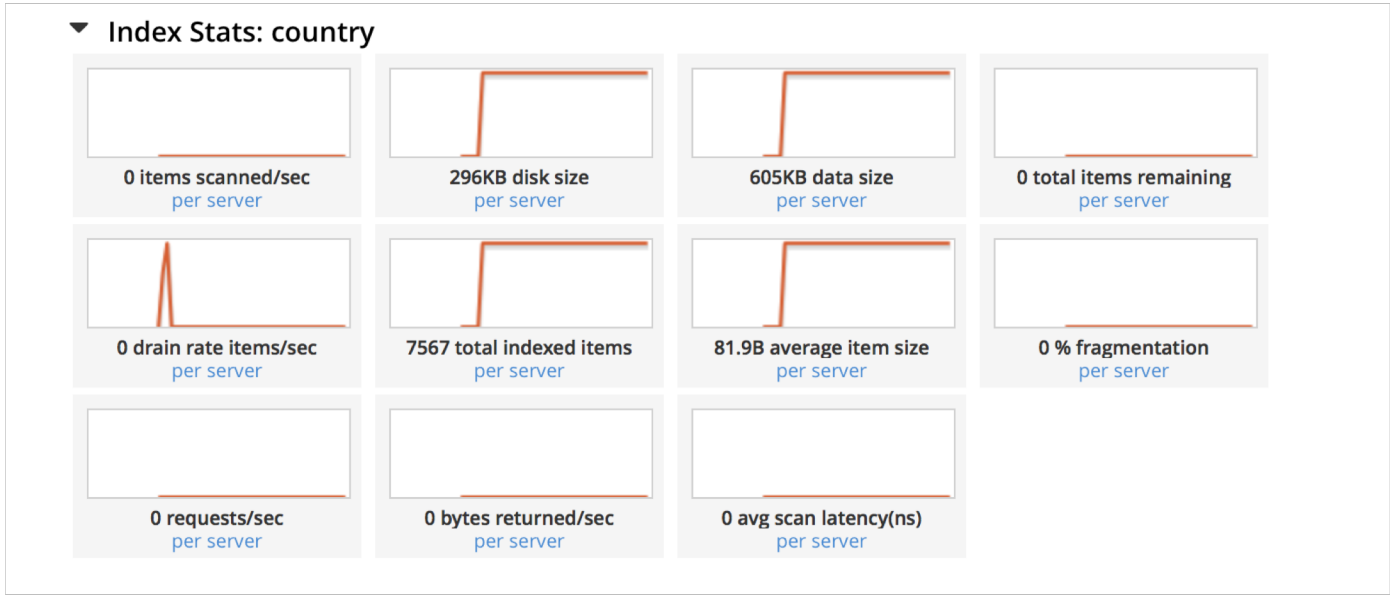
Index Status

- Show the status of each index
  - Created – Created but not yet built
  - Building – Index is being built
  - Ready – Index built is finished, ready for query



Index Statistics

- Statistics of each index is displayed under bucket statistics



Index Statistics

- Disk Size
  - Total bytes on disk (including fragmentation)
- Data Size
  - Total bytes used by index
- % Fragmentation
  - Standard index only
  - Percentage of obsolete data to be reclaimed
- Total Indexed Items
  - Total number secondary keys
- Average Item Size
  - Average size of secondary key
- Total Items Remaining
  - Estimated number of pending documents remained to be processed
- Drain Rate items/sec
  - Average update throughput in terms of # of documents
- Request/sec
  - Scan throughput in terms of # of requests
- Bytes returned/sec
  - Scan throughput in terms of bytes
- Item Scanned/sec
  - Scan throughput in terms of # of returned keys/rows
- Avg Scan Latency
  - Average scan latency

Indexer Stats

- Indexer collects a lot of stats related to its progress, performance of storage, scan timing, various other important operational aspects.
- Indexer Stats are published at runtime on REST endpoint

```
http://<NodeIP>:9102/stats
```

- Indexer Stats are periodically logged at INFO log level as a JSON.

8 Index Administration

Deferred Index Build

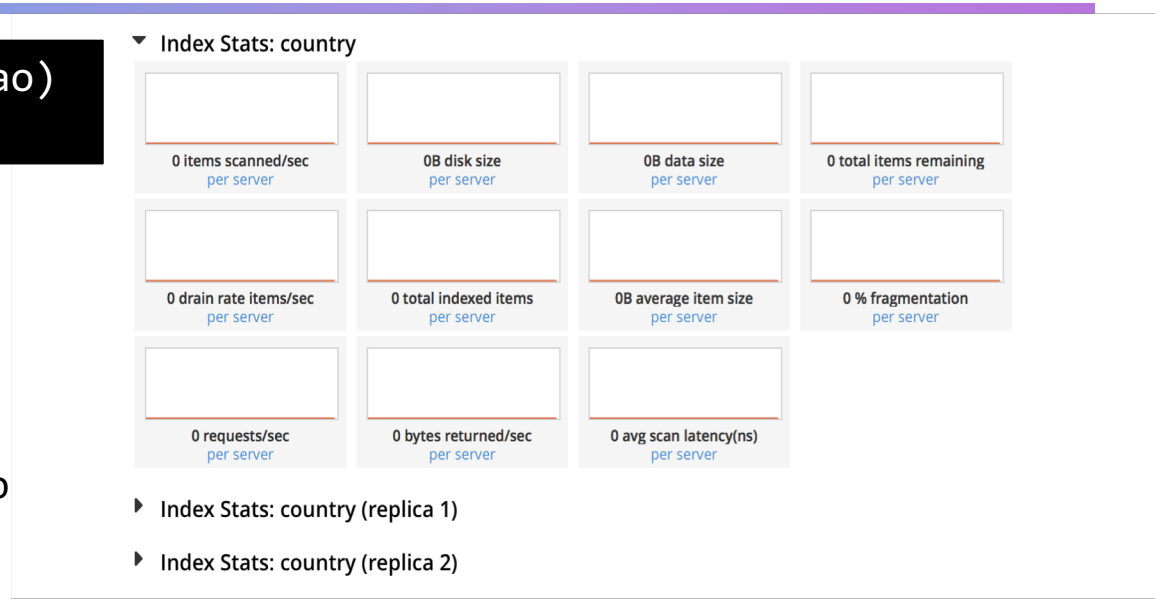
- Defer builds offers a two stage process of creating indexes
- It is recommended that Defer Builds be put to optimum use always, as the same change feed is used to create indexes on a node
- If DEFER builds are not used, the change feed from the data nodes has to be accessed multiple times, leading to more data transfer across the network and a slightly increased load on the data nodes

```
CREATE INDEX `idx_ts_type_iata`
ON `travel-sample`(`type`,`iata`) WITH { "defer_build":true };

CREATE INDEX `idx_ts_type_icao`
ON `travel-sample`(`type`,`icao`) WITH { "defer_build":true };

BUILD INDEX ON `travel-sample`(`idx_ts_type_iata`, `idx_ts_type_icao`);
```

Index Replica Management – Auto Placement

```
CREATE INDEX ts_di ON `travel-sample`(LOWER(name), id, icao)
WHERE type = "airline" WITH {"num_replica":2}
```

- Creates 2 additional replica (total 3 copies)
- Couchbase picks index nodes for initial placement
  - Rule 1: Do not place replica onto the same node
  - Rule 2: Spread out replica onto as many server groups as possible
  - Rule 3: Pick the nodes that have fewest number of indexes (room fo improvement in the future)
- As index is created, index build will run concurrently for each replica
- Index can be used for query as soon as any replica becomes online



bucket	node	index name	storage type	status	build progress
default	127.0.0.1:9001	country	Memory-Optimized Global Secon...	Ready	100%
default	127.0.0.1:9005	country (replica 1)	Memory-Optimized Global Secon...	Ready	100%
default	127.0.0.1:9004	country (replica 2)	Memory-Optimized Global Secon...	Ready	100%

Index Replica Management – Rebalance

- When an index node is removed from the cluster, its **indexes will be redistributed** to the remaining nodes automatically.
- Adding a new index node to the cluster **will not cause the indexes to be re-distributed**. This is by design to prevent excess movement of indexes and also to preserve the placement that an administrator may have specified. (may change in future if customer demand shows value).
- If an index node fails and is replaced, the appropriate **replicas will be re-created**. Similarly, those index replicas are re-created on the remaining nodes of a cluster if a rebalance is performed without replacing the failed node.

Index Replica Management – Manual Placement

- What if you want to take back control and do manual placement for specific reasons?
- Can specify a list of nodes at index creation time:

```
CREATE INDEX ts_di ON `travel-sample`(LOWER(name), id, icao)
WITH {“nodes”:[“17.2.33.101:8091”, “127.2.33.102:8091”, “127.2.33.103:8091”]}
```

Failover and Delta Recovery

- Use Index Replica to ensure index remains online for query
- When index node recovers, it will recover both metadata and index data
  - Index will catch up to latest mutations
  - If an index has been dropped when index node is offline, it will be dropped upon recovery
  - If index build is executed when index node is offline, index build will start upon recovery

Index Rebalancing Rules

- Redistribute indexes from ejected nodes to available nodes in cluster
- Will not redistribute index residing on non-ejected nodes
  - Preserve original layout as much as possible
  - Minimize index movement

- Rebalancing of replicas
  - Does not place replica onto the same node
  - Spreads out replica onto as many server groups as possible
  - If there are more replica than available nodes, rebalancing can drop replica
  - Can repair lost replica when new node is added in next rebalance

Manual Rebalancing of Indexes

- ALTER INDEX statement allows moving an index to a different node:

```
ALTER INDEX `travel-sample`.def_icao
WITH {"action":"move", "nodes": ["10.112.190.102:8091"]}
```

Indexes & Partitioning

- Why Partition Indexes?
  - Index may not fit on the node
    - Fit index to the nodes you have
    - Distribute Index load to many nodes
  - Minimize the tree scanned
    - Large/Tall indexes will take longer to scan

Index Partitioning (since 5.5)

- Index can be partitioned by a hash value of an arbitrary attribute

```
CREATE INDEX ix_route ON `travel-sample` (airline, flight, source_airport, destination_airport)
PARTITION BY HASH(airline);
```

- Exact lookups by the secondary key is are always performed on one node
- Partitioning by the first attribute of an index enables range scans of all other attributes within one node
- Partitioning key must be selected carefully
  - Partitioning key on small cardinality attributes (e.g. country) can produce a significant skew
  - Partitioning by meta().id, would produce a very even distribution, but all range lookups would fall back to scatter-and-gather

https://blog.couchbase.com/index-partitioning-couchbase-server/

Backup / Restore

- cbbackup/cbrestore and cbbackupmgr
- Only Index Metadata is backed up
- Restore to the same cluster topology will result in same index layout
- Restore to different cluster topology will result in a different index layout

Thank you

Couchbase