Architecture and Administration Basics

Workshop Day 2 – N1QL “Nickel”

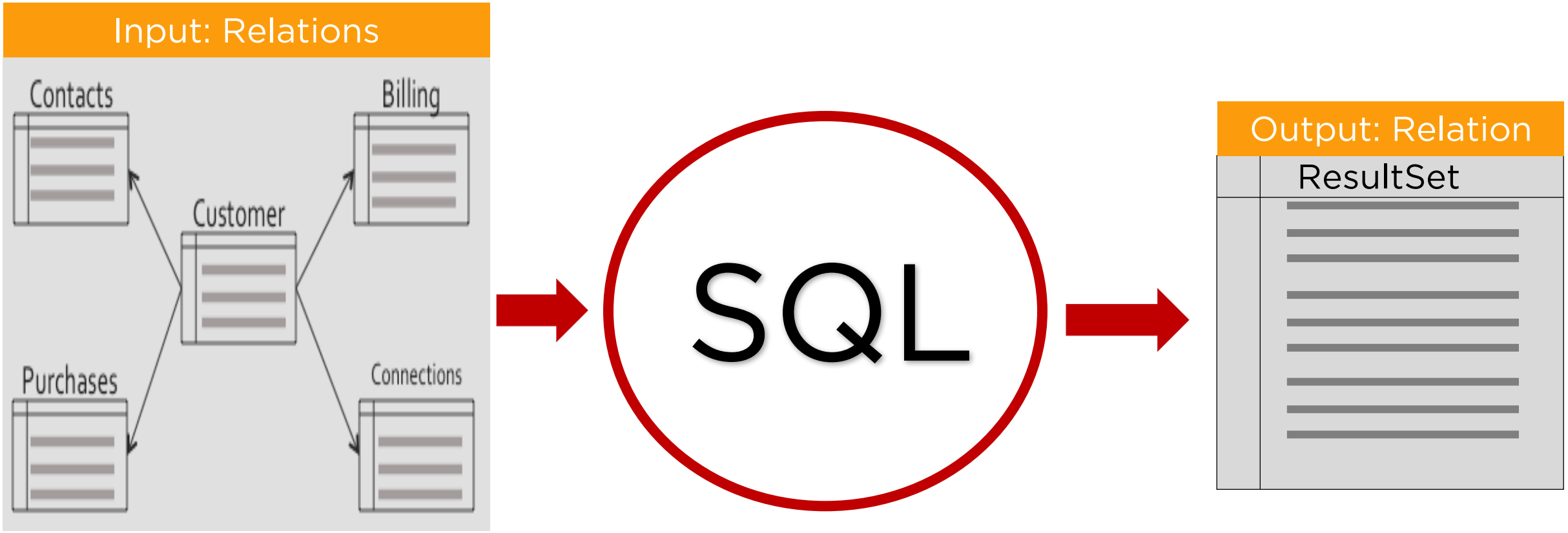
What is N1QL?

- **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

SQL


Input: Relations
Contacts
Billing
Customer
Purchases
Connections
SQL
Output: Relation
ResultSet

N1QL is Declarative: What Vs How

You specify **WHAT**
Couchbase Server figures out **HOW**

Input: JSON

LoyaltyInfo

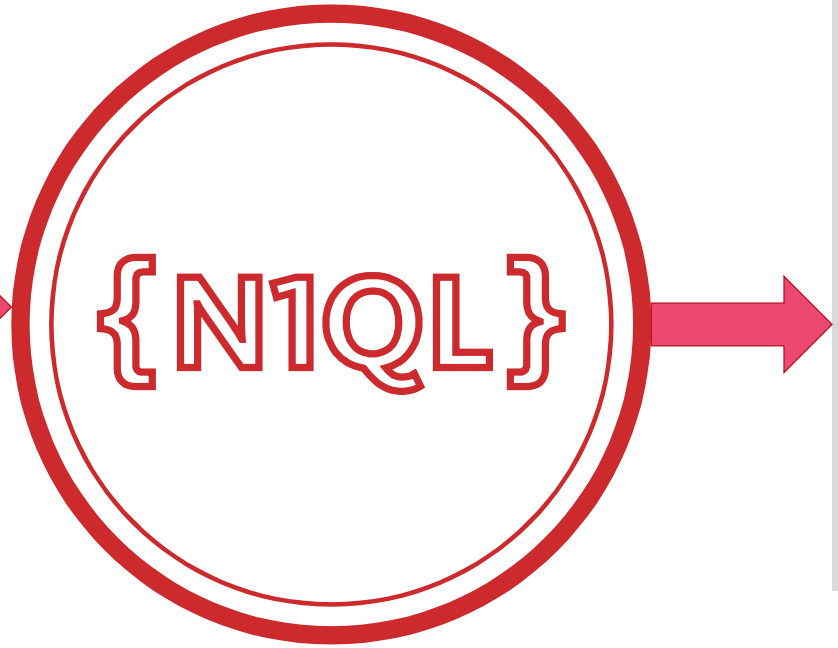
```
[
  "Name" : "Jane Smith",
  "DOB"  : "1990-01-30",
  "Billing" : [
     {
        "type"   : "visa",
        "cardnum" : "5827-2842-2847-3909",
        "expiry" : "2019-03"
     },
     {
        "type"   : "master",
        "cardnum" : "6274-2842-2847-3909",
        "expiry" : "2019-03"
     }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

Customer


```
{
  "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 }
  ]
}
```


Orders


```
{
  "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 }
  ]
}
```

Output: JSON

Result Documents

```
{
   "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 }
   ]
}
```

Motivation for N1QL (SQL for JSON)

- NoSQL databases provide excellent performance and scalability
- Toolset for working with data in aggregates
- Before N1QL, NoSQL tends to come with some unexpected baggage
- Map-Reduce for processing is great for some types of processing, but is not expressive enough for other kinds
- Some NoSQL solutions say “but no joins” or have odd ways of duplicating data to make it fit the query model

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

N1QL (EXPRESSIVE)

- Access to every part of JSON document
- Scalar & Aggregate functions
- Issue subquery in any expressions
- Subqueries
- Subqueries in the FROM clause

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

N1QL (POWERFUL)

- Access to every part of JSON document
- JOINS, Aggregations, standard scalar functions
- Aggregation on arrays
- NEST & UNNEST operations
- Covering Index

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

N1QL (QUERYING)

- INSERT
- UPDATE
- DELETE
- MERGE
- SELECT
- EXPLAIN

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

N1QL (TRANSFORMING & MANIPULATING)

- Full Transformation of the data via Query.
- INSERT
- INSERT single & multiple documents
- INSERT result a SELECT statement
- DELETE documents based on complex filter
- UPDATE any part of JSON document & use complex filter.
- MERGE two sets of documents using traditional MERGE statement

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

N1QL (EXPRESSIVE)

- Access to every part of JSON document
- Scalar & Aggregate functions
- Subqueries in the FROM clause
- Aggregation on arrays

Expressive, familiar, and feature-rich language for querying, transforming, and manipulating JSON data

N1QL (FAMILIAR)

- SELECT * FROM bucket WHERE ...
- INSERT single & multiple documents
- UPDATE any part of JSON document & use complex filter
- DELETE
- MERGE two sets of documents using traditional MERGE statement
- EXPLAIN to understand the query plan
  - EXPLAIN SELECT * FROM bucket WHERE ...

Expressive, familiar, and feature-rich language for querying, transforming, and manipulating JSON data

N1QL (FEATURE-RICH)

- Access to every part of JSON document
- Functions (Date, Pattern, Array, Conditional, etc)
  - https://developer.couchbase.com/documentation/server/current/n1ql/n1ql-language-reference/functions.html
- JOIN, NEST, UNNEST
- Covering Index
- Prepared Statements
- USE KEYS, LIKE

Expressive, familiar, and feature-rich language for querying, transforming, and manipulating JSON data

N1QL (Example)

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

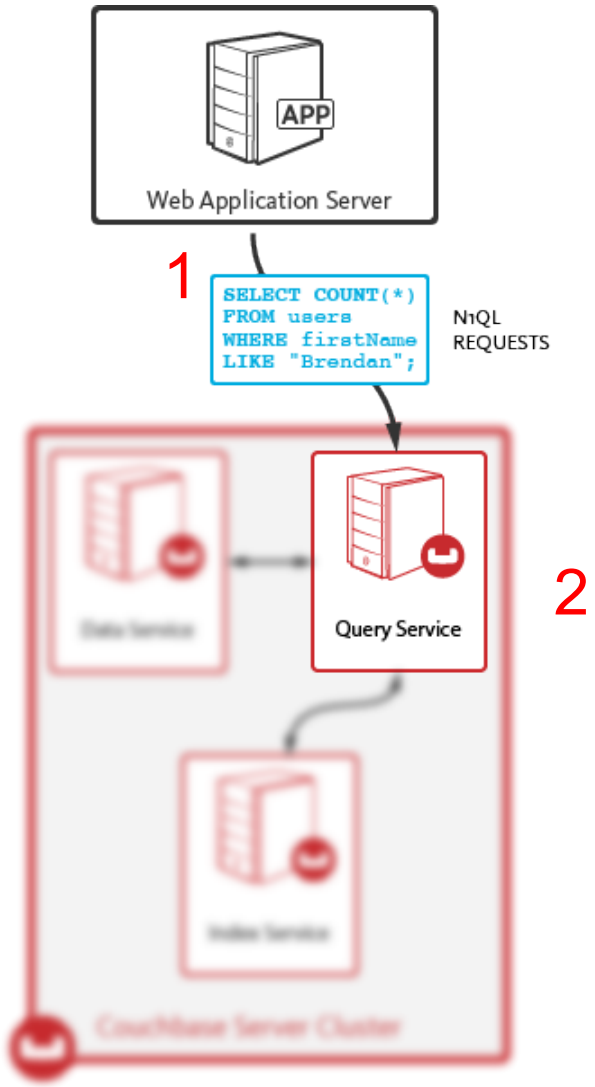
Dotted sub-document reference
Names are CASE-SENSITIVE

UNNEST to flatten the arrays

JOINS with Document KEY of customers

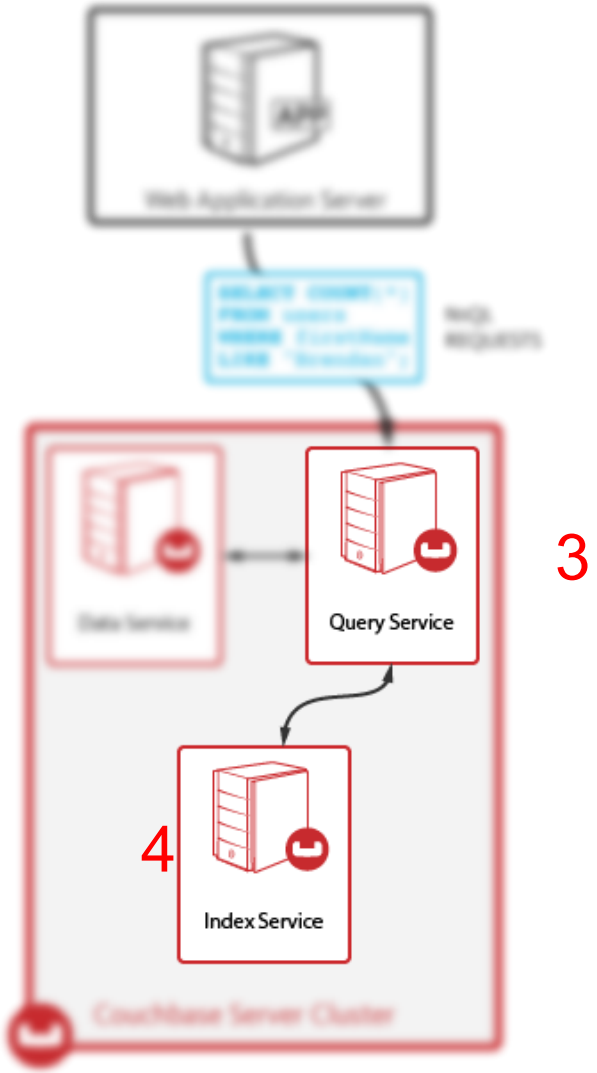
N1QL Query : Execution

Query Execution Flow



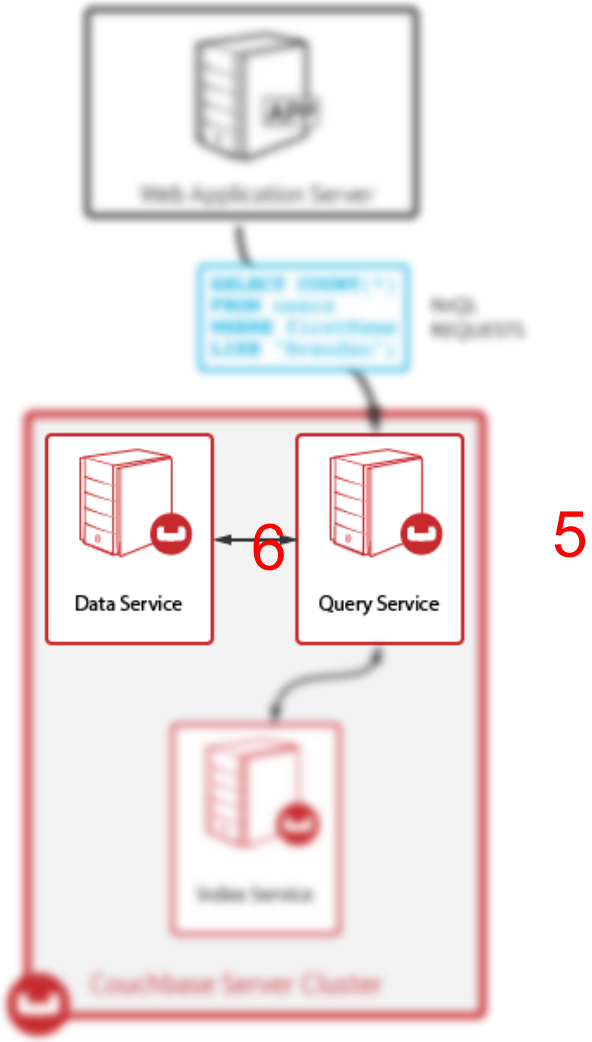
1. Application submits N1QL query
2. Query is parsed, analyzed and plan is created

Query Execution Flow



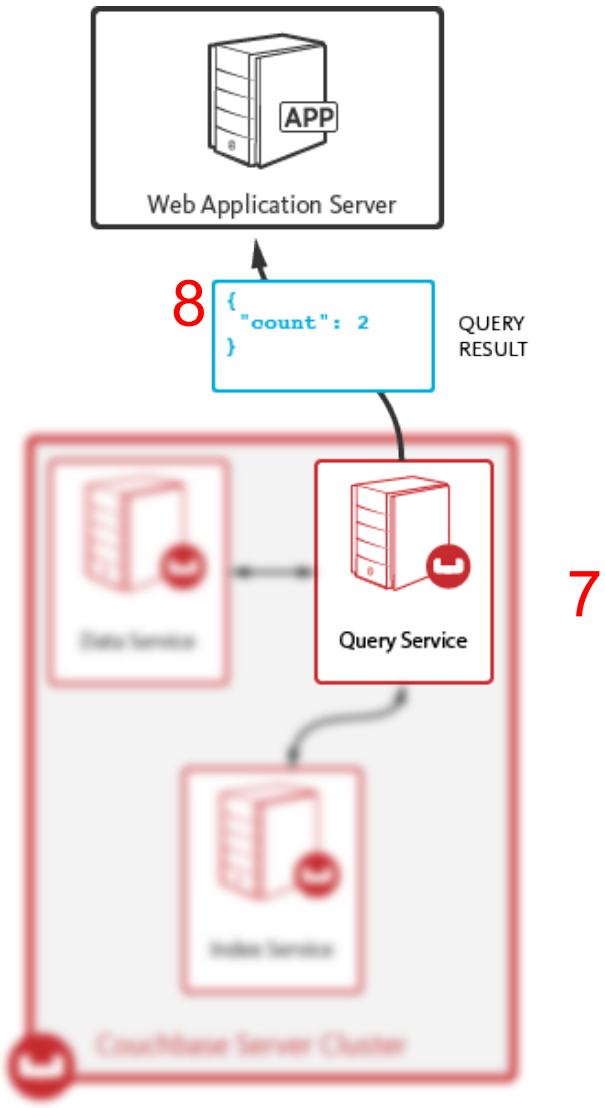
3. Query Service makes request to Index Service
4. Index Service returns document keys and data

Query Execution Flow



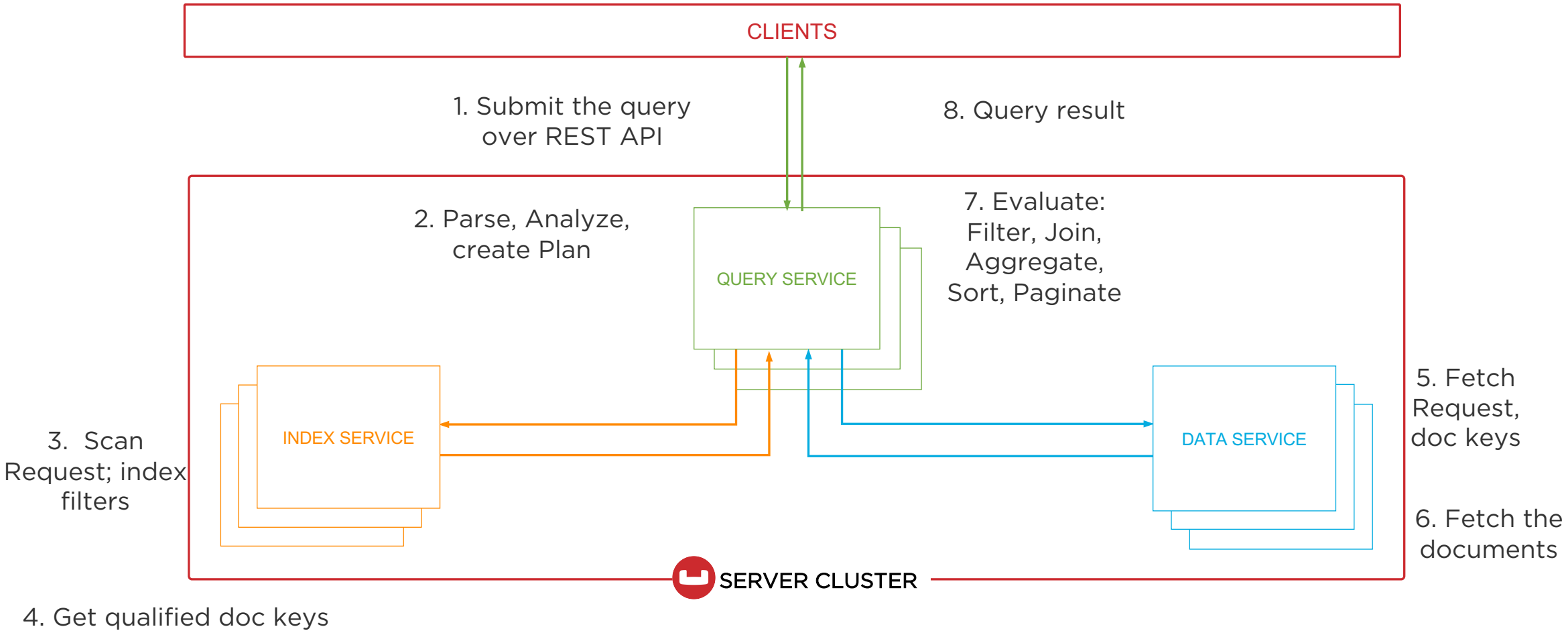
5. If Covering Index, skip step 6
6. If filtering is required, fetch documents from Data Service

Query Execution Flow

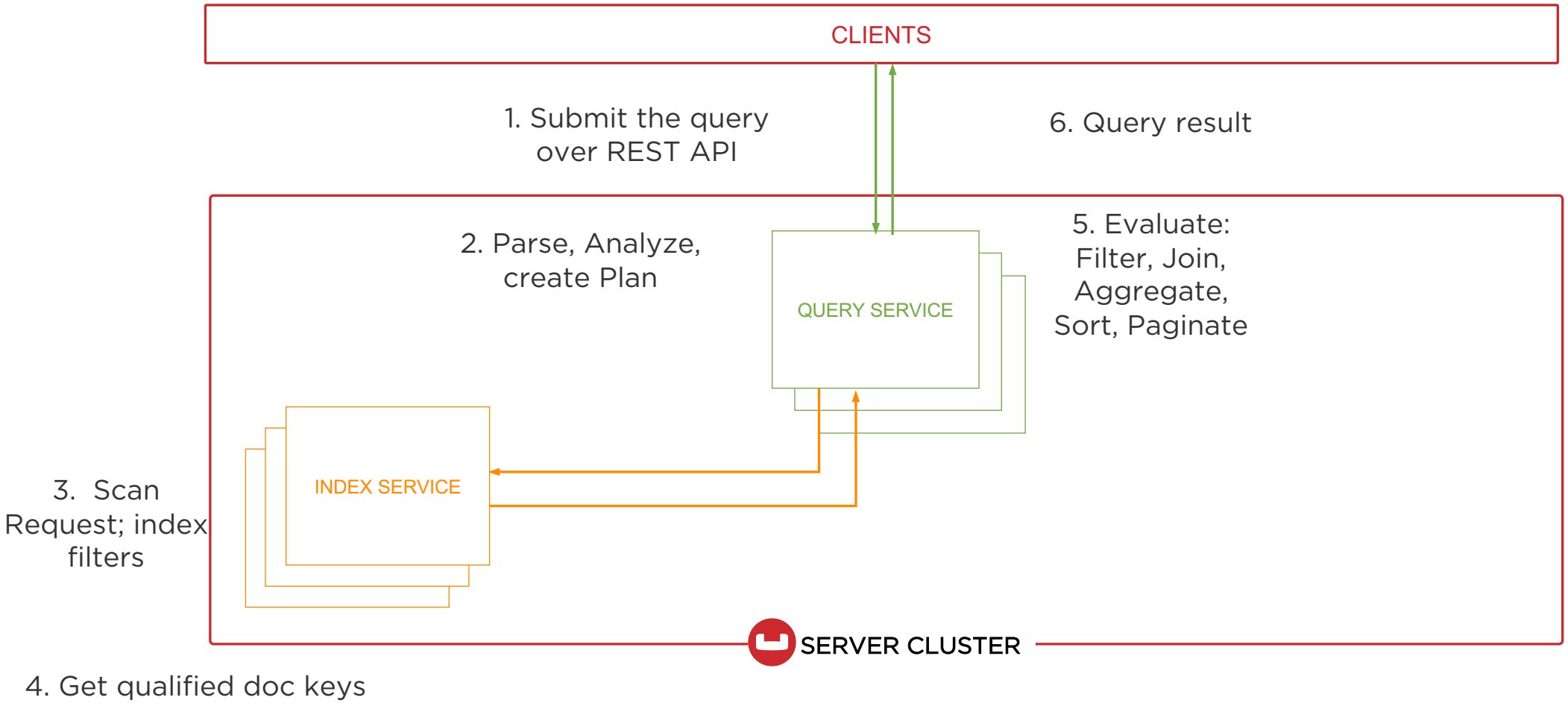


7. Apply final logic (e.g. SORT, ORDER BY)
8. Return formatted results to application

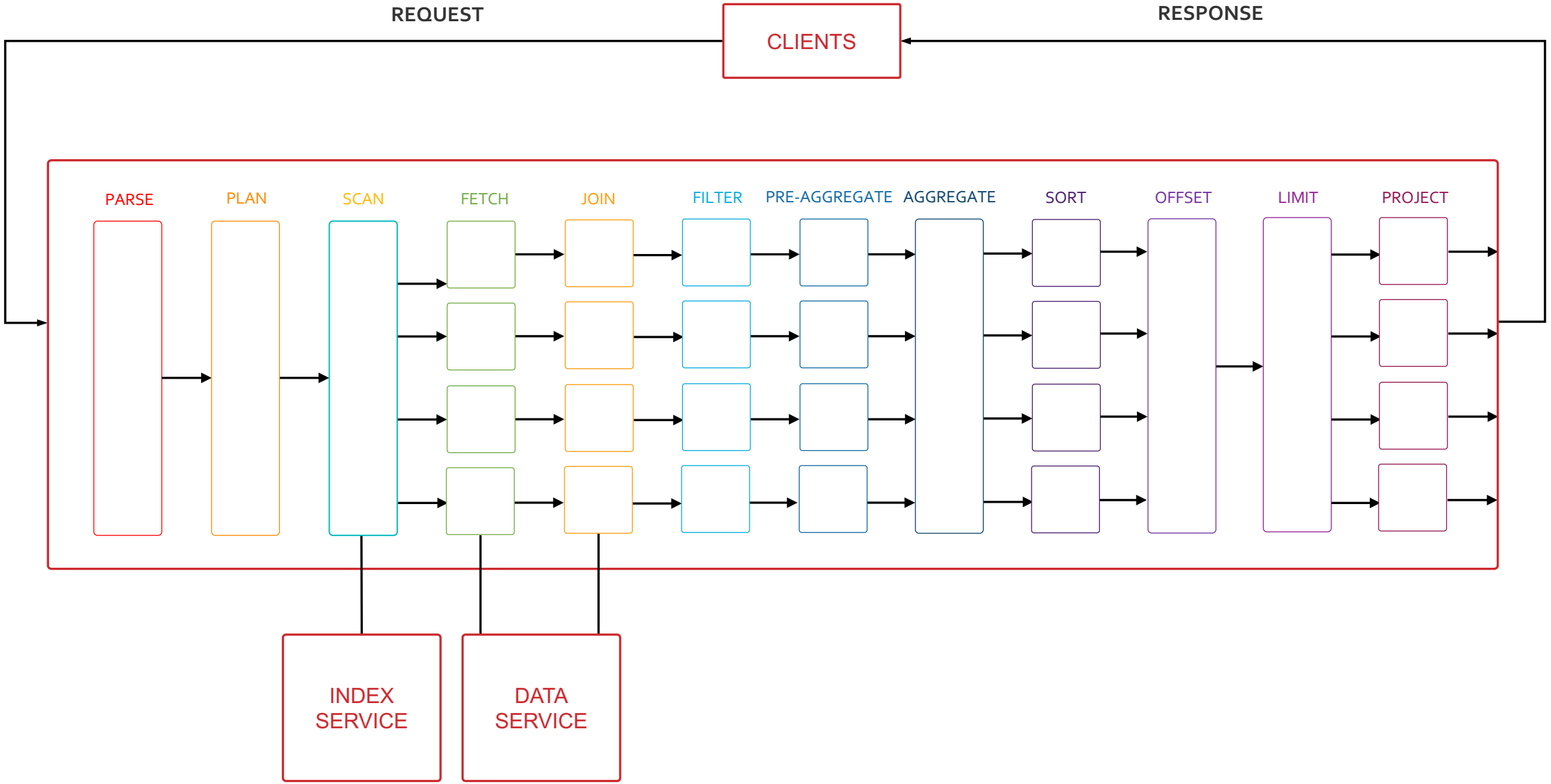
Query Execution Flow (KV Fetch)



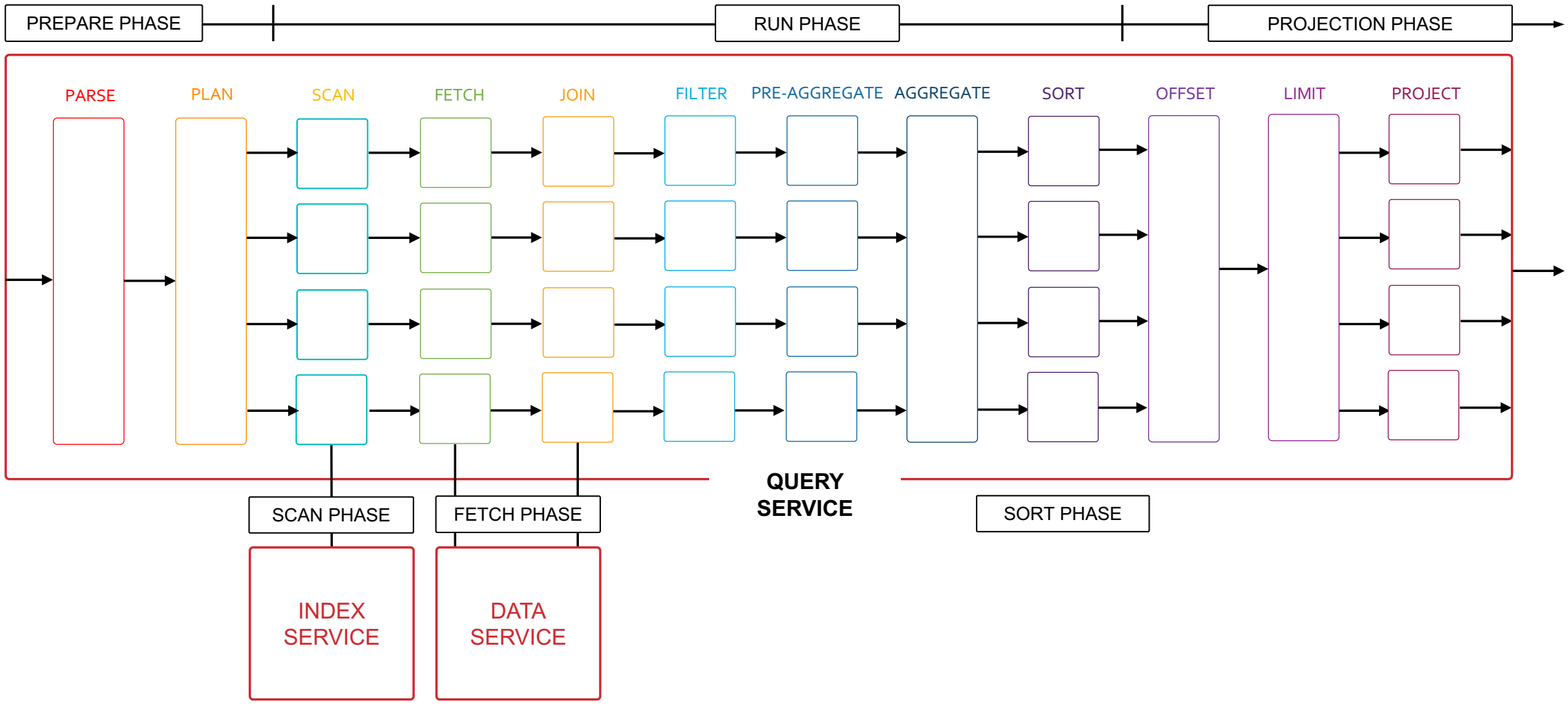
Query Execution Flow (Covering Index)



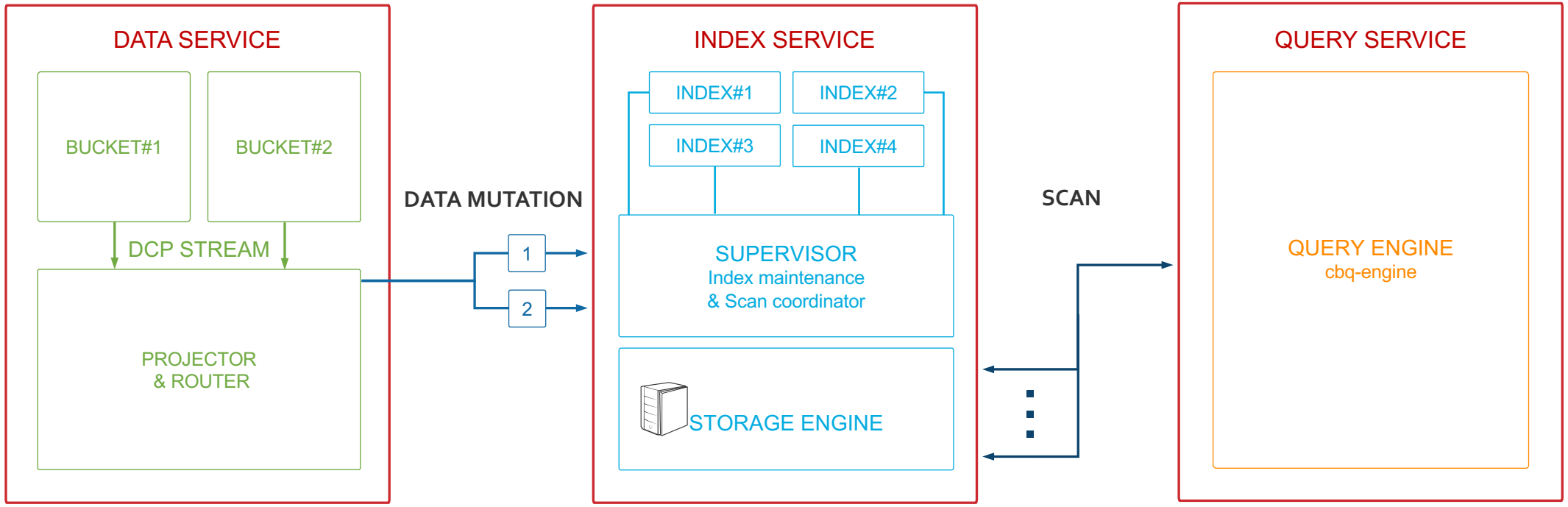
Full Query Pipeline



Inside a Query Service



Couchbase 4.X Services



Couchbase

Indexing

RECAP

Why do we need indexes?

Indexes are used to

efficiently look up objects

meeting *user specified criteria*

without having to

search *every* object

in the collection

Secondary Indexes

- Global Secondary Indexes (GSI) – Released in 4.0 (Now called Legacy GSI / Deprecated in 5.0)
  - Lower query latency and high throughput
  - Isolate from KV operations – multidimensional scaling
- Memory Optimized Indexes (MOI) – new in 4.5
  - Complete index is stored in memory.
  - Supports much higher mutations and better performance
- New Standard Global Secondary Index(GSI) – new in 5.0 (AKA Plasma)
  - New storage engine

Index Options (RECAP)

	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	Duplicate/Replica Index	Pre 5.0 Duplicates/Replicas need to be created manually. The feature of indexing that allows load balancing. Thus providing scale-out, multi-dimensional scaling, performance, and high availability.

N1QL (EXPLAIN)

- EXPLAIN statement when used before any N1QL statement, provides the execution plan for the statement

```
EXPLAIN SELECT log, type, runtime FROM logger ORDER BY runtime

"results": [
  {
    "#operator": "Sequence",
    "~children": [
       {
         "#operator": "Sequence",
         "~children": [
            {
              "#operator": "PrimaryScan",
              "index": "#primary",
              "keyspace": "catalog",
              "namespace": "default"
            } ... and more...
```

Couchbase

N1QL Language

N1QL examples

```
SELECT d.C_ZIP, SUM(ORDLINE.OL_QUANTITY) AS  TOTALQTY
FROM CUSTOMER d
         UNNEST ORDERS as CUSTORDERS
             UNNEST CUSTORDERS.ORDER_LINE AS ORDLINE
WHERE d.C_STATE = "NY"
GROUP BY d.C_ZIP
ORDER BY TOTALQTY DESC;


INSERT INTO CUSTOMER("PQR847", {"C_ID":4723, "Name":"Joe"});


UPDATE CUSTOMER c
SET c.STATE="CA", c.C_ZIP = 94501
WHERE c.ID = 4723;
```

SELECT Statement

```
SELECT   customers.id,

         customers.NAME.lastname,

         customers.NAME.firstname

         Sum(orderline.amount)

FROM     orders UNNEST orders.lineitems AS orderline

         JOIN customers ON KEYS orders.custid

WHERE customers.state = 'NY'

GROUP BY customers.id,

         customers.NAME.lastname
```

Dotted sub-document reference
Names are CASE-SENSITIVE

UNNEST to flatten the arrays

JOINS with Document KEY of customers

Composable SELECT statement

```
SELECT *
FROM
       (           SELECT   a, b, c
                FROM     us_cust
                WHERE    x = 1
                ORDER BY x LIMIT 100 OFFSET 0
             UNION ALL
                SELECT   a, b, c
                FROM     canada_cust
                WHERE    y = 2
                ORDER BY x LIMIT 100 OFFSET 0   ) AS newtab
LEFT OUTER JOIN contacts
     ON KEYS newtab.c.contactid
ORDER BY a, b, c
LIMIT 10 OFFSET 1oo
```

SELECT Statement Highlights

- Querying across relationships
  - JOINs
  - Subqueries
- Aggregation
  - MIN, MAX
  - SUM, COUNT, AVG, ARRAY_AGG [DISTINCT]
- Combining result sets using set operators
  - UNION, UNION ALL, INTERSECT, EXCEPT

N1QL Query Operators [1 of 2]

- USE KEYS ...
  - Direct primary key lookup bypassing index scans
  - Ideal for hash-distributed datastore
  - Available in SELECT, UPDATE, DELETE
- JOIN ... ON KEYS ...
  - Nested loop JOIN using key relationships
  - Ideal for hash-distributed datastore
  - Current implementation supports INNER and LEFT OUTER joins

N1QL Query Operators [2 of 2]

- NEST
  - Special JOIN that embeds external child documents under their parent
  - Ideal for JSON encapsulation
- UNNEST
  - Flattening JOIN that surfaces nested objects as top-level documents
  - Ideal for decomposing JSON hierarchies

JOIN, NEST, and UNNEST can be chained in any combination

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 Data Types from JSON

N1QL supports all JSON data types

- Numbers
- Strings
- Booleans
- Null
- Arrays
- Objects

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

Data Modification Statements

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

  WHEN [NOT] MATCHED THEN ...

Note: Couchbase provides per-document atomicity.

Data Modification Statements

JSON literals can be used in any expression

```
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
```

Index Statements

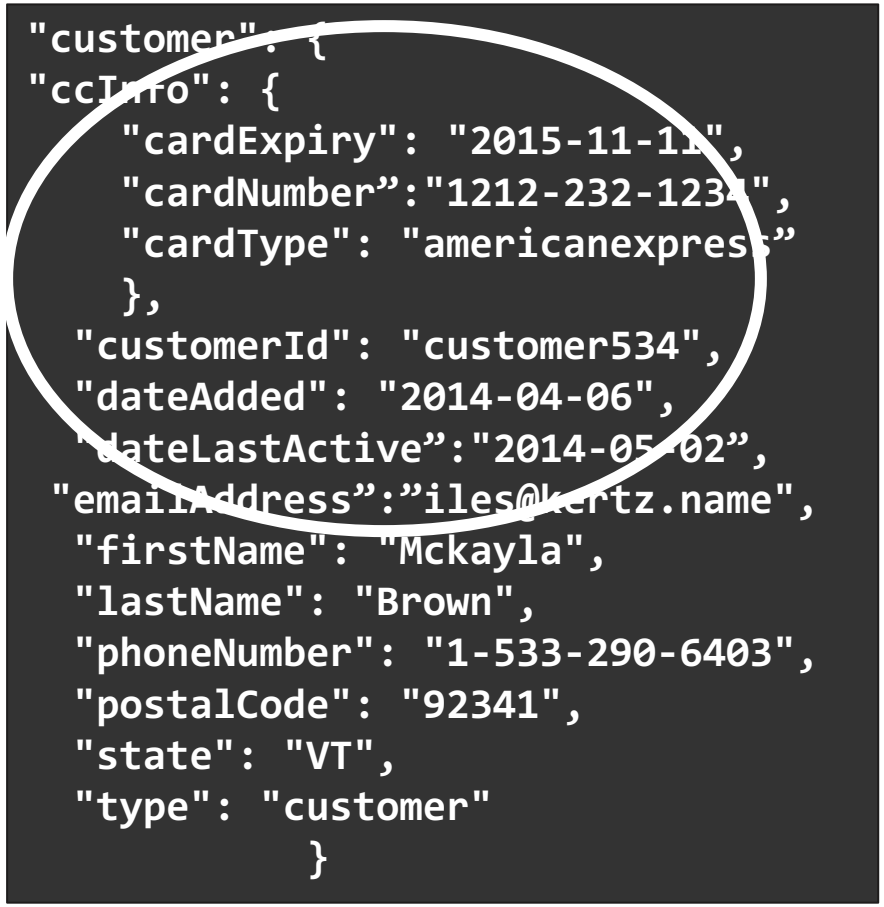
- CREATE INDEX ON ...
- DROP INDEX ...
- EXPLAIN ...

Highlights

- Functional indexes
  - on any data expression
- Partial indexes

Index Overview: Secondary Index

Document key: "customer534"

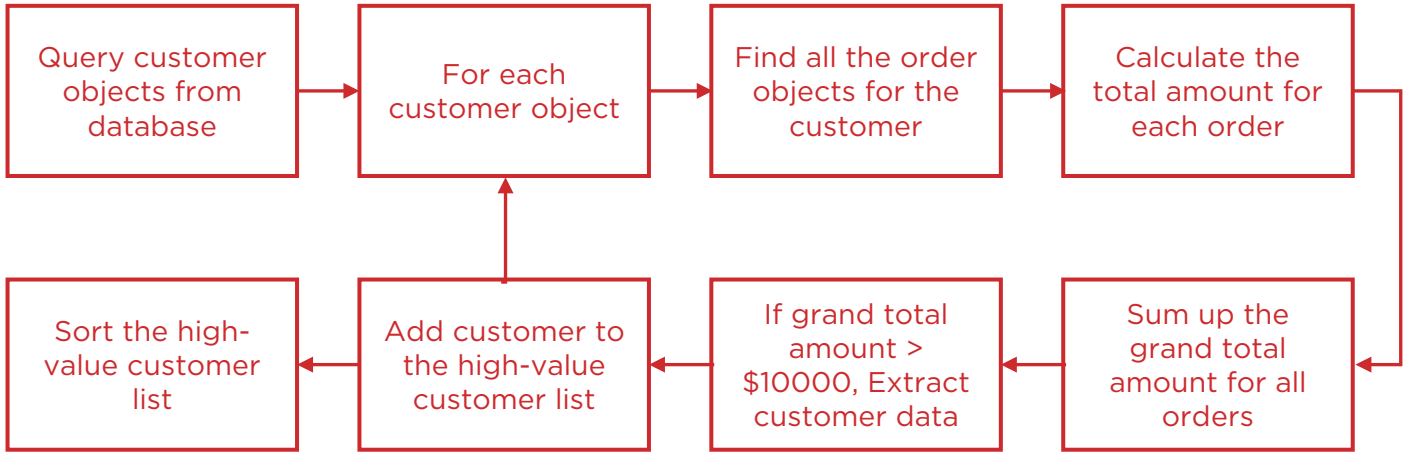


- Secondary Index can be created on any combination of expressions.
  - CREATE INDEX idx_cust_cardnum customer(ccInfo.cardNumber, postalcode)
- Useful in speeding up the queries.
- Need to have matching indexes with matching key-ordering
  - (ccInfo.cardExpiry, postalCode)
  - (type, state, lastName firstName)

Find High-Value Customers with Orders > $10000

LOOPING OVER MILLIONS OF CUSTOMERS IN APPLICATION!!!

API QUERY



- Complex codes and logic
- Inefficient processing on client side

VS.

SQL for JSON

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

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

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

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)

Using N1QL: LIKE

- select SUFFIXES("Baltimore Washington Intl")
- create index suffixes_airport_name on `travel-sample`

  (DISTINCT ARRAY array_element FOR array_element IN

  SUFFIXES(LOWER(airportname)) END)

  WHERE type = "airport";

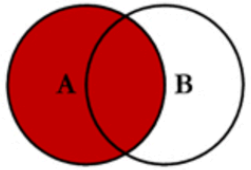
```
SELECT airportname FROM `travel-
sample`
WHERE airportname LIKE
“%Washington%”
AND type = “airport”
```

Using N1QL: JOIN

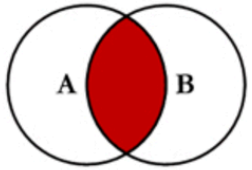
- Inner, Left, Right, Outer, Exclude,

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

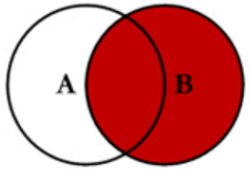
N1QL JOINs



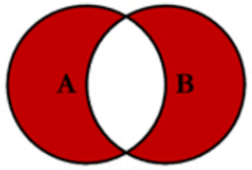

```
SELECT <select_list>
FROM Table_A A
LEFT JOIN Table_B B
ON KEYS <keys-clause(A)>
```



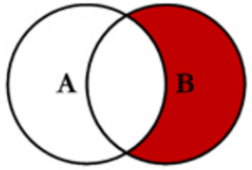

```
SELECT <select_list>
FROM Table_A A
INNER JOIN Table_B B
ON KEYS <keys-clause(A)>
```



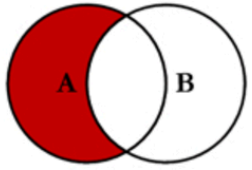

```
SELECT <select_list>
FROM Table_B B
LEFT JOIN Table_A A
ON KEYS <keys-clause(B)>
```



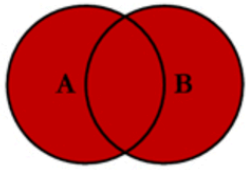

```
SELECT <select_list>
FROM Table_B B
LEFT JOIN Table_A A
ON KEYS <keys-clause(B)>
WHERE META(A).id IS MISSING
UNION ALL
SELECT <select_list>
FROM Table_A A
LEFT JOIN Table_B B
ON KEYS <keys-clause(A)>
WHERE META(B).id IS MISSING
```




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



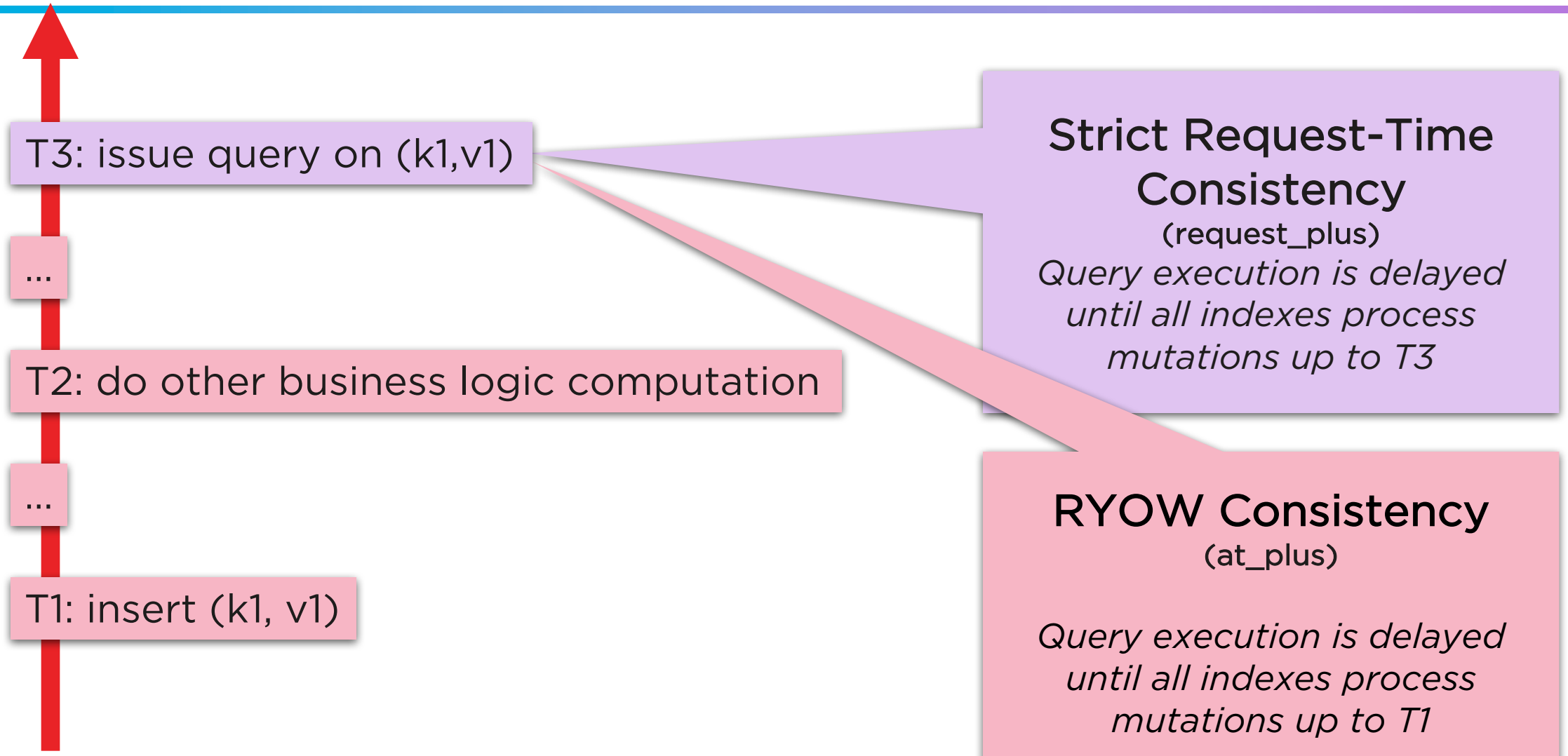

```
SELECT <select_list>
FROM Table_A A
LEFT JOIN Table_B B
ON KEYS <keys-clause(A)>
WHERE META(B).id IS MISSING
```




```
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

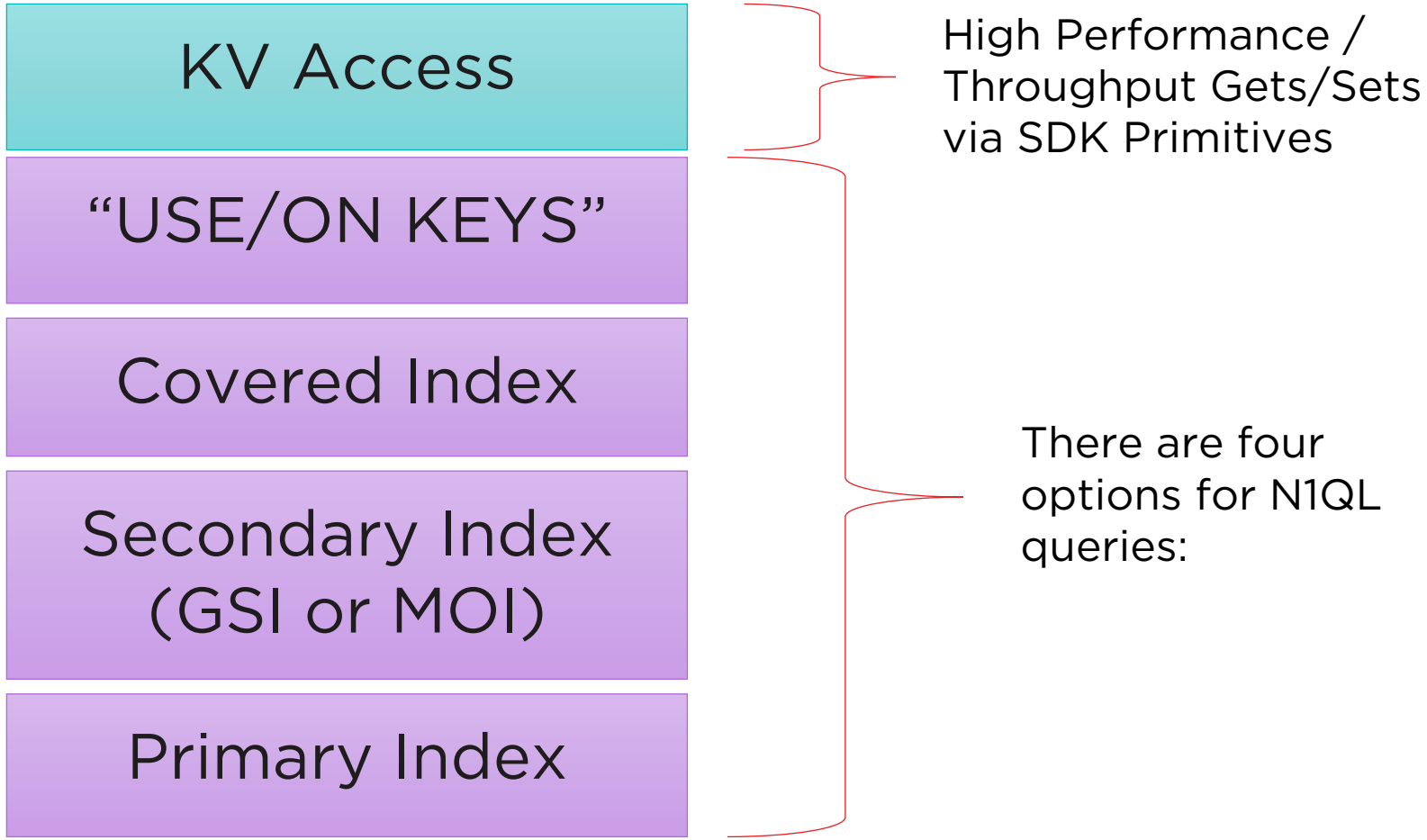
Query Consistency



Couchbase

Query Tuning

Optimizing Query Speed



Query Tuning

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

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

Pattern “USE KEYS”

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

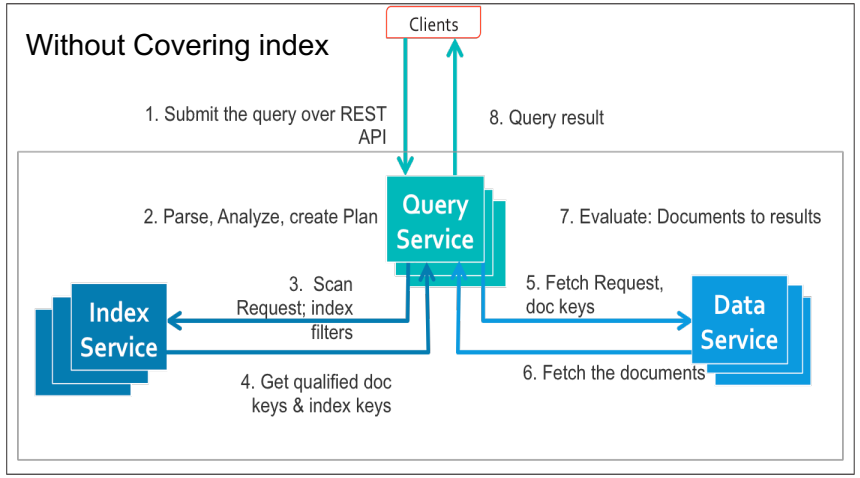
```
SELECT b.destinationairport, b.sourceairport,
  (SELECT c.name
    FROM `travel-sample` c
      USE KEYS b.airlineid
      LEFT UNNEST c.name) as theAirline
FROM `travel-sample` b
USE KEYS "route_5966"
```

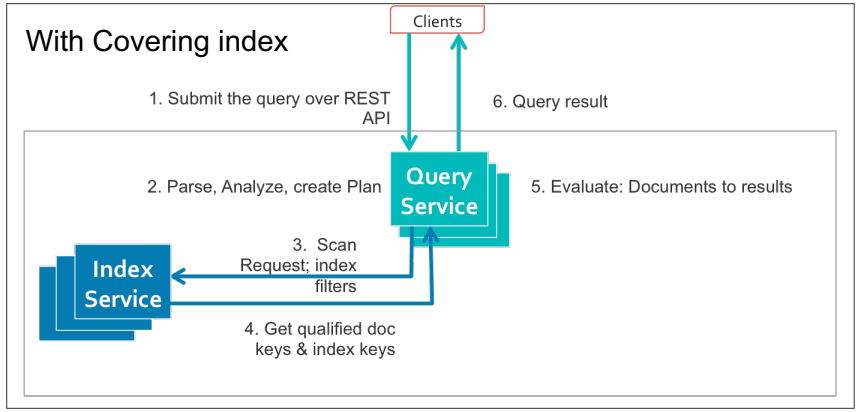
Pattern “Covering Index”

- 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”;
```





Pattern “Array Index and Query”

- Ad Targeting System
  - Raw Logs
  - User Interaction Log
  - User Profile (Ref Doc)

- Add an Index (you can do this at any time!)
  - CREATE INDEX iArray ON default(distinct (array v for v in interactions end))
  - SELECT meta().id from default WHERE ANY v IN interactions SATISFIES v.last_interaction='Wait' END;

```
Hailie20.Doyle34@hotmail.com::interactions
[
 {
  "default": {
   "interactions": [
    {
     "interaction_id": "04ec514a-d6c2-42b7-962a-
bf8976677e79",
     “interaction_type": ”wait"
    }
    {
     "interaction_id": "04ec514a-d6c2-42b7-962a-
ac9467788e88",
     “interaction_type": ”email"
    }
...
```

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>

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>

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>

- Provide an overall health picture of the query service
  - Using REST: GET http://localhost:8093/admin/vitals

Using N1QL: There's MORE!!

- Great Read
  - https://dzone.com/articles/couchbase-n1ql-continuing-to-simplify-transition-f
- Tutorial
  - http://query.pub.couchbase.com/tutorial/#8
- PDF Publication
  - https://drive.google.com/a/couchbase.com/file/d/0B5dv096FGVjNemQ4MENuc0lxQVE/view?usp=sharing

Thank you

Couchbase