SPECIAL EDITION

A GUIDE TO N1QL IN COUCHBASE 5.5

N1QL

SQL + JSON

Sitaram Vemulapalli
Bingjie Miao
Johan Larson
Ajit Yagaty
John Liang
Deepkaran Salooja
Isha Kandaswamy
Keshav Murthy

FOREWORD

N1QL was first released in New York in 2015. Now, it's great to release Couchbase 5.5 and this N1QL 5.5 feature booklet with it. Couchbase 5.5 includes important N1QL language, security and performance features.

ANSI join improvements not only makes the N1QL joins closer to SQL standard joins, but also extends the joins expressions to include arrays. Index partitioning improves index capacity, usability, and performance. Couchbase 5.5 improves grouping and aggregate performance by orders of magnitude. N1QL auditing and x.509 support make it easier for compliance and security with N1QL. Query, indexing and the performance teams have worked closely to measure the performance of all these operations.

Hope you'll find the improvements and the articles about these improvements useful to progress in your enterprise journey.

Keshav Murthy
For the Couchbase N1QL and Indexing Team

April, 20th, 2018
Mountain View, California

AUTHORS

Sitaram Vemulapalli

Sitaram Vemulapalli is a Principal Software Engineer at Couchbase. Prior to Couchbase, he served as an architect for IBM Informix SQL and has more than 20 years experience in database design and development. Sitaram holds a master's degree in system science and automation from the Indian Institute of Science, India.

John Liang

John is a senior architect at Couchbase leading the Couchbase XDCR (replication) and indexing. Previously, he has worked in Oracle, Siebel, Asera. John worked on the full spectrum of the application stack, from database kernel, middleware, enterprise data integration, cloud, and application framework. John holds master's degree in computer engineering from University of Wisconsin, Madison.

Deepkaran Salooja

Deepkaran is a Principal Software Engineer at Couchbase working as a core developer for Global Secondary Indexes. Prior to joining Couchbase, he worked with column oriented database startup, Vertica. He holds bachelor's degree in computer science from GND University, India.

Bingjie Miao

Bingjie is a senior software engineer at Couchbase working on N1QL query processing. Bingjie has worked in IBM and Informix (IDS and XPS) database R&D, developing advanced query processing features for OLTP, OLAP engines including star join optimizations, windowed aggregates, optimizer features. He has a PhD from University of Wisconsin, Madison and several US patents.

Johan Larson

Johan Larson has been a software developer since 2001. He studied databases in graduate school, worked with them from the user side for a couple of now-forgotten startups in the Bay area, and has built server-side systems for companies such as Open Text, Informatica, and Google. He has been a Senior Software Engineer in the Couchbase N1QL team since 2015.

Isha Kandaswamy

Isha Kandaswamy is a Senior Software Engineer in the query group at Couchbase. Prior to Couchbase, she worked in the query optimization group at Teradata. Isha earned her master's degree in computer science and robotics from Johns Hopkins University.

Ajit Yagaty

Ajit Yagaty is a senior software engineer working cluster management and security for Couchbase NoSQL database. He has diverse technology background and experience in designing/developing software components across the stack, right from distributed systems development to kernel module development to firmware development. He has bachelor's degree in Computer Science from Visvesvaraya Technological University, India.

Keshav Murthy

Keshav Murthy is a Senior Director at Couchbase of N1QL R&D. Previously, he was a Senior Director at MapR, Senior Architect for IBM, with more than 20 years experience in database design & development. He lead the SQL and NoSQL R&D team at IBM Informix. He has received two President's Club awards at Couchbase, two Outstanding Technical Achievement Awards at IBM. Keshav has a bachelors degree in Computer Science and Engineering from the University of Mysore, India, holds eight US patents and invented databases for systems of engagement.

TABLE OF CONTENTS

COUCHBASE 5.5: OVERVIEW OF QUERY AND INDEXING FEATURES

Keshav Murthy 9

ANSI JOINS IN N1QL

Bingjie Miao 17

UNDERSTANDING INDEX GROUPING AND AGGREGATION IN COUCHBASE N1QL QUERY

Sitaram Vemulapalli 49

INDEX PARTITIONING

John Liang • Keshav Murthy 79

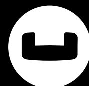
AUDITING COUCHBASE N1QL STATEMENTS

Johan Larson 91

N1QL SUPPORT FOR X.509

Isha Kandaswamy • Ajit Yagaty 105

Overview of Couchbase Query and Indexing Features



COUCHBASE 5.5: OVERVIEW OF QUERY AND INDEXING FEATURES.

Author: Keshav Murthy, Couchbase R&D

Language Features	**Security & Infrastructure Features**

• ANSI Joins support
  • INNER JOIN
  • LEFT OUTER
  • RIGHT OUTER
  • NEST and UNNEST
  • JOIN on arrays
	• N1QL Auditing
	• X.509 Support
	• IPV6 Support
	• Backfill
	• ALTER INDEX

Performance Features	**Query Workbench Features**
• GROUP BY performance • Aggregation performance • Index Partitioning • Query pipeline performance • Hash join	• Visual Explain improvements • Parameters for Query • Easy copy results to Excel

Language Features:

JOIN is one of the foundational operations in SQL. Hence, N1QL has implemented INNER JOIN and LEFT OUTER JOIN from the first release. In addition, N1QL has added NEST and UNNEST operations to work with arrays, a commonly used data structure in JSON. So far, the join expression was limited to equality child-to-parent or parent-to-child documents.

In Couchbase 5.5, you can join on any complex expression just like SQL joins. We've also added limited support for RIGHT OUTER joins.

So far, the join execution used a method called Block Nested Loop Join. This algorithm works fine when the number of documents in the outer table is limited. When the number of documents is high (e.g. reporting queries), the query latency can be high.

In Couchbase 5.5, we introduce hash join to improve the performance of join queries with a large number of documents. The article on "ANSI JOINs in N1QL" describes the details on join language and performance improvements.

Performance Features:

One of the common ways to improve query performance is to create an index that covers the query. For queries which require grouping and aggregation, even when the query is covered by an index, we fetch all of the qualified document keys, index keys to the query service, then group and aggregate on that data. This process copies the data multiple times, from the indexer, over the network to the indexer client, to the query service. If the scan result is large, we end up using backfill to write the results into the file and read it later on.

After selecting the index for the query, Couchbase 5.5 automatically evaluates the possibility of the indexer evaluating the grouping and aggregation. If it's feasible, the scan request will include additional requests to group and aggregate within the indexer. Couchbase 5.5's index service can now do runtime grouping and aggregation. That means, the indexer can scan, group and aggregate the data within the indexer, eliminating multiple data copy operations and improving both the latency and throughput of these queries.

The article, "Understanding Index Grouping And Aggregation in Couchbase N1QL Query" describes this process in details.

In Couchbase, data is always partitioned using the consistent hash value of the document key into buckets which are stored in multiple nodes. For indexer, however, you had to partition them manually using the WHERE clause on the CREATE INDEX. This manual partitioning is cumbersome to manage and required queries to be written carefully to match one of the predicates in the WHERE clause.

In Couchbase 5.5, you can simply partition the index using the hash strategy. The hash partitioned index will increase the index capacity by creating multiple indexes on a single large collection of documents easily. For query processing, for simple queries with equality predicates on hash key, you get uniform distribution of the workload on these multiple indexes. For complex queries including the grouping & aggregation we discussed above, the scans, partial aggregations are done in parallel, improving the query latency. The article "index partitioning" will describe this feature in detail.

Security and Infrastructure Features:

N1QL Auditing helps customers implement regulatory compliance like HIPPA, GDPR by auditing every N1QL statement executed on the system. You can configure the auditing to auditing specific or all users and/or roles, specific or all statements. The article "N1QL Auditing" will explain the feature, its use case in detail.

Couchbase 5.5 supports using certificate authority signed certificates for public key certificates used in the TLS (Transport Layer Security). The article "N1QL Support for X.509" describes the feature in N1QL.

ALTER INDEX helps you to move the indexes from one node to the other, change the number of replica for the index, generally improving the index manageability. See ALTER INDEX documentation for details.

Index scans are efficient and fast. When an index scan request returns a large number of qualified keys, query service may not be able to consume at the same speed. To avoid any backlog of these index scan results in the indexer, the query service (actually the index client within the query service) fetches rest of the results and writes into a temporary file. These backfill files will be purged once the query completes. The location of these files was always /tmp on Linux and TEMPDIR on Windows. In Couchbase 5.5, you can change the location of the directory where the backfill files are created and set a quote for maximum storage that can be used. See Query Temp Disk Path documentation for details.

Query Workbench Features

Couchbase 5.0 added visual explain to easily understand the query plan and debug performance issues by looking at the profile statistics on each of the query operator. The picture below should say it all!

Query Editor

```
1  select count(1)
2  from `beer-sample` beer inner join `beer-sample` brewery USE HASH(build)
3                          on (beer.brewery_id = brewery.myid)
4  where beer.type = 'beer'
```

Execute Explain ✔ success | elapsed: 392.15ms | execution: 392.11ms | count: 1 | size: 11

Query Results

Indexes: *beer-sample.beer_primary*
Buckets: *beer-sample*
Fields: *beer-sample.type*

Stream
00:00.0000
1 in / 1 out

Project
1 terms
00:00.0000
1 in / 1 out

Group
Aggrs: count(1)
00:00.0159 (2.9%)
5891 in / 1 out

Filter
((`beer`.`type`) = "beer")
00:00.0428 (7.8%)
5891 in / 5891 out

HashJoin
on: ((`beer`.`brewery_id`) = cover ...
00:00.1273 (23.2%)
7303 in / 5891 out

Fetch
beer-sample as beer
00:00.3057 (55.7%)
7303 in / 7303 out

PrimaryScan3
00:00.0560 (10.2%)
7303 out

Authorize
00:00.0012 (0.2%)

IndexScan3
beer-sample.ixid
as: brewery
7303 out

HashJoin
#stats
- #itemsIn - 7303
- #itemsOut - 5891
- #phaseSwitches - 41001
- execTime - 127.344515ms
- kernTime - 261.939286ms

build_aliases
- brewery

build_exprs
- cover ((`brewery`.`myid`))

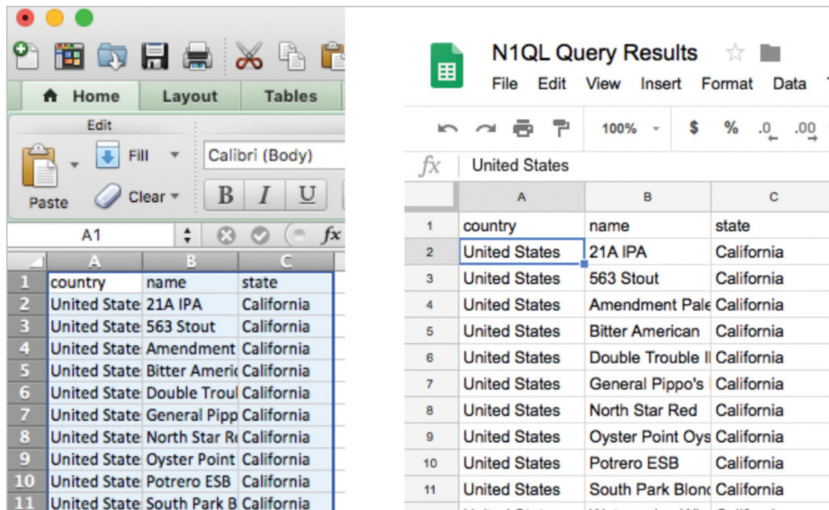
on_clause - ((`beer`.`brewery_id`) = cover ((`brewery`.`myid`)))
probe_exprs
- (`beer`.`brewery_id`)

#time_normal - 00:00.1273
#time_absolute - 0.127344515

In addition, you can now run prepared statement with different parameters by setting different query specific parameters, timeouts, profiling parameters for each run.

Last, but not the least, we've made it easier to copy the results from query workbench to Excel or any other tool expecting tabular output. The picture below shows tabular output and then simply click on the copy icon next to Query results.

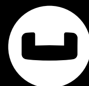
Simply paste into Excel or Google Sheets. It'll copy like magic :-)



References

1. What's new in Couchbase 5.5?
   https://developer.couchbase.com/documentation/server/5.5/introduction/whats-new.html
2. Couchbase blogs: https://blog.couchbase.com/
3. Couchbase Documentation:
   https://developer.couchbase.com/documentation/server/5.5/introduction/intro.html

ANSI Joins in N1QL



ANSI JOINS IN N1QL

Author: Bingjie Miao, Senior Software Engineer, Couchbase R&D

Overview:

ANSI JOIN support is added in N1QL to Couchbase version 5.5. Previous versions of Couchbase only support lookup join and index join. Lookup join and index join works great when the document key from one side of the join can be produced by the other side of the join, i.e., joining on a parent-child or child-parent relationship through document key. Where they fall short is when the join is on arbitrary fields or expressions of fields, or when multiple join conditions are required. ANSI JOIN is a standardized join syntax widely used in relational databases. ANSI JOIN is much more flexible than lookup join and index join, allowing join to be done on arbitrary expressions on any fields in a document, this makes join operations much simpler and more powerful.

ANSI JOIN syntax:

```
lhs-expression [ join-type ] JOIN rhs-keyspace ON [ join-condition ]
```

The left-hand side of the join, lhs-expression can be a keyspace, a N1QL expression, a subquery, or a previous join. The right-hand side of the join, rhs-keyspace, must be a keyspace. The ON-clause specifies the join condition, which can be any arbitrary expression, although it should contain predicates that allows an index scan on the right-hand side keyspace. Join-type can be INNER, LEFT OUTER, RIGHT OUTER. The INNER and OUTER keywords are optional, thus JOIN is the same as INNER JOIN, and LEFT JOIN is the same as LEFT OUTER JOIN. In relational databases join-type can also be FULL OUTER or CROSS, although FULL OUTER JOIN and CROSS JOIN are not supported currently in N1QL.

Details of ANSI JOIN support

We'll use examples to show you new ways you can run queries using ANSI JOIN syntax, and how to transform your existing join queries in N1QL from lookup join or index join syntax into new ANSI JOIN syntax. It should be noted that lookup join and index join will continue to be supported in N1QL for backward compatibility, however you cannot mix lookup join or index join with the new ANSI JOIN syntax in the same query block, thus customers are encouraged to migrate to the new ANSI JOIN syntax.

To follow along, install travel-sample sample bucket.

Example 1: ANSI JOIN with arbitrary join condition

The join condition (ON-clause) for ANSI JOIN can be any expression, involving any fields of the documents being joined. For example:

Required index:

```
CREATE INDEX route_airports ON `travel-sample`(sourceairport, destinationairport)
WHERE type = “route”;
```

Optional index:

```
CREATE INDEX airport_city_country ON `travel-sample`(city, country) WHERE type =
“airport”;
```

Query:

```
SELECT DISTINCT route.destinationairport
FROM `travel-sample` airport JOIN `travel-sample` route
       ON airport.faa = route.sourceairport
        AND route.type = "route"
WHERE airport.type = "airport"
  AND airport.city = "San Francisco"
 AND airport.country = "United States";
```

In this query we are joining a field (“faa”) from the airport document with a field (“sourceairport”) from the route document (see the ON clause of the join). Such join is not possible with lookup join or index join in N1QL, since both requires joining on document key only.

ANSI JOIN requires an appropriate index on the right-hand side keyspace (“Required index” above). You can also create other indexes (e.g. “Optional index” above) to speed up your query. Without the optional index a primary scan will be used and query still works, however without the required index the query will not work and will return an error.

Looking at the explain:

```
"plan": {
  "#operator": "Sequence",
  "~children": [
    {
      "#operator": "IndexScan3",
      "as": "airport",
      "index": "airport_city_country",
      "index_id": "8e782fd1b124eec3",
      "index_projection": {
        "primary_key": true
      },
      "keyspace": "travel-sample",
      "namespace": "default",
      "spans": [
        {
          "exact": true,
          "range": [
            {
              "high": "\"San Francisco\"",
              "inclusion": 3,
              "low": "\"San Francisco\""
            },
            {
              "high": "\"United States\"",
              "inclusion": 3,
              "low": "\"United States\""
            }
          ]
        }
      ],
      "using": "gsi"
    },
    {
      "#operator": "Fetch",
      "as": "airport",
      "keyspace": "travel-sample",
      "namespace": "default"
    },
    {
      "#operator": "Parallel",
      "~child": {
        "#operator": "Sequence",
        "~children": [
          {
            "#operator": "NestedLoopJoin",
            "alias": "route",
```

```
            "on_clause": "(((`airport`.`faa`) = cover ((`route`.`sourceairport`))) and
(cover ((`route`.`type`)) = \"route\"))",
            "~child": {
              "#operator": "Sequence",
              "~children": [
                {
                  "#operator": "IndexScan3",
                  "as": "route",
                  "covers": [
                    "cover ((`route`.`sourceairport`))",
                    "cover ((`route`.`destinationairport`))",
                    "cover ((meta(`route`).`id`))"
                  ],
                  "filter_covers": {
                    "cover ((`route`.`type`))": "route"
                  },
                  "index": "route_airports",
                  "index_id": "f1f4b9fbe85e45fd",
                  "keyspace": "travel-sample",
                  "namespace": "default",
                  "nested_loop": true,
                  "spans": [
                    {
                      "exact": true,
                      "range": [
                        {
                          "high": "(`airport`.`faa`)",
                          "inclusion": 3,
                          "low": "(`airport`.`faa`)"
                        }
                      ]
                    }
                  ],
                  "using": "gsi"
                }
              ]
            }
          },
          {
            "#operator": "Filter",
            "condition": "((((`airport`.`type`) = \"airport\") and ((`airport`.`city`)
= \"San Francisco\")) and ((`airport`.`country`) = \"United States\"))"
          },
          {
            "#operator": "InitialProject",
            "distinct": true,
            "result_terms": [
              {
```

```
                "expr": "cover ((`route`.`destinationairport`))"
              }
            ]
          },
          {
            "#operator": "Distinct"
          },
          {
            "#operator": "FinalProject"
          }
        ]
      }
    },
    {
      "#operator": "Distinct"
    }
  ]
}
```

You will see a NestedLoopJoin operator is used to perform the join, and underneath that an IndexScan3 operator is used to access the right-hand side keyspace, “route”. The spans for the index scan looks like:

```
"spans": [
  {
    "exact": true,
    "range": [
      {
        "high": "(`airport`.`faa`)",
        "inclusion": 3,
        "low": "(`airport`.`faa`)"
      }
    ]
  }
]
```

The index scan for the right-hand side keyspace ("route") is using a field ("faa") from the left-hand side keyspace ("airport") as search key. For each document from outer side keyspace "airport" the NestedLoopJoin operator performs an index scan on the inner side keyspace "route" to find matching documents, and produces join results. The join is performed in a nested-loop fashion, where the outer loop produces document from outer side keyspace, and a nested inner loop searches for matching inner side document for the current outer side document.

The explain information can also be view graphically on Query Workbench, by clicking the Explain button followed by the Plan button:

Query Editor ← history (96/97) →

```
1 SELECT DISTINCT route.destinationairport
2 FROM `travel-sample` airport JOIN `travel-sample` route
3      ON airport.faa = route.sourceairport
4         AND route.type = "route"
5 WHERE airport.type = "airport"
6   AND airport.city = "San Francisco"
7   AND airport.country = "United States";
```

Execute | Explain | ✔ explain success | elapsed: 3.76ms | execution: 3.74ms | count: 1 | size: 1914 | preferences

Query Results | JSON | Table | Tree | Plan | Plan Text

Indexes	Buckets	Fields
travel-sample.airport_city_country	travel-sample route	travel-sample.type travel-sample.city travel-sample.country route.destinationairport

Filter
(((`airport`.`type`) = "airport") ...

NestedLoopJoin
on: (((`airport`.`faa`) = cover ((`...

Fetch
travel-sample as airport

IndexScan3
travel-sample.airport_city_country
as: airport

IndexScan3
travel-sample.route_airports
as: route

Visual Explain for ANSI JOIN

In this example the index scan on the right-hand side keyspace is a covered index scan. In case the index scan is not covered, a fetch operator will be following the index scan operator to fetch the document.

It should be noted that nested-loop join requires an appropriate secondary index on the right-hand side keyspace of ANSI JOIN. Primary index is not considered for this purpose. If an appropriate secondary cannot be found, an error will be returned for the query.

In addition, you might have noticed that the filter route.type = "route" appears in the ON-clause as well. The ON-clause is different than the WHERE clause in that the ON-clause is evaluated as part of the join, while the WHERE clause is evaluated after all joins are done. This distinction is important, especially for outer joins. Therefore it is recommended that you include filters on the right-hand side keyspace for a join in the ON-clause as well, in addition to any join filters.

Example 2: ANSI JOIN with multiple join conditions

While lookup join and index join only joins on a single join condition (equality of document key), the ON-clause of ANSI JOIN can contain multiple join conditions.

Required index:

```
CREATE INDEX landmark_city_country ON `travel-sample`(city, country) WHERE type =
"landmark"; Optional index:
CREATE INDEX hotel_title ON `travel-sample`(title) WHERE type = "hotel";
```

Query:

```
SELECT hotel.name hotel_name, landmark.name landmark_name, landmark.activity
FROM `travel-sample` hotel JOIN `travel-sample` landmark
    ON hotel.city = landmark.city AND hotel.country = landmark.country AND landmark.
type = "landmark"
WHERE hotel.type = "hotel" AND hotel.title like "Yosemite%" AND array_length(hotel.
public_likes) > 5;
```

Looking at the explain, the index spans for the index ("landmark_city_country") of the right-hand side keyspace ("landmark") is:

```
"spans": [
  {
    "exact": true,
    "range": [
      {
```

```
        "high": "(`hotel`.`city`)",
        "inclusion": 3,
        "low": "(`hotel`.`city`)"
      },
      {
        "high": "(`hotel`.`country`)",
        "inclusion": 3,
        "low": "(`hotel`.`country`)"
      }
    ]
  }
]
```

Thus multiple join predicates can potentially generate multiple index search keys for the index scan of the inner side of a nested-loop join.

Example 3: ANSI JOIN with complex join expressions

The join condition in the ON-clause can be complex join expression. For example, the "airlineid" field in "route" document corresponds to the document key for "airline" document, but it can also be constructed by concatenating "airline_" with the "id" field of the "airline" document.

Required index:

```
1  CREATE INDEX route_airlineid ON `travel-sample`(airlineid) WHERE type = "route";
```

Optional index:

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

Query:

```
SELECT count(*)
FROM `travel-sample` airline JOIN `travel-sample` route
       ON route.airlineid = "airline_" || tostring(airline.id) AND route.type = "route"
   WHERE airline.type = "airline" AND airline.name = "United Airlines";
```

The explain contains the following index spans for the right-hand side keyspace("route"):

```
"spans": [
  {
    "exact": true,
    "range": [
      {
        "high": "(\"airline_\" || to_string((`airline`.`id`)))",
```

```
        "inclusion": 3,
        "low": "(\"airline_\" || to_string((`airline`.`id`)))"
      }
    ]
  }
]
```

The expression will be evaluated at runtime to generate the search keys for the index scan on the inner side of nested-loop join.

Example 4: ANSI JOIN with IN clause

The join condition does not need to be an equality predicate. An IN-clause can be used as join condition.

Required index:

```
CREATE INDEX airport_faa_name ON `travel-sample`(faa, airportname) WHERE type =
“airport”;
```

Optional index:

```
CREATE INDEX route_airline_distance ON `travel-sample`(airline, distance) WHERE type =
“route”;
```

Query

```
SELECT DISTINCT airport.airportname

FROM `travel-sample` route JOIN `travel-sample` airport

    ON airport.faa IN [ route.sourceairport, route.destinationairport ] AND airport.
type = “airport”

WHERE route.type = “route” AND route.airline = “F9” AND route.distance > 3000;
```

The explain contains the following index spans for the right-hand side keyspace(“airport”):

```
"spans": [
  {
    "range": [
      {
        "high": "(`route`.`sourceairport`)",
        "inclusion": 3,
        "low": "(`route`.`sourceairport`)"
```

```
      }
    ]
  },
  {
    "range": [
      {
        "high": "(`route`.`destinationairport`)",
        "inclusion": 3,
        "low": "(`route`.`destinationairport`)"
      }
    ]
  }
]
```

Example 5: ANSI JOIN with OR clause

Similar to IN-clause, the join condition for an ANSI JOIN can also contain an OR-clause. Different arms of the OR-clause can potentially reference different fields of the right-hand side keyspace, as long as appropriate indexes exist.

Required index (route_airports index same as example 1):

```
CREATE INDEX route_airports ON `travel-sample`(sourceairport, destinationairport) WHERE
type = “route”;
CREATE INDEX route_airports2 ON `travel-sample`(destinationairport, sourceairport)
WHERE type = “route”;
```

Optional index (same as example 1):

```
1       CREATE INDEX airport_city_country ON `travel-sample`(city, country) WHERE type =
“airport”;
```

Query:

```
SELECT count(*)
FROM `travel-sample` airport JOIN `travel-sample` route
    ON (route.sourceairport = airport.faa OR route.destinationairport = airport.faa)
AND route.type = “route”
WHERE airport.type = “airport” AND airport.city = “Denver” AND airport.country =
“United States”;
```

The explain shows an UnionScan being used under NestedLoopJoin, to handle the OR-clause:


```
"#operator": "UnionScan",
"scans": [
  {
    "#operator": "IndexScan3",
    "as": "route",
    "index": "route_airports",
    "index_id": "f1f4b9fbe85e45fd",
    "index_projection": {
      "primary_key": true
    },
    "keyspace": "travel-sample",
    "namespace": "default",
    "nested_loop": true,
    "spans": [
      {
        "exact": true,
        "range": [
          {
            "high": "(`airport`.`faa`)",
            "inclusion": 3,
            "low": "(`airport`.`faa`)"
          }
        ]
      }
    ],
    "using": "gsi"
  },
  {
    "#operator": "IndexScan3",
    "as": "route",
    "index": "route_airports2",
    "index_id": "cdc9dca18c973bd3",
    "index_projection": {
      "primary_key": true
    },
    "keyspace": "travel-sample",
    "namespace": "default",
    "nested_loop": true,
    "spans": [
      {
        "exact": true,
        "range": [
          {
            "high": "(`airport`.`faa`)",
            "inclusion": 3,
            "low": "(`airport`.`faa`)"
```

```
            }
          ]
        }
      ],
      "using": "gsi"
    }
]
```

Example 6: ANSI JOIN with hints

For lookup join and index join, hints can only be specified on the keyspace on the left-hand side of the join. For ANSI JOIN, hints can be specified on the right-hand side keyspace as well. Using the same query as example 1 (with addition of USE INDEX hint):

```
SELECT DISTINCT route.destinationairport
FROM `travel-sample` airport JOIN `travel-sample` route USE INDEX(route_airports)
    ON airport.faa = route.sourceairport AND route.type = “route”
WHERE airport.type = “airport” AND airport.city = “San Francisco” AND airport.country =
“United States”;
```

The USE INDEX hint limits the number of indexes the planner needs to consider for performing the join.

Hints can also be specified on the left-hand side keyspace of ANSI JOIN.

```
SELECT DISTINCT route.destinationairport
FROM `travel-sample` airport USE INDEX(airport_city_country)
   JOIN `travel-sample` route USE INDEX(route_airports)
    ON airport.faa = route.sourceairport AND route.type = “route”
WHERE airport.type = “airport” AND airport.city = “San Francisco” AND airport.country =
“United States”;
```

Example 7: ANSI LEFT OUTER JOIN

So far we’ve been looking at inner joins. You can also perform LEFT OUTER JOIN by just including LEFT or LEFT OUTER keywords in front of JOIN keyword in join specification.

Required index (same as example 1):

```
CREATE INDEX route_airports ON `travel-sample`(sourceairport, destinationairport)
WHERE type = “route”;
```

Optional index (same as example 1):

```
CREATE INDEX airport_city_country ON `travel-sample`(city, country) WHERE type =
“airport”;
```

Query:

```
SELECT airport.airportname, route.airlineid
FROM `travel-sample` airport LEFT JOIN `travel-sample` route
    ON airport.faa = route.sourceairport AND route.type = “route”
WHERE airport.type = “airport” AND airport.city = “Denver” AND airport.country =
“United States”;
```

The result set for this query contains all the joined results, as well as any left-hand side (“airport”) document that does not join with the right-hand side (“route”) document, according to semantics of LEFT OUTER JOIN. Thus you’ll find results that just contain airport.airportname but not route.airlineid (which is missing). You can also select just the left-hand side (“airport”) document that does not join with right-hand side (“route”) document by adding a IS MISSING predicate on the right-hand side keyspace (“route”):

```
SELECT airport.airportname, route.airlineid
FROM `travel-sample` airport LEFT JOIN `travel-sample` route
       ON airport.faa = route.sourceairport AND route.type = “route”
WHERE airport.type = “airport” AND airport.city = “Denver” AND airport.country =
“United States”
  AND route.airlineid IS MISSING;
```

Example 8: ANSI RIGHT OUTER JOIN

ANSI RIGHT OUTER JOIN is similar to ANSI LEFT OUTER JOIN except we preserve the right-hand side document if no join occurs. We can modify the query in example 7 by switching the left-hand side and right-hand side keyspaces, and replacing LEFT keyword with RIGHT keyword:

```
SELECT airport.airportname, route.airlineid
FROM `travel-sample` route RIGHT JOIN `travel-sample` airport
    ON airport.faa = route.sourceairport AND route.type = “route”
WHERE airport.type = “airport” AND airport.city = “Denver” AND airport.country =
“United States”;
```

Note that although we switched airport and route in join specification, the filter on route (now the left-hand side keyspace) still appears in the ON-clause of the join, since route is still on the subservient side in this outer join.

RIGHT OUTER JOIN is internally converted to LEFT OUTER JOIN.

If a query contains multiple joins, a RIGHT OUTER JOIN can only be the first join specified. Since N1QL only support linear joins, i.e., the right-hand side of each join must be a single keyspace, if a RIGHT OUTER JOIN is not the first join specified, then after converting to LEFT OUTER JOIN, the right-hand side of the join now contains multiple keyspaces, which is not supported. If you specify RIGHT OUTER JOIN in any position other than the first join, a syntax error will be returned.

Example 9: ANSI JOIN using Hash Join

N1QL supports two join methods for ANSI JOIN. The default join method for an ANSI JOIN is nested-loop join. The alternative is hash join. Hash join uses a hash table to match documents from both sides of the join. Hash join has a build side and a probe side, where each document from the build side is inserted into a hash table based on values of equi-join expression from the build side; subsequently each document from the probe side looks up from the hash table based on values of equi-join expression from the probe side. If a match is found then the join operation is performed.

Compared with nested-loop join, hash join can be more efficient when the join is large, e.g., when there are tens of thousand or more documents from the left-hand side of the join (after applying filters). If using nested-loop join, then for each document from the left-hand side an index scan needs to be performed on the right-hand side index. As the number of documents from the left-hand side increases, nested-loop join becomes less efficient.

For hash join, the smaller side of the join should be used for building the hash table, and the larger side of the join should be used for probing the hash table. It should be noted that hash join does require more memory than nested-loop join, since an in-memory hash table is required. The amount of memory required is proportional to the number of documents from the build side, as well as average size of each document.

Hash join is supported in **enterprise edition only**. To use hash join, a USE HASH hint must be specified on the right-hand side keyspace of ANSI JOIN. Using the same query as example 1:

```
SELECT DISTINCT route.destinationairport
FROM `travel-sample` airport JOIN `travel-sample` route USE HASH(build)
    ON airport.faa = route.sourceairport AND route.type = “route”
WHERE airport.type = “airport” AND airport.city = “San Jose” AND airport.country =
“United States”;
```

The USE HASH(build) hint directs the N1QL planner to perform hash join for the ANSI JOIN specified, and the right-hand side keyspace (“route”) is used on the build side of the hash join. Looking at the explain, there is a HashJoin operator:

```
{
  "#operator": "HashJoin",
  "build_aliases": [
    "route"
  ],
  "build_exprs": [
    "cover ((`route`.`sourceairport`))"
  ],
  "on_clause": "(((`airport`.`faa`) = cover ((`route`.`sourceairport`))) and (cover
((`route`.`type`)) = \"route\"))",
  "probe_exprs": [
    "(`airport`.`faa`)"
  ],
  "~child": {
    "#operator": "Sequence",
    "~children": [
      {
        "#operator": "IndexScan3",
        "as": "route",
        "covers": [
          "cover ((`route`.`sourceairport`))",
          "cover ((`route`.`destinationairport`))",
          "cover ((meta(`route`).`id`))"
        ],
        "filter_covers": {
          "cover ((`route`.`type`))": "route"
        },
        "index": "route_airports",
        "index_id": "f1f4b9fbe85e45fd",
        "keyspace": "travel-sample",
        "namespace": "default",
        "spans": [
          {
            "range": [
              {
                "inclusion": 0,
                "low": "null"
              }
            ]
          }
        ],
        "using": "gsi"
      }
    ]
  }
}
```

The child operator ("~child") for a HashJoin operator is always the build side of the hash join. For this query, it's an index scan on the right-hand side keyspace "route".

Note that for accessing the "route" document we can no longer use information from the left-hand side keyspace ("airport") for index search key (look at the "spans" information in the explain section above). Unlike nested-loop join, the index scan on "route" is no longer tied to an individual document from the left-hand side, and thus no value from the "airport" document can be used as search key for the index scan on "route".

The USE HASH(build) hint used in the query above directs the planner to use the right-hand side keyspace as the build side of the hash join. You can also specify USE HASH(probe) hint to direct the planner to use the right-hand side keyspace as the probe side of the hash join.

```
SELECT DISTINCT route.destinationairport
FROM `travel-sample` airport JOIN `travel-sample` route USE HASH(probe)
    ON airport.faa = route.sourceairport AND route.type = "route"
WHERE airport.type = "airport" AND airport.city = "San Jose" AND airport.country =
"United States";
```

Looking at the explain, you'll find the HashJoin operator:

```
{
  "#operator": "HashJoin",
  "build_aliases": [
    "airport"
  ],
  "build_exprs": [
    "(`airport`.`faa`)"
  ],
  "on_clause": "(((`airport`.`faa`) = cover ((`route`.`sourceairport`))) and (cover
((`route`.`type`)) = \"route\"))",
  "probe_exprs": [
    "cover ((`route`.`sourceairport`))"
  ],
  "~child": {
    "#operator": "Sequence",
    "~children": [
      {
        "#operator": "IntersectScan",
        "scans": [
          {
            "#operator": "IndexScan3",
            "as": "airport",
            "index": "airport_city_country",
            "index_id": "8e782fd1b124eec3",
            "index_projection": {
```

```
          "primary_key": true
        },
        "keyspace": "travel-sample",
        "namespace": "default",
        "spans": [
          {
            "exact": true,
            "range": [
              {
                "high": "\"San Jose\"",
                "inclusion": 3,
                "low": "\"San Jose\""
              },
              {
                "high": "\"United States\"",
                "inclusion": 3,
                "low": "\"United States\""
              }
            ]
          }
        ],
        "using": "gsi"
      },
      {
        "#operator": "IndexScan3",
        "as": "airport",
        "index": "airport_faa",
        "index_id": "c302afbf811470f5",
        "index_projection": {
          "primary_key": true
        },
        "keyspace": "travel-sample",
        "namespace": "default",
        "spans": [
          {
            "exact": true,
            "range": [
              {
                "inclusion": 0,
                "low": "null"
              }
            ]
          }
        ],
        "using": "gsi"
      }
    ]
  },
  {
```

```
        "#operator": "Fetch",
        "as": "airport",
        "keyspace": "travel-sample",
        "namespace": "default"
      }
    ]
  }
}
```

The child operator ("~child") for HashJoin is an intersect index scan on the left-hand side keyspace of the ANSI JOIN, "airport", followed by a fetch operator.

The USE HASH hint can only be specified on the right-hand side keyspace in an ANSI JOIN. Therefore depending on whether you want the right-hand side keyspace to be the build side or the probe side of a hash join, a USE HASH(build) or USE HASH(probe) hint should be specified on the right-hand side keyspace.

Hash join is only considered when USE HASH(build) or USE HASH(probe) hint is specified. Hash join requires equality join predicates to work. Nested-loop join requires an appropriate secondary index on the right-hand side keyspace, hash join does not (a primary index scan is an option for hash join). However, hash join does require more memory than nested-loop join since an in-memory hash table is required for hash join to work. In addition, hash join is considered a "blocking" operation, meaning the query engine must finish building the hash table before it can produce the first join result, thus for queries needing only the first few results quickly (e.g. with a LIMIT clause) hash join may not be the best fit.

If a USE HASH hint is specified, but a hash join cannot be generated successfully (e.g., lack of equality join predicates), then a nested-loop join will be considered.

Example 10: ANSI JOIN with multiple hints

You can now specify multiple hints for a keyspace on the right-hand side of an ANSI JOIN. For example, USE HASH hint can be used together with USE INDEX hint.

```
SELECT DISTINCT route.destinationairport
FROM `travel-sample` airport JOIN `travel-sample` route USE HASH(probe)
INDEX(route_airports)
    ON airport.faa = route.sourceairport AND route.type = "route"
WHERE airport.type = "airport" AND airport.city = "San Jose" AND airport.country =
"United States";
```

Note when multiple hints are used together, you only need to specify the "USE" keyword once, as in the example above.

USE HASH hint can also be combined with USE KEYS hint.

Example 11: ANSI JOIN with multiple joins

ANSI JOIN can be chained together. For example:

Required indexes (route_airports index same as example 1):

```
CREATE INDEX route_airports ON `travel-sample`(sourceairport, destinationairport)
WHERE type = "route";
CREATE INDEX airline_iata ON `travel-sample`(iata) WHERE type = "airline";
Optional index (same as example 1):
CREATE INDEX airport_city_country ON `travel-sample`(city, country) WHERE type =
"airport";
```

Query:

```
SELECT DISTINCT airline.name
FROM `travel-sample` airport INNER JOIN `travel-sample` route
    ON airport.faa = route.sourceairport AND route.type = "route"
  INNER JOIN `travel-sample` airline
    ON route.airline = airline.iata AND airline.type = "airline"
WHERE airport.type = "airport" AND airport.city = "San Jose"
 AND airport.country = "United States";
```

Since there is no USE HASH hint specified in the query the explain should show two NestedLoopJoin operators.

You can mix hash join with nested-loop join by adding USE HASH hint to any of the joins in a chain of ANSI JOINs.

```
SELECT DISTINCT airline.name
FROM `travel-sample` airport INNER JOIN `travel-sample` route
       ON airport.faa = route.sourceairport AND route.type = "route"
   INNER JOIN `travel-sample` airline USE HASH(build)
       ON route.airline = airline.iata AND airline.type = "airline"
WHERE airport.type = "airport" AND airport.city = "San Jose"
 AND airport.country = "United States";
```

or

```
SELECT DISTINCT airline.name
FROM `travel-sample` airport INNER JOIN `travel-sample` route USE HASH(probe)
    ON airport.faa = route.sourceairport AND route.type = "route"
  INNER JOIN `travel-sample` airline
    ON route.airline = airline.iata AND airline.type = "airline"
WHERE airport.type = "airport" AND airport.city = "San Jose"
  AND airport.country = "United States";
```

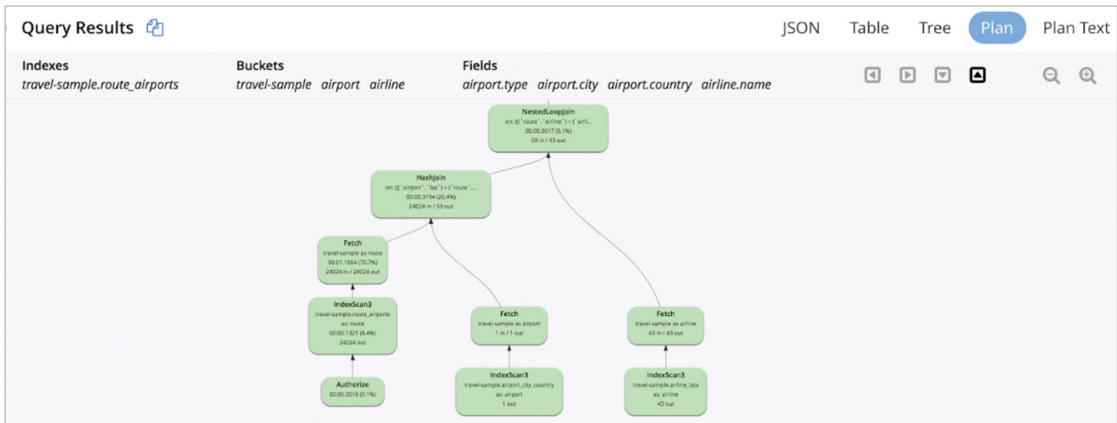
The visual explain for the last query is follows:

Query Editor

history (104/104)

```
1  SELECT DISTINCT airline.name
2  FROM `travel-sample` airport INNER JOIN `travel-sample` route USE HASH(probe)
3      ON airport.faa = route.sourceairport AND route.type = "route"
4    INNER JOIN `travel-sample` airline
5      ON route.airline = airline.iata AND airline.type = "airline"
6  WHERE airport.type = "airport" AND airport.city = "San Jose"
7   AND airport.country = "United States";
```

Execute | Explain | success | elapsed: 1.18s | execution: 1.18s | count: 9 | size: 235 | preferences



As mentioned before, N1QL only supports linear joins, i.e., the right-hand side of each join must be a keyspace.

Example 12: ANSI JOIN involving right-hand side arrays

Although ANSI JOIN comes from SQL standard, since Couchbase N1QL handles JSON documents and array is an important aspect of JSON, we extended ANSI JOIN support to arrays as well.

For examples in array handling please create a bucket "default" and insert the following documents:

```
INSERT INTO default (KEY,VALUE) VALUES("test11_ansijoin", {"c11": 1, "c12": 10, "a11":
[ 1, 2, 3, 4 ], "type": "left"}),
                 VALUES("test12_ansijoin", {"c11": 2, "c12": 20, "a11": [ 3, 3, 5, 10
], "type": "left"}),
                 VALUES("test13_ansijoin", {"c11": 3, "c12": 30, "a11": [ 3, 4, 20, 40
```

```
], "type": "left"}),
                 VALUES("test14_ansijoin", {"c11": 4, "c12": 40, "a11": [ 30, 30, 30 ],
"type": "left"});
INSERT INTO default (KEY,VALUE) VALUES("test21_ansijoin", {"c21": 1, "c22": 10, "a21":
[ 1, 10, 20], "a22": [ 1, 2, 3, 4 ], "type": "right"}),
                 VALUES("test22_ansijoin", {"c21": 2, "c22": 20, "a21": [ 2, 3, 30],
"a22": [ 3, 5, 10, 3 ], "type": "right"}),
                 VALUES("test23_ansijoin", {"c21": 2, "c22": 21, "a21": [ 2, 20, 30],
"a22": [ 3, 3, 5, 10 ], "type": "right"}),
                 VALUES("test24_ansijoin", {"c21": 3, "c22": 30, "a21": [ 3, 10, 30],
"a22": [ 3, 4, 20, 40 ], "type": "right"}),
                 VALUES("test25_ansijoin", {"c21": 3, "c22": 31, "a21": [ 3, 20, 40],
"a22": [ 4, 3, 40, 20 ], "type": "right"}),
                 VALUES("test26_ansijoin", {"c21": 3, "c22": 32, "a21": [ 4, 14, 24],
"a22": [ 40, 20, 4, 3 ], "type": "right"}),
                 VALUES("test27_ansijoin", {"c21": 5, "c22": 50, "a21": [ 5, 15, 25],
"a22": [ 1, 2, 3, 4 ], "type": "right"}),
                 VALUES("test28_ansijoin", {"c21": 6, "c22": 60, "a21": [ 6, 16, 26],
"a22": [ 3, 3, 5, 10 ], "type": "right"}),
                 VALUES("test29_ansijoin", {"c21": 7, "c22": 70, "a21": [ 7, 17, 27],
"a22": [ 30, 30, 30 ], "type": "right"}),
                 VALUES("test30_ansijoin", {"c21": 8, "c22": 80, "a21": [ 8, 18, 28],
"a22": [ 30, 30, 30 ], "type": "right"});
```

Then create the following indexes:

```
CREATE INDEX default_ix_left on default(c11, DISTINCT a11) WHERE type = "left";
CREATE INDEX efault_ix_right on default(c21, DISTINCT a21) WHERE type = "right";
```

When the join predicate involves an array on the right-hand side of ANSI JOIN, you need to create an array index on the right-hand side keyspace.

Query:

```
SELECT b1.c11, b2.c21, b2.c22
FROM default b1 JOIN default b2
    ON b2.c21 = b1.c11 AND ANY v IN b2.a21 SATISFIES v = b1.c12 END AND b2.type =
"right"
WHERE b1.type = "left";
```

Note that part of the join condition is an ANY clause which specifies that the left-hand side field b1.c12 can match any element of the right-hand side array b2.a21. For this join to work properly, we need an array index on b2.a21, e.g., default_ix_right index created above.

The explain plan shows a NestedLoopJoin, with child operator being a distinct scan on the array index default_ix_right.

```
{
  "#operator": "NestedLoopJoin",
  "alias": "b2",
  "on_clause": "((((`b2`.`c21`) = (`b1`.`c11`)) and any `v` in (`b2`.`a21`) satisfies
(`v` = (`b1`.`c12`)) end) and ((`b2`.`type`) = \"right\"))",
  "~child": {
    "#operator": "Sequence",
    "~children": [
      {
        "#operator": "DistinctScan",
        "scan": {
          "#operator": "IndexScan3",
          "as": "b2",
          "index": "default_ix_right",
          "index_id": "ef4e7fa33f33dce",
          "index_projection": {
            "primary_key": true
          },
          "keyspace": "default",
          "namespace": "default",
          "nested_loop": true,
          "spans": [
            {
              "exact": true,
              "range": [
                {
                  "high": "(`b1`.`c11`)",
                  "inclusion": 3,
                  "low": "(`b1`.`c11`)"
                },
                {
                  "high": "(`b1`.`c12`)",
                  "inclusion": 3,
                  "low": "(`b1`.`c12`)"
                }
              ]
            }
          ],
          "using": "gsi"
        }
      },
      {
        "#operator": "Fetch",
        "as": "b2",
        "keyspace": "default",
        "namespace": "default",
        "nested_loop": true
```

```
      }
    ]
  }
}
```

Example 13: ANSI JOIN involving left-hand side arrays

If ANSI JOIN involves an array on the left-hand side, then there are two options for performing the join.

Option 1: use UNNEST

Use UNNEST clause to flatten the left-hand side array first before performing the join.

```
SELECT b1.c11, b2.c21, b2.c22
FROM default b1 UNNEST b1.a11 AS ba1
    JOIN default b2 ON ba1 = b2.c21 AND b2.type = “right”
WHERE b1.c11 = 2 AND b1.type = “left”;
```

After the UNNEST the array becomes individual fields, and the subsequent join is just like a “regular” ANSI JOIN with fields from both sides.

Option 2: use IN-clause

Alternatively, use IN-clause as join condition.

```
SELECT b1.c11, b2.c21, b2.c22
FROM default b1 JOIN default b2
    ON b2.c21 IN b1.a11 AND b2.type = “right”
WHERE b1.c11 = 2 AND b1.type = “left”;
```

The IN-clause is satisfied when any element of the array on the left-hand side keyspace (“b1.a11”) matches the right-hand side field (“b2.c21”).

Note that there is a semantics difference between the two options. When there are duplicates in the array, the UNNEST option does not care about duplicates and will flatten the left-hand side documents to as many documents as number of elements in the array, thus may produce duplicated results; the IN-clause option will not produce duplicated results if there are duplicated elements in the array. In addition, when LEFT OUTER JOIN is performed, there may be different number of preserved documents due to the flattening of the array with the UNNEST option. Thus the user is advised to pick the option that reflect the semantics needed for the query.

Example 14: ANSI JOIN involving arrays on both sides

Although uncommon, it is possible to perform an ANSI JOIN when both sides of the join are arrays. In such cases, you can use a combination of the techniques described above. Use array index to handle array on the right-hand side, and use either UNNEST option or IN-clause option to handle array on the left-hand side.

Option 1: use UNNEST clause

```
SELECT b1.c11, b2.c21, b2.c22
FROM default b1 UNNEST b1.a11 AS ba1
    JOIN default b2 ON b2.c21 = b1.c11 AND ANY v IN b2.a21 SATISFIES v = ba1 END AND
b2.type = “right”
WHERE b1.type = “left”;
```

Option 2: use IN-clause

```
SELECT b1.c11, b2.c21, b2.c22
FROM default b1 JOIN default b2
    ON b2.c21 = b1.c11 AND ANY v IN b2.a21 SATISFIES v IN b1.a11 END AND b2.type =
“right”
WHERE b1.type = “left”;
```

Again the two options are not semantically identical, and may give different results. Pick the option that reflects the semantics desired.

Example 15: lookup join migration

N1QL will continue to support lookup join and index join for backward compatibility, however, you cannot mix ANSI JOIN with lookup join or index join in the same query. You can convert your existing queries from using lookup join and index join to the ANSI JOIN syntax. This example shows you how to convert a lookup join into ANSI JOIN syntax.

Create the following index to speed up the query (same as example 1):

```
CREATE INDEX route_airports ON `travel-sample`(sourceairport, destinationairport) WHERE
type = “route”;
```

This is a query using lookup join syntax (note the ON KEYS clause):

```
SELECT airline.name
FROM `travel-sample` route JOIN `travel-sample` airline
    ON KEYS route.airlineid
WHERE route.type = “route” AND route.sourceairport = “SFO” AND route.destinationairport
= “JFK”;
```

In lookup join the left-hand side of the join ("route") needs to produce document keys for the right-hand side of the join ("airline"), this is achieved by the ON KEYS clause. The join condition (which is implied from the syntax) is route.airlineid = meta(airline).id, thus the same query can be specified using ANSI JOIN syntax:

```
SELECT airline.name
FROM `travel-sample` route JOIN `travel-sample` airline
    ON route.airlineid = meta(airline).id
WHERE route.type = "route" AND route.sourceairport = "SFO" AND route.destinationairport
= "JFK";
```

In this example the ON KEYS clause contain a single document key. It's possible for the ON KEYS clause to contain an array of document keys, in which case the converted ON clause will be in the form of an IN clause instead of an equality clause. Let's assume each route document has an array of document keys for airline, then the original ON KEYS clause:

```
    ON KEYS route.airlineids
```

can be converted to:

```
    ON meta(airline).id IN route.airlineids
```

Example 16: index join migration

This example shows you how to convert an index join into ANSI JOIN syntax.

Required index (same as example 3):

```
CREATE INDEX route_airlineid ON `travel-sample`(airlineid) WHERE type = "route";
Optional index (same as example 3):
```

```
CREATE INDEX airline_name ON `travel-sample`(name) WHERE type = "airline";
Query using index join syntax (note the ON KEY … FOR … clause):
```

```
SELECT count(*)
FROM `travel-sample` airline JOIN `travel-sample` route
    ON KEY route.airlineid FOR airline
WHERE airline.type = "airline" AND route.type = "route" AND airline.name = "United
Airlines";
```

In index join the document key for left-hand side ("airline") is used to probe an index on an expression ("route.airlineid" which appears in the ON KEY clause) from the right-hand side ("route") that corresponds to the document key for the left-hand side ("airline" which appears in the FOR clause). The join condition (implied from syntax) is route.airlineid = meta(airline).id, thus the same query can be specified using ANSI JOIN syntax:

```
SELECT count(*)
FROM `travel-sample` airline JOIN `travel-sample` route
    ON route.airlineid = meta(airline).id
WHERE airline.type = "airline" AND route.type = "route" AND airline.name = "United
Airlines";
```

Example 17: ANSI NEST

Couchbase N1QL supports NEST operation. Previously NEST can be done using lookup nest or index nest, similar to lookup join and index join, respectively. With ANSI JOIN support, NEST operation can also be done using similar syntax, i.e., using ON clause instead of ON KEYS (lookup nest) or ON KEY … FOR … (index nest) clauses. This new variant is referred to as ANSI NEST.

Required index (route_airports index same as example 1, route_airline_distance index same as example 4):

```
CREATE INDEX route_airports ON `travel-sample`(sourceairport, destinationairport) WHERE
type = "route";
CREATE INDEX route_airline_distance ON `travel-sample`(airline, distance) WHERE type =
"route";
```

Optional index:

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

Query:

```
SELECT airline.name, ARRAY {"destination": r.destinationairport} FOR r in route END as
destinations
FROM `travel-sample` airline NEST `travel-sample` route
    ON airline.iata = route.airline AND route.type = "route" AND route.sourceairport =
"SFO"
WHERE airline.type = "airline" AND airline.country = "United States";
```

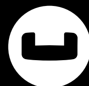
As you can see the syntax for ANSI NEST is very similar to that of ANSI JOIN. There is one peculiar property for nest though. By definition the nest operation creates an array of all matching right-hand side document for each left-hand side document, which means the reference to the right-hand side keyspace, "route" in this query, has different meaning depending on where the reference is. The ON-clause is evaluated as part of the NEST operation, and thus references to "route" is referencing a single document. In contrast, references in the projection

clause, or the WHERE clause, are evaluated after the NEST operation, and thus references to “route” means the nested array, thus it should be treated as an array. Notice the projection clause of the query above has an ARRAY construct with a FOR clause to access each individual document within the array (i.e., the reference to “route” is now in an array context).

Summary

ANSI JOIN provides much more flexibility in join operations in Couchbase N1QL, compared to previously supported lookup join and index join, both of which requires joining on document key only. The examples above show various ways you can use ANSI JOIN in queries. Since ANSI JOIN is widely used in relational world, the support for ANSI JOIN in Couchbase N1QL should make it much easier to migrate applications from a relational database to Couchbase N1QL.

Grouping and Aggregate Performance



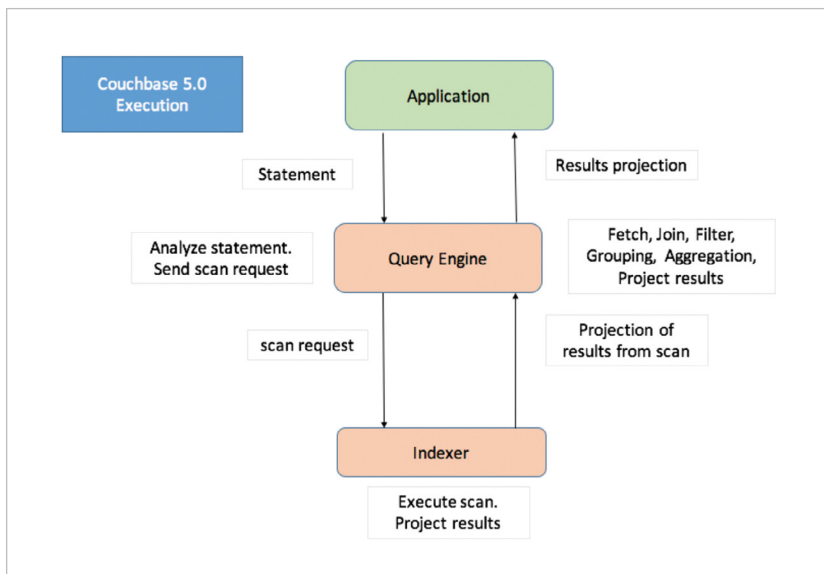
UNDERSTANDING INDEX GROUPING AND AGGREGATION IN COUCHBASE N1QL QUERY

Authors: Sitaram Vemulapalli, Principal Engineer, Couchbase R&D
Deepkaran Salooja, Principal Engineer, Couchbase R&D

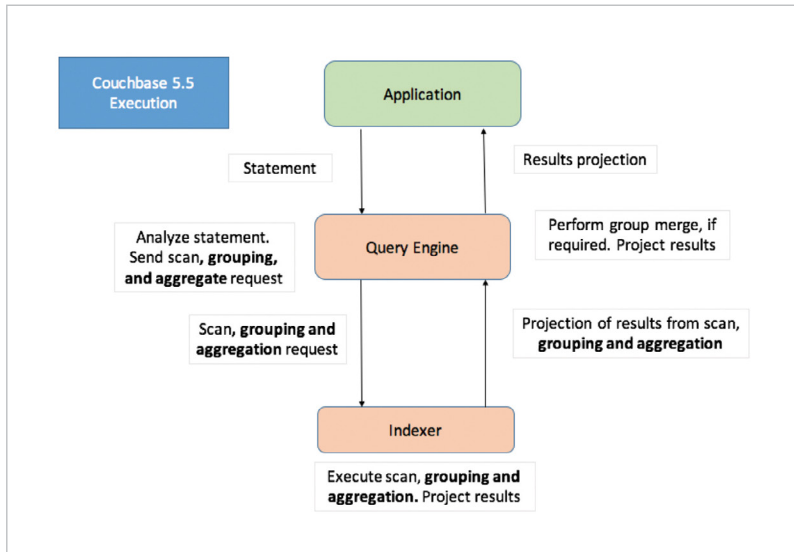
Couchbase N1QL is a modern query processing engine designed to provide SQL for JSON on distributed data with a flexible data model. Modern databases are deployed on massive clusters. Using JSON provides a flexible data mode. N1QL supports enhanced SQL for JSON to make query processing easier.

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 data 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.

Prior to Couchbase 5.5, even when a query with GROUP BY and/or aggregates is covered by an index, the query fetched all relevant data from the indexer and performed grouping/aggregation of the data within the query engine.



In Couchbase 5.5 query planner enhanced to intelligently requests the indexer to perform grouping and aggregation in addition to range scan **for covering index**. The Indexer has been enhanced to perform grouping, COUNT(), SUM(), MIN(), MAX(), AVG(), and related operations on-the-fly.



This requires no changes to the user query, but a good index design to cover the query and order the index keys is required. Not every query will benefit from this optimization, and not every index can accelerate every grouping and aggregation operation. Understanding the right patterns will help you design your indexes and queries. Index grouping and aggregation on global secondary index is supported with both storage engines: Standard GSI and Memory Optimized GSI (MOI). Index grouping and aggregation is supported in Enterprise Edition only.

This reduction step of performing the GROUP BY and Aggregation by the indexer reduces the amount of data transfer and disk I/O, resulting in:

- Improved query response time
- Improved resource utilization
- Low latency
- High scalability
- Low Total Cost of Ownership

Performance

The Index grouping and aggregations can improve query performance by orders of magnitude and reduce the latencies drastically. The following table list few sample query latency measurements.

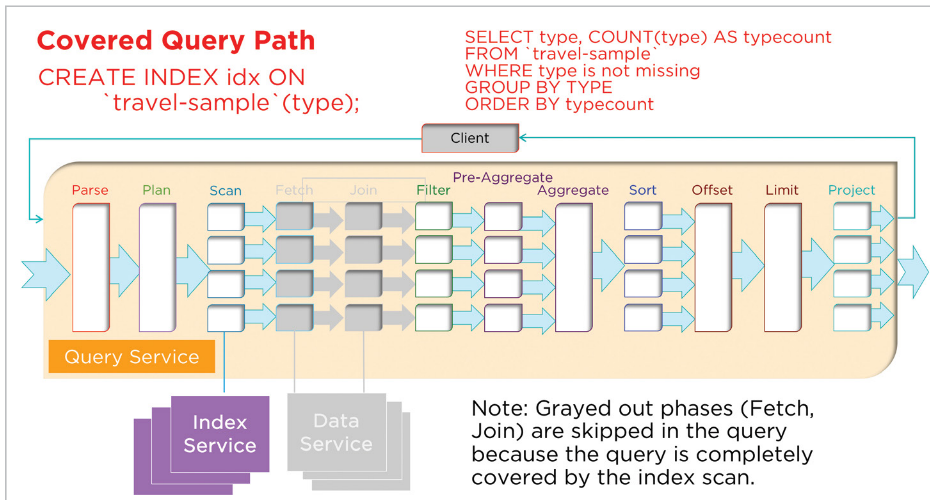
```
Index : CREATE INDEX idx_ts_type_country_city ON `travel-sample` (type, country, city);
```

Query	Description	5.0 Latencies	5.5 Latencies
`SELECT t.type, COUNT(type) AS cnt FROM `travel-sample` AS t WHERE t.type IS NOT NULL GROUP BY t.type;`	• GROUP BY leading index key • Aggregation	230ms	13ms
`SELECT t.type, COUNT(1) AS cnt, COUNT(DISTINCT city) AS cntdcity FROM `travel-sample` AS t WHERE t.type IN [“hotel”,”airport”] GROUP BY t.type, t.country;`	• GROUP BY multiple leading index keys • Multiple Aggregates • Distinct Aggregate	40ms	7ms
`SELECT t.country, COUNT(city) AS cnt FROM `travel-sample` AS t WHERE t.type = “airport” GROUP BY t.country;`	• GROUP BY first non-equality leading index key • Aggregation	25ms	3ms
`SELECT t.city, cnt FROM `travel-sample` AS t WHERE t.type IS NOT NULL GROUP BY t.city LETTING cnt = COUNT(city) HAVING cnt > 0 ;`	• GROUP BY non-leading index key • LETTING clause • HAVING clause	300ms	160ms

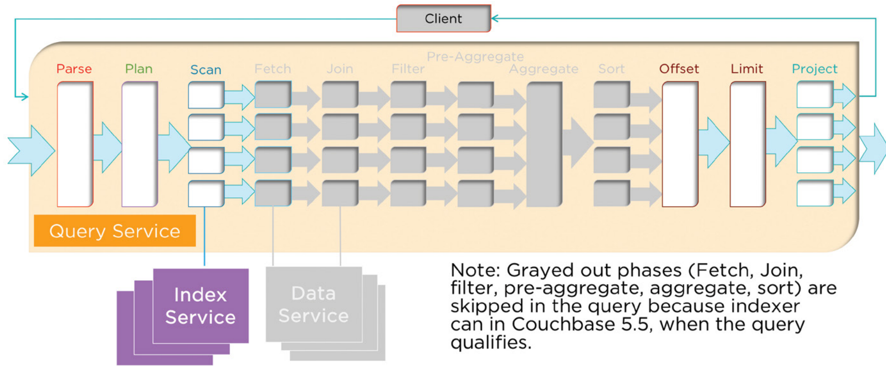
Index Grouping and Aggregation Overview

The above figure shows all the possible phases a SELECT query goes through to return the results. 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 range scans in the query plan and passed to Indexer.

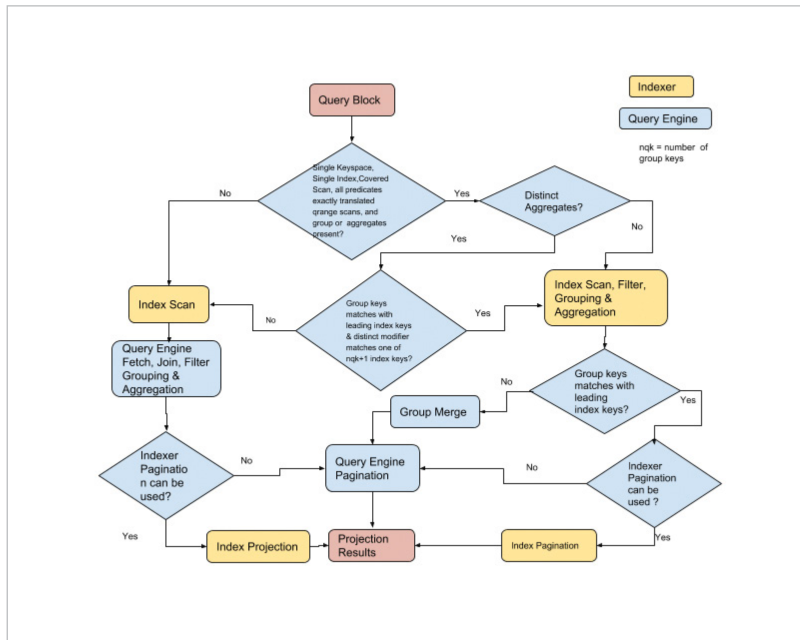
If the query doesn't have JOINs and is covered by index, both Fetch and Join phases can be eliminated.



When all predicates are exactly translated to range scans Filter phase also can be eliminated. In that situation Scan and Aggregates are side by side, and since indexer has ability to do aggregation that phase can be done on indexer node. In some cases Sort, Offset, Limit phases can also be done indexer node.



The following flow chart describes how query planner decides to perform index aggregation for each query block of the query. If the index aggregation is not possible aggregations are done in query engine.



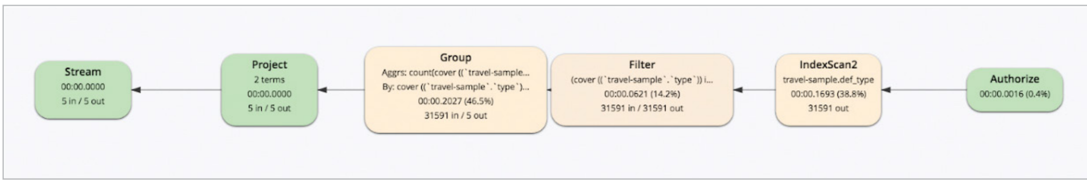
For example, let's compare the previous vs. current performance of using GROUP BY and examine the EXPLAIN plan of the following query that uses an index defined on the Couchbase `travel-sample` bucket:

```
CREATE INDEX `def_type` ON `travel-sample`(`type`)
```

Consider the query:

```
SELECT type, COUNT(type)
FROM `travel-sample`
WHERE type IS NOT MISSING
GROUP BY type;
```

Before Couchbase version 5.5, this query engine fetched relevant data from the indexer and grouping and aggregation of the data is done within query engine. This simple query takes about 250 ms.



Now, in Couchbase version 5.5, this query uses the same def_type index, but executes in under 20 ms. In the explain below, you can see fewer steps and the lack of the grouping step after the index scan because the index scan step does the grouping and aggregation as well.

As the data and query complexity grows, the performance benefit (both latency and throughput) will grow as well.

Understanding EXPLAIN of Index Grouping and Aggregation

Looking at the explain of the query:

```
EXPLAIN SELECT type, COUNT(type) FROM `travel-sample` WHERE type IS NOT MISSING GROUP
BY type;
```

```
1.      {
2.        "plan": {
3.          "#operator": "Sequence",
4.          "~children": [
5.            {
6.              "#operator": "IndexScan3",
7.              "covers": [
8.                "cover ((`travel-sample`.`type`))",
9.                "cover ((meta(`travel-sample`).`id`))",
10.               "cover (count(cover ((`travel-sample`.`type`))))"
11.             ],
12.             "index": "def_type",
13.             "index_group_aggs": {
14.               "aggregates": [
15.                 {
16.                   "aggregate": "COUNT",
17.                   "depends": [
18.                     0
19.                   ],
20.                   "expr": "cover ((`travel-sample`.`type`))",
21.                   "id": 2,
22.                   "keypos": 0
23.                 }
24.               ],
25.               "depends": [
26.                 0
27.               ],
28.               "group": [
29.                 {
30.                   "depends": [
31.                     0
32.                   ],
33.                   "expr": "cover ((`travel-sample`.`type`))",
34.                   "id": 0,
35.                   "keypos": 0
36.                 }
37.               ]
38.             },
39.             "index_id": "b948c92b44c2739f",
40.             "index_projection": {
41.               "entry_keys": [
```

```
42.                 0,
43.                 2
44.               ]
45.             },
46.             "keyspace": "travel-sample",
47.             "namespace": "default",
48.             "spans": [
49.               {
50.                 "exact": true,
51.                 "range": [
52.                   {
53.                     "inclusion": 1,
54.                     "low": "null"
55.                   }
56.                 ]
57.               }
58.             ],
59.             "using": "gsi"
60.           },
61.           {
62.             "#operator": "Parallel",
63.             "~child": {
64.               "#operator": "Sequence",
65.               "~children": [
66.                 {
67.                   "#operator": "InitialProject",
68.                   "result_terms": [
69.                     {
70.                       "expr": "cover ((`travel-sample`.`type`))"
71.                     },
72.                     {
73.                       "expr": "cover (count(cover ((`travel-sample`.`type`))))"
74.                     }
75.                   ]
76.                 },
77.                 {
78.                   "#operator": "FinalProject"
79.                 }
80.               ]
81.             }
82.           }
83.         ]
84.       },
85.       "text": "SELECT type, COUNT(type) FROM `travel-sample` WHERE type IS
        NOT MISSING GROUP BY type;"
86.     }
```

You will see “index_group_aggs” in the IndexScan section (i.e “#operator”: “IndexScan3”). If “index_group_aggs” is MISSING then query service is performing grouping and aggregation. If present query is using Index grouping and aggregation and it has all relevant information indexer required for grouping and aggregation. The following table describe how to interpret the various information of index_group_aggs object.

Field Name	Description	Line numbers from Example	Explain Text in Example
`aggregates`	Array of Aggregate objects, and each object represents one aggregate. The absence of this item means only group by is present in the query.	`14-24`	`aggregates`
`aggregate`	Aggregate operation (`MAX/MIN/SUM/COUNT/COUNTN`).	`16`	`COUNT`
`distinct`	Aggregate modifier is DISTINCT	`-`	`False(When true only it appears)`
`depends`	List of index key positions (starting with 0) the aggregate expression depends on.	`17-19`	`0` (because type is 0th index key of def_type index)
`expr`	aggregate expression	20	cover ((`travel-sample`.`type`))
`id`	Unique ID given internally and will be used in `index_projection`	`21`	`2`

Field Name	Description	Line numbers from Example	Explain Text in Example
keypos	Indicator to that tells use expression at the index key position or from the expr field. • A value > -1 means the aggregate expression is exactly matches the corresponding index key position(starting with 0). • A value of -1 means the] aggregate expression does not exactly match with the index key position and use expression from expr field. means the aggregate expression is exactly matches the corresponding index key position(starting with 0). A value of -1 means the] aggregate expression does not exactly match with the index key position and use expression from expr field.	22	0 (because type is 0th index key of def_type index)

Field Name	Description	Line numbers from Example	Explain Text in Example
`depends`	List of index key positions the groups/aggregates expressions depends on (consolidated list)	`25-27`	`0`
`group`	Array of GROUP BY objects, and each object represents one group key. The absence of this item means there is no GROUP BY clause present in the query.	`28-37`	`group`
`depends`	List of index key positions (starting with 0) the group expression depends on.	`30-32`	`0` (because type is 0th key of index key of def_type index)
`expr`	group expression.	33	cover ((`travel-sample`.`type`))
`id`	Unique ID given internally and will be used in `index_projection`.	`34`	`0`

Field Name	Description	Line numbers from Example	Explain Text in Example
`keypos`	Indicator to that tells use expression at the index key position or from the expr field. • A value > -1 means the group expression is exactly matches the corresponding index key position (starting with 0). • A value of -1 means the group key does not exactly match with the index key position and use expression from expr field.	`35`	`0` (because type is 0th index key of def_type index)

The covers field is array and it has all the index keys, document key(META().id), group keys expressions that are not exactly matched with index keys (sorted by id), aggregates sorted by id. Also “Index_projection” will have all the group/aggregate ids.

```
"covers": [
    "cover ((`travel-sample`.`type`))",                ← Index key (0)
    "cover ((meta(`travel-sample`).`id`))",            ← document key (1)
    "cover (count(cover ((`travel-sample`.`type`))))"  ←  aggregate (2)
  ]
```

In above case group expression `type` is same Index key of index `def_type`. It is not included twice.

Details of Index Grouping and Aggregation

We will use examples to show how Index grouping and aggregations works. To follow the examples please create a bucket “default” and insert the following documents:

```
INSERT INTO default (KEY,VALUE)
VALUES ("ga0001", {"c0":1, "c1":10, "c2":100, "c3":1000, "c4":10000, "a1":[{"id":1},
{"id":1}, {"id":2}, {"id":3}, {"id":4}, {"id":5}]}),
VALUES ("ga0002", {"c0":1, "c1":20, "c2":200, "c3":2000, "c4":20000, "a1":[{"id":1},
{"id":1}, {"id":2}, {"id":3}, {"id":4}, {"id":5}]}),
VALUES ("ga0003", {"c0":1, "c1":10, "c2":300, "c3":3000, "c4":30000, "a1":[{"id":1},
{"id":1}, {"id":2}, {"id":3}, {"id":4}, {"id":5}]}),
VALUES ("ga0004", {"c0":1, "c1":20, "c2":400, "c3":4000, "c4":40000, "a1":[{"id":1},
{"id":1}, {"id":2}, {"id":3}, {"id":4}, {"id":5}]}),
VALUES ("ga0005", {"c0":2, "c1":10, "c2":100, "c3":5000, "c4":50000, "a1":[{"id":1},
{"id":1}, {"id":2}, {"id":3}, {"id":4}, {"id":5}]}),
VALUES ("ga0006", {"c0":2, "c1":20, "c2":200, "c3":6000, "c4":60000, "a1":[{"id":1},
{"id":1}, {"id":2}, {"id":3}, {"id":4}, {"id":5}]}),
VALUES ("ga0007", {"c0":2, "c1":10, "c2":300, "c3":7000, "c4":70000, "a1":[{"id":1},
{"id":1}, {"id":2}, {"id":3}, {"id":4}, {"id":5}]}),
VALUES ("ga0008", {"c0":2, "c1":20, "c2":400, "c3":8000, "c4":80000, "a1":[{"id":1},
{"id":1}, {"id":2}, {"id":3}, {"id":4}, {"id":5}]});
```

Example 1: Group by leading index keys

Let consider the following query and index:

```
SELECT d.c0 AS c0, d.c1 AS c1, SUM(d.c3) AS sumc3,
AVG(d.c4) AS avgc4, COUNT(DISTINCT d.c2) AS dcountc2
FROM default AS d
WHERE d.c0 > 0
GROUP BY d.c0, d.c1
ORDER BY d.c0, d.c1
OFFSET 1
LIMIT 2;
```

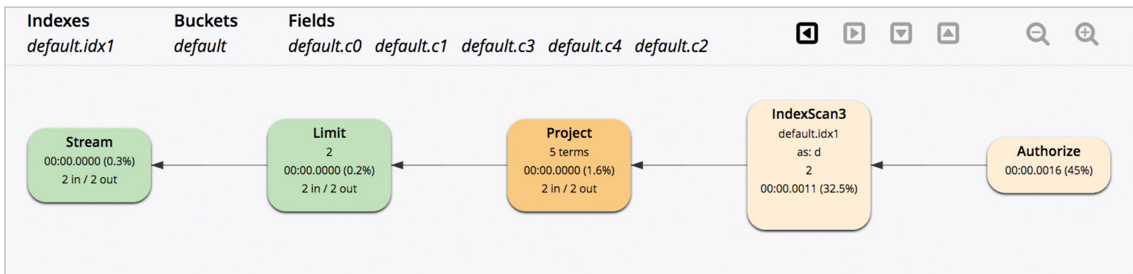
Required Index:

```
CREATE INDEX idx1 ON default(c0,c1,c2,c3,c4);
```

The query has GROUP BY and multiple aggregates, some of aggregates has DISTINCT modifier. The query can be covered by index idx1 and the predicate (d.c0 > 0) can be converted into exact range scan and passed it to index scan. So, the index and query combination qualifies Index grouping and aggregations.

Indexes are naturally ordered and grouped by the order of the index key definition. In the above query, the GROUP BY keys (d.c0, d.c1) exactly matches with the leading keys (c0, c1) of the index. Therefore, index has each group data together, indexer will produce one row per group i.e. Full aggregation. Also, query has aggregate that has DISTINCT modifier and it exactly matches with one of the index keys with position less than or equal to number of group keys plus one (i.e. there 2 group keys, DISTINCT modifier can be any one of index key at position 0,1,2 because index key followed by group keys and DISTINCT modifier can applied without sort). Therefore, the query above is suitable for indexer to handle grouping and aggregation.

If group by missing one of the leading index key and there is equality predicate, then special optimization is done by treating the index key implicitly present in group keys and determine if Full aggregation is possible or not. For partition index the all the partition keys needs to present in the group keys to generate Full aggregations.




```
1.      {
2.        "plan": {
3.          "#operator": "Sequence",
4.          "~children": [
5.            {
6.              "#operator": "Sequence",
7.              "~children": [
8.                {
9.                  "#operator": "IndexScan3",
10.                 "as": "d",
11.                 "covers": [
12.                   "cover ((`d`.`c0`))",
13.                   "cover ((`d`.`c1`))",
14.                   "cover ((`d`.`c2`))",
15.                   "cover ((`d`.`c3`))",
16.                   "cover ((`d`.`c4`))",
```

```
17.                     "cover ((meta(`d`).`id`))",
18.                     "cover (count(distinct cover ((`d`.`c2`))))",
19.                     "cover (countn(cover ((`d`.`c4`))))",
20.                     "cover (sum(cover ((`d`.`c3`))))",
21.                     "cover (sum(cover ((`d`.`c4`))))"
22.                   ],
23.                   "index": "idx1",
24.                   "index_group_aggs": {
25.                     "aggregates": [
26.                       {
27.                         "aggregate": "COUNT",
28.                         "depends": [
29.                           2
30.                         ],
31.                         "distinct": true,
32.                         "expr": "cover ((`d`.`c2`))",
33.                         "id": 6,
34.                         "keypos": 2
35.                       },
36.                       {
37.                         "aggregate": "COUNTN",
38.                         "depends": [
39.                           4
40.                         ],
41.                         "expr": "cover ((`d`.`c4`))",
42.                         "id": 7,
43.                         "keypos": 4
44.                       },
45.                       {
46.                         "aggregate": "SUM",
47.                         "depends": [
48.                           3
49.                         ],
50.                         "expr": "cover ((`d`.`c3`))",
51.                         "id": 8,
52.                         "keypos": 3
53.                       },
54.                       {
55.                         "aggregate": "SUM",
56.                         "depends": [
57.                           4
58.                         ],
59.                         "expr": "cover ((`d`.`c4`))",
60.                         "id": 9,
61.                         "keypos": 4
62.                       }
63.                     ],
64.                     "depends": [
65.                       0,
```

```
66.                     1,
67.                     2,
68.                     3,
69.                     4
70.                   ],
71.                   "group": [
72.                     {
73.                       "depends": [
74.                         0
75.                       ],
76.                       "expr": "cover ((`d`.`c0`))",
77.                       "id": 0,
78.                       "keypos": 0
79.                     },
80.                     {
81.                       "depends": [
82.                         1
83.                       ],
84.                       "expr": "cover ((`d`.`c1`))",
85.                       "id": 1,
86.                       "keypos": 1
87.                     }
88.                   ]
89.                 },
90.                 "index_id": "d06df7c5d379cd5",
91.                 "index_order": [
92.                   {
93.                     "keypos": 0
94.                   },
95.                   {
96.                     "keypos": 1
97.                   }
98.                 ],
99.                 "index_projection": {
100.                  "entry_keys": [
101.                    0,
102.                    1,
103.                    6,
104.                    7,
105.                    8,
106.                    9
107.                  ]
108.                },
109.                "keyspace": "default",
110.                "limit": "2",
111.                "namespace": "default",
112.                "offset": "1",
113.                "spans": [
```

```
114.                     {
115.                       "exact": true,
116.                       "range": [
117.                         {
118.                           "inclusion": 0,
119.                           "low": "0"
120.                         }
121.                       ]
122.                     }
123.                   ],
124.                   "using": "gsi"
125.                 },
126.                 {
127.                   "#operator": "Parallel",
128.                   "maxParallelism": 1,
129.                   "~child": {
130.                     "#operator": "Sequence",
131.                     "~children": [
132.                       {
133.                         "#operator": "InitialProject",
134.                         "result_terms": [
135.                           {
136.                             "as": "c0",
137.                             "expr": "cover ((`d`.`c0`))"
138.                           },
139.                           {
140.                             "as": "c1",
141.                             "expr": "cover ((`d`.`c1`))"
142.                           },
143.                           {
144.                             "as": "sumc3",
145.                             "expr": "cover (sum(cover ((`d`.`c3`))))"
146.                           },
147.                           {
148.                             "as": "avgc4",
149.                             "expr": "(cover (sum(cover ((`d`.`c4`)))) / cover (count-
n(cover ((`d`.`c4`)))))"
150.                           },
151.                           {
152.                             "as": "dcountc2",
153.                             "expr": "cover (count(distinct cover ((`d`.`c2`))))"
154.                           }
155.                         ]
156.                       },
157.                       {
158.                         "#operator": "FinalProject"
159.                       }
160.                     ]
```

```
161.                }
162.              }
163.            ]
164.          },
165.          {
166.            "#operator": "Limit",
167.            "expr": "2"
168.          }
169.        ]
170.      },
171.      "text": "SELECT d.c0 AS c0, d.c1 AS c1, SUM(d.c3) AS sumc3, AVG(d.c4) AS
    avgc4, COUNT(DISTINCT d.c2) AS dcountc2 FROM default AS d\nWHERE d.c0 > 0 GROUP BY
    d.c0, d.c1 ORDER BY d.c0, d.c1 OFFSET 1  LIMIT 2;"
172.    }
```

- The “index_group_aggs” (lines 24-89) in the IndexScan section (i.e “#operator”: “IndexScan3”) shows query using index grouping and aggregations.
- If query uses index grouping and aggregation the predicates are exactly converted to range scans and passed to index scan as part of spans, so there will not be any Filter operator in the explain.
- As group by keys exactly match the leading index keys, indexer will produce full aggregations. Therefore, we also eliminate grouping in query service (There is no InitialGroup, IntermediateGroup, FinalGroup operators in the explain).
- Indexer projects “index_projection” (lines 99-107) including all group keys and aggregates.
- Query ORDER BY matches with leading index keys and GROUP BY is on leading index keys we can use index order. This can be found in explain (lines 91-98) and will not use “#operator”: “Order” between line 164-165.
- As query can use index order and there is no HAVING clause in the query the “offset” and “limit” values can be passed to indexer.
- This can be found at line 112, 110. The “offset” can be applied only once you will not see “#operator”: “Offset” between line 164-165, But re-applying “limit” is no-op. This can be seen at line 165-168.
- Query contains AVG(x) it has been rewritten as SUM(x)/COUNTN(x). The COUNTN(x) only counts when x is numeric value.

Example 2: Group by leading index keys, LETTING, HAVING

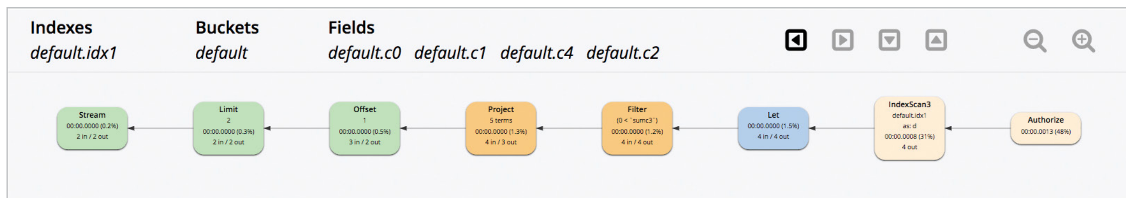
Let consider the following query and index:

```
SELECT d.c0 AS c0, d.c1 AS c1, sumc3 AS sumc3,
AVG(d.c4) AS avgc4, COUNT(DISTINCT d.c2) AS dcountc2
FROM default AS d
WHERE d.c0 > 0
GROUP BY d.c0, d.c1
LETTING sumc3 = SUM(d.c3)
HAVING sumc3 > 0
ORDER BY d.c0, d.c1
OFFSET 1
LIMIT 2;
```

Required Index:

```
CREATE INDEX idx1 ON default(c0,c1,c2,c3,c4);
```

The above query is similar to Example 1 but it has LETTING, HAVING clause. Indexer will not be able to handle these and thus LETTING and HAVING clauses are applied in query service after grouping and aggregations. Therefore you see Let, Filter operators after IndexScan3 in execution tree. Having clause is filter and further eliminates items thus “offset”, “limit” can’t be pushed to indexer and need to be applied in query service, but we still can use index order.



Example 3: Group by non-leading index keys

Let consider the following query and index:

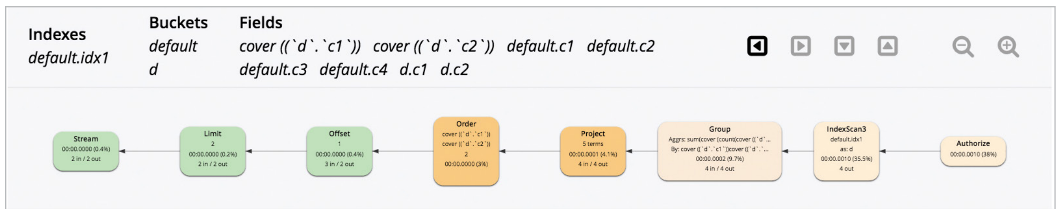
```
SELECT d.c1 AS c1, d.c2 AS c2, SUM(d.c3) AS sumc3,
AVG(d.c4) AS avgc4, COUNT(d.c2) AS countc2
FROM default AS d
WHERE d.c0 > 0
GROUP BY d.c1, d.c2
ORDER BY d.c1, d.c2
OFFSET 1
LIMIT 2;
```

Required Index:

```
CREATE INDEX idx1 ON default(c0,c1,c2,c3,c4);
```

The query has GROUP BY and multiple aggregates. The query can be covered by index idx1 and the predicate (d.c0 > 0) can be converted into exact range scan and passed it to index scan. So, the index and query combination qualifies Index grouping and aggregations.

In the above query, the GROUP BY keys (d.c1, d.c2) do NOT match the leading keys (c0, c1) of the index. The groups are scattered across the index. Therefore, indexer will produce multiple rows per each group i.e. Partial aggregation. In case of partial aggregation query service does group merge, query can't use index order or push "offset", "limit" to indexer. In case of partial aggregation if any aggregate has DISTINCT modifier index grouping and aggregation is not possible. The query above is suitable for indexer to handle grouping and aggregation.



The above graphical execution tree shows index scan (IndexScan3) performing scan and index grouping aggregations. The results from the index scan are grouped again and projected.

Let's look at the text based explain:

```
1.     {
2.       "plan": {
3.         "#operator": "Sequence",
4.         "~children": [
5.           {
6.             "#operator": "Sequence",
7.             "~children": [
8.               {
9.                 "#operator": "IndexScan3",
10.                "as": "d",
11.                "covers": [
12.                  "cover ((`d`.`c0`))",
13.                  "cover ((`d`.`c1`))",
14.                  "cover ((`d`.`c2`))",
15.                  "cover ((`d`.`c3`))",
16.                  "cover ((`d`.`c4`))",
```

```
17.                 "cover ((meta(`d`).`id`))",
18.                 "cover (count(cover ((`d`.`c2`))))",
19.                 "cover (countn(cover ((`d`.`c4`))))",
20.                 "cover (sum(cover ((`d`.`c3`))))",
21.                 "cover (sum(cover ((`d`.`c4`))))"
22.               ],
23.               "index": "idx1",
24.               "index_group_aggs": {
25.                 "aggregates": [
26.                   {
27.                     "aggregate": "COUNT",
28.                     "depends": [
29.                       2
30.                     ],
31.                     "expr": "cover ((`d`.`c2`))",
32.                     "id": 6,
33.                     "keypos": 2
34.                   },
35.                   {
36.                     "aggregate": "COUNTN",
37.                     "depends": [
38.                       4
39.                     ],
40.                     "expr": "cover ((`d`.`c4`))",
41.                     "id": 7,
42.                     "keypos": 4
43.                   },
44.                   {
45.                     "aggregate": "SUM",
46.                     "depends": [
47.                       3
48.                     ],
49.                     "expr": "cover ((`d`.`c3`))",
50.                     "id": 8,
51.                     "keypos": 3
52.                   },
53.                   {
54.                     "aggregate": "SUM",
55.                     "depends": [
56.                       4
57.                     ],
58.                     "expr": "cover ((`d`.`c4`))",
59.                     "id": 9,
60.                     "keypos": 4
61.                   }
62.                 ],
63.                 "depends": [
64.                   1,
65.                   2,
```

```
66.                    3,
67.                    4
68.                  ],
69.                  "group": [
70.                    {
71.                      "depends": [
72.                        1
73.                      ],
74.                      "expr": "cover ((`d`.`c1`))",
75.                      "id": 1,
76.                      "keypos": 1
77.                    },
78.                    {
79.                      "depends": [
80.                        2
81.                      ],
82.                      "expr": "cover ((`d`.`c2`))",
83.                      "id": 2,
84.                      "keypos": 2
85.                    }
86.                  ],
87.                  "partial": true
88.                },
89.                "index_id": "d06df7c5d379cd5",
90.                "index_projection": {
91.                  "entry_keys": [
92.                    1,
93.                    2,
94.                    6,
95.                    7,
96.                    8,
97.                    9
98.                  ]
99.                },
100.               "keyspace": "default",
101.               "namespace": "default",
102.               "spans": [
103.                 {
104.                   "exact": true,
105.                   "range": [
106.                     {
107.                       "inclusion": 0,
108.                       "low": "0"
109.                     }
110.                   ]
111.                 }
112.               ],
113.               "using": "gsi"
114.             },
```

```
115.              {
116.                "#operator": "Parallel",
117.                "~child": {
118.                  "#operator": "Sequence",
119.                  "~children": [
120.                    {
121.                      "#operator": "InitialGroup",
122.                      "aggregates": [
123.                        "sum(cover (count(cover ((`d`.`c2`)))))",
124.                        "sum(cover (countn(cover ((`d`.`c4`)))))",
125.                        "sum(cover (sum(cover ((`d`.`c3`)))))",
126.                        "sum(cover (sum(cover ((`d`.`c4`)))))"
127.                      ],
128.                      "group_keys": [
129.                        "cover ((`d`.`c1`))",
130.                        "cover ((`d`.`c2`))"
131.                      ]
132.                    }
133.                  ]
134.                }
135.              },
136.              {
137.                "#operator": "IntermediateGroup",
138.                "aggregates": [
139.                  "sum(cover (count(cover ((`d`.`c2`)))))",
140.                  "sum(cover (countn(cover ((`d`.`c4`)))))",
141.                  "sum(cover (sum(cover ((`d`.`c3`)))))",
142.                  "sum(cover (sum(cover ((`d`.`c4`)))))"
143.                ],
144.                "group_keys": [
145.                  "cover ((`d`.`c1`))",
146.                  "cover ((`d`.`c2`))"
147.                ]
148.              },
149.              {
150.                "#operator": "FinalGroup",
151.                "aggregates": [
152.                  "sum(cover (count(cover ((`d`.`c2`)))))",
153.                  "sum(cover (countn(cover ((`d`.`c4`)))))",
154.                  "sum(cover (sum(cover ((`d`.`c3`)))))",
155.                  "sum(cover (sum(cover ((`d`.`c4`)))))"
156.                ],
157.                "group_keys": [
158.                  "cover ((`d`.`c1`))",
159.                  "cover ((`d`.`c2`))"
160.                ]
161.              },
162.              {
163.                "#operator": "Parallel",
```

```
164.                   "~child": {
165.                     "#operator": "Sequence",
166.                     "~children": [
167.                       {
168.                         "#operator": "InitialProject",
169.                         "result_terms": [
170.                           {
171.                             "as": "c1",
172.                             "expr": "cover ((`d`.`c1`))"
173.                           },
174.                           {
175.                             "as": "c2",
176.                             "expr": "cover ((`d`.`c2`))"
177.                           },
178.                           {
179.                             "as": "sumc3",
180.                             "expr": "sum(cover (sum(cover ((`d`.`c3`)))))"
181.                           },
182.                           {
183.                             "as": "avgc4",
184.                             "expr": "(sum(cover (sum(cover ((`d`.`c4`))))) / sum(cover
(countn(cover ((`d`.`c4`))))))"
185.                           },
186.                           {
187.                             "as": "countc2",
188.                             "expr": "sum(cover (count(cover ((`d`.`c2`)))))"
189.                           }
190.                         ]
191.                       }
192.                     ]
193.                   }
194.                 }
195.               ]
196.             },
197.             {
198.               "#operator": "Order",
199.               "limit": "2",
200.               "offset": "1",
201.               "sort_terms": [
202.                 {
203.                   "expr": "cover ((`d`.`c1`))"
204.                 },
205.                 {
206.                   "expr": "cover ((`d`.`c2`))"
207.                 }
208.               ]
209.             },
210.             {
211.               "#operator": "Offset",
```

```
212.            "expr": "1"
213.          },
214.          {
215.            "#operator": "Limit",
216.            "expr": "2"
217.          },
218.          {
219.            "#operator": "FinalProject"
220.          }
221.        ]
222.      },
223.      "text": "SELECT d.c1 AS c1, d.c2 AS c2, SUM(d.c3) AS sumc3, AVG(d.c4) AS avgc4, COUNT(d.c2) AS countc2 FROM default AS d WHERE d.c0 > 0 GROUP BY d.c1, d.c2 ORDER BY d.c1, d.c2 OFFSET 1  LIMIT 2;"
224.    }
```

- The “index_group_aggs” (lines 24-88) in the IndexScan section (i.e “#operator”: “IndexScan3”) shows query using index grouping and aggregations.
- If query uses index grouping and aggregation the predicates are exactly converted to range scans and passed to index scan as part of spans, so there will not be any Filter operator in the explain.
- As group by keys did NOT match the leading index keys, indexer will produce partial aggregations. This can be seen as “partial”:true inside “index_group_aggs” at line 87. Query service does Group merging (see line 119-161)
- Indexer projects “index_projection” (lines 91-99) containing group keys and aggregates.
- If the Indexer generates partial aggregations query can’t use index order and requires explicit sort, and “offset”, “limit” can’t be pushed to indexer. The plan will have explicit “Order”, “Offset”, and “Limit” operators (line 197 - 217)
- Query contains AVG(x) which has been rewritten as SUM(x)/COUNTN(x). The COUNTN(x) only counts when x is numeric value.
- During Group merge
    - MIN becomes MIN of MIN
    - MAX becomes MAX of MAX
    - SUM becomes SUM of SUM
    - COUNT becomes SUM of COUNT
    - CONTN becomes SUM of COUNTN
    - AVG becomes SUM of SUM divided by SUM of COUNTN

Example 4: Group and Aggregation with array index

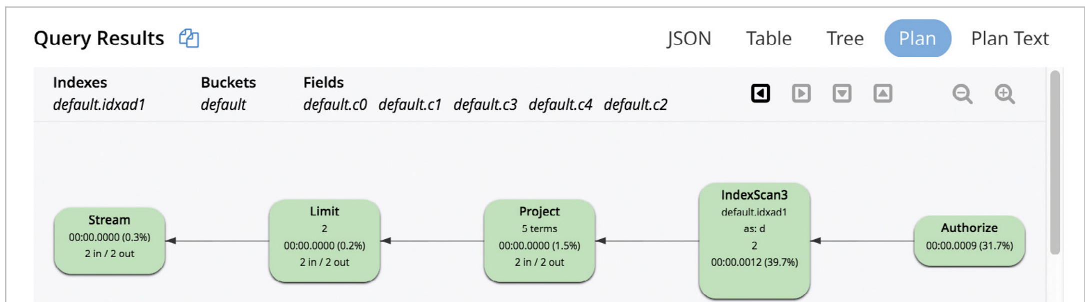
Let consider the following query and index:

```
SELECT d.c0 AS c0, d.c1 AS c1, SUM(d.c3) AS sumc3,
AVG(d.c4) AS avgc4, COUNT(DISTINCT d.c2) AS dcountc2
FROM default AS d
WHERE d.c0 > 0 AND d.c1 >= 10 AND ANY v IN d.a1 SATISFIES v.id = 3 END
GROUP BY d.c0, d.c1
ORDER BY d.c0, d.c1
OFFSET 1
LIMIT 2;
```

Required Index:

```
CREATE INDEX idxad1 ON default(c0,c1, DISTINCT ARRAY v.id FOR v IN a1 END, c2,c3,c4);
```

The query has GROUP BY and multiple aggregates, some of aggregates has DISTINCT modifier. The query predicate has ANY clause and query can be covered by array index index idxad1. The predicate (d.c0 > 0 AND d,c11 >= 10 AND ANY v IN d.a1 SATISFIES v.id = 3 END) can be converted into exact range scans and passed to index scan. For array index Indexer maintain separate element for each array index key, in order to use index group and aggregation the SATISFIES predicate must have a single equality predicate and the array index key must have DISTINCT modifier. Therefore index and query combination is suitable to handle Index grouping and aggregations.



This example is similar to example 1 except it uses an array index. The above graphical execution tree shows index scan (IndexScan3) performing scan, index grouping aggregations, order, offset and limit. The results from the index scan are projected.

Example 5: Group and Aggregation of UNNEST Operation

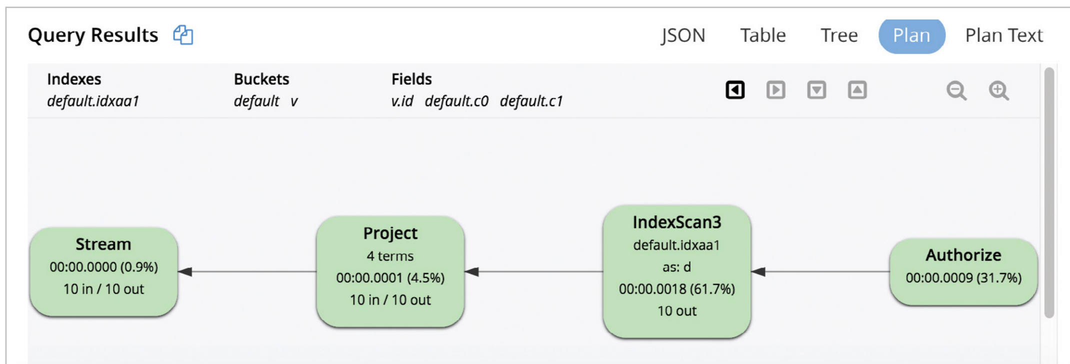
Let consider the following query and index:

```
SELECT v.id AS id, d.c0 AS c0, SUM(v.id) AS sumid,
AVG(d.c1) AS avgc1
FROM default AS d UNNEST d.a1 AS v
WHERE v.id > 0
GROUP BY v.id, d.c0;
```

Required Index:

```
CREATE INDEX idxaa1 ON default(ALL ARRAY v.id FOR v IN a1 END, c0,c1);
```

The query has GROUP BY and multiple aggregates. The query has UNNEST on array d.a1 and have predicate on the array key (v.id > 0). The index idxaa1 qualifies query (For Unnest to use Array index for Index scan the array index must be leading key and array variable in the index definition must match with UNNEST alias). The predicate (v.id > 0) can be converted into exact range scans and passed to index scan. Therefore index and query combination is suitable to handle Index grouping and aggregations.



The above graphical execution tree shows index scan (IndexScan3) performing scan, index grouping aggregations. The results from the index scan are projected. The UNNEST is special type of JOIN between parent and each array element. Therefore, the UNNEST repeats the parent document fields (d.c0, d.c1) and the d.c0, dc.1 reference would have duplicates compared to the original d documents (Need to aware this while using in SUM(), AVG()).

Rules for Index Grouping and Aggregation

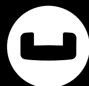
The Index grouping and aggregation are per query block, and decision on whether or not use index grouping/aggregation is made only after index selection process.

- Query block should not contain Joins, NEST, SUBqueries.
- Query block must be covered by singline index.
- Query block should not contain ARRAY_AGG()
- Query block can't be correlated
- All the predicates must be exactly translated into range scans.
- GROUP BY, Aggregate expressions can't reference any subquires, named parameters, positional parameters.
- GROUP BY keys, aggregate expressions can be index keys, document key, expression on index keys, or expression on document key
- Index needs to be able to do grouping and aggregation on all the aggregates in query block otherwise no index aggregation. (i.e. ALL or None)
- Aggregate contain DISTINCT modifier
    - The group keys must exactly match with leading index keys (if the query contains equality predicate on the index key, then it assumes this index key is implicitly included in GROUP keys if not already present).
    - The aggregate expression must be on one of the n+1 leading index keys (n represents number of group keys).
    - In case of partition index the partition keys must exactly match with group keys.

Summary

When you analyze the explain plan, correlate the predicates in the explain to the spans and make sure all the predicate exactly translated to range scans and query is covered. Ensure query using index grouping and aggregations, and if possible query using full aggregations from indexer by adjusting index keys for better performance.

Index Partitioning



INDEX PARTITIONING

Authors: John Liang, Couchbase R&D
Keshav Murthy, Couchbase R&D

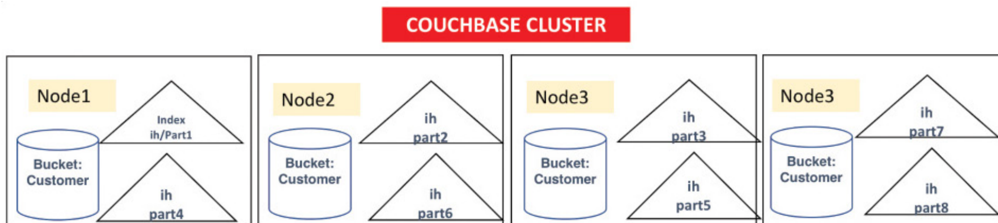
In Couchbase, data is always partitioned using the consistent hash value of the document key into vbukets which are stored on the data nodes. Couchbase Global Secondary Index (GSI) abstracts the indexing operations and runs as a distinct service within the Couchbase data platform. When a single index can cover a whole type of documents, everything is good. But, there are cases where you're want to partition an index.

1. Capacity: You want increased capacity because a single node is unable to hold a big index
2. Queriability: You want to avoid rewriting the query to work with manual partitioning of the index using a partial index.
3. Performance: Single index is unable to meet the SLA

To address this, Couchbase 5.5 introduces automatic hash partitioning of the index. You're used to having bucket data hashed into multiple nodes. Index partitioning enables you to hash the index into multiple nodes as well. There is good symmetry.

Creating the index is easy. Simply add a PARTITION BY clause to the CREATE index definition.

```
CREATE INDEX ih ON customer(state, name, zip, status)
     PARTITION BY HASH(state)
     WHERE type = "cx" WITH {"num_partition":8}
```



This as the following meta data in the system:indexes. Note the new field partition with the hash expression. The HASH(state) is the basis on which the index logically named **`customer`.`ih`** is divided into a number of physical index partitions. By default, the number of index partitions are 16 and it can be changed by specifying num_partition parameter. In the example above, we create 8 partitions for the index **`customer`.`ih`**.

```
select *
from system:indexes
where keyspace_id = "customer" and name = "ih" ;
  {
    "indexes": {
      "condition": "(`type` = \"cx\")",
      "datastore_id": "http://127.0.0.1:8091",
      "id": "b3ce745f84256319",
      "index_key": [
        "`state`",
        "`name`",
        "`zip`",
        "`status`"
      ],
      "keyspace_id": "customer",
      "name": "ih",
      "namespace_id": "default",
      "partition": "HASH(`state`)",
      "state": "online",
      "using": "gsi"
    }
  }
```

Now, issue the following query. You don't need additional predicate on the hash key for the query to use the index. The index scan simply scans all of the index partitions as part of the index scan.

```
SELECT *
FROM customer
WHERE type = "cx"
      and name = "acme"
      and zip = "94051";
```

However, if you do have an equality predicate on the hash key, index scan detects the right index partition having the right range of data and prunes rest of the index nodes from the index scan. This makes the index scan very efficient.

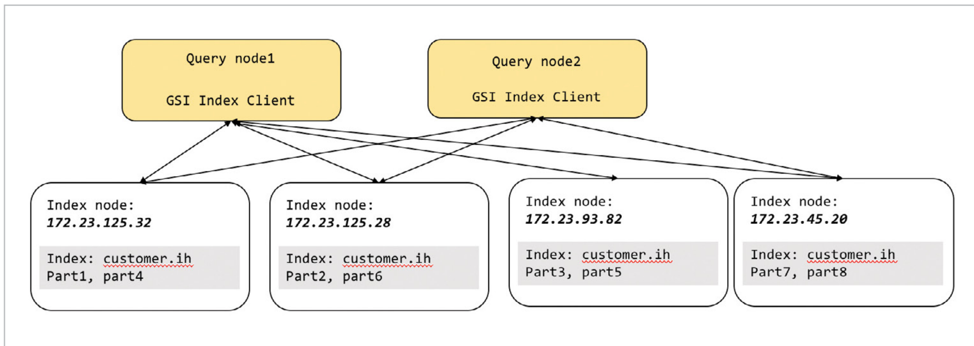
```
SELECT *
FROM customer
WHERE type = "cx"
      and name = "acme"
      and zip = "94051"
      and state = "CA";
```

Now, let's look at how this index helps you with three things we mentioned before: Capacity, Queriability and Performance.

Capacity

The query **`customer`.`ih`** will be partitioned to a specified number of partitioned with each partition stored on one of the index nodes on the cluster. The indexer uses a stochastic optimization algorithm to determine how to distribute the partitions onto the set of indexer nodes, based on the free resource available on each node. Alternatively, to restrict the index to a specific set of nodes, use the nodes parameter. This index will create eight index partitions and store four each on the four index nodes specified.

```
CREATE INDEX ih ON customer(state, name, zip, status)
     PARTITION BY HASH(state)
     WHERE type = "cx" WITH {"num_partition":8,
     "nodes":["172.23.125.32:9001", "172.23.125.28:9001"],
"172.23.93.82:9001","172.23.45.20:9001" ]}
```



So, with this hash partitioned index, one logical index (**`customer`.`ih`**) will be partitioned into a number of physical index partitions (in this case, 8 partitions) and give the query an illusion of a single index.

Because this index uses the multiple physical nodes, the index will have more disk, memory and CPU resources available. Increased storage in these nodes makes it possible to create larger indexes.

You write your queries, as usual, requiring predicates only the WHERE clause (type = “cx”) an at least on one of the leading index keys (e.g. name).

Queriability

Limitations in the Couchbase 5.0 indexing:

Until Couchbase 5.0, you could manually partition the index like below. You had to partition them manually using the WHERE clause on the CREATE INDEX. Consider the following indexes, one per state. By using the node parameter, you could place them in specific index nodes or the index will try to automatically spread out within the index nodes.

```
CREATE INDEX i1 ON customer(name, zip, status) WHERE state = "CA";
CREATE INDEX i2 ON customer(name, zip, status) WHERE state = "NV";
CREATE INDEX i3 ON customer(name, zip, status) WHERE state = "OR";
CREATE INDEX i4 ON customer(name, zip, status) WHERE state = "WA";
```

For a simple query with equalify predicate on state, it all works well.

```
SELECT *
FROM customer
WHERE state = "CA" and name = "acme" and zip = "94051";
```

There are two issues with this manual partitioning.

1. Consider the following with slightly complex predicate on the state. Because the predicate (state IN [“CA”, “OR”]) is not a subset of any of the WHERE clauses of the index, none of the indexes can be used for the query below.

   ```
   SELECT * FROM customer
        WHERE state IN ["CA", "OR"] and name = ACME;
   SELECT * FROM customer
        WHERE state > "CA" and name = ACME;
   ```

2. If you get data to a new state, you’re to be aware of it and create the index in advance.

   ```
   SELECT * FROM customer WHERE state = "CO" and name = ACME
   ```

   If the field numerical field, you can use the MOD() function.

   ```
   CREATE INDEX ix1 ON customer(name, zip, status)
             WHERE (MOD(cxid) % 4 = 0);
   CREATE INDEX ix2 ON customer(name, zip, status)
             WHERE (MOD(cxid) % 4 = 1);
   CREATE INDEX ix3 ON customer(name, zip, status)
             WHERE (MOD(cxid) % 4 = 2);
   CREATE INDEX ix4 ON customer(name, zip, status)
   WHERE (MOD(cxid) % 4 = 3);
   ```

Even this work around each query block can only use one index and requires queries to be written carefully to match one of the predicates in the WHERE clause.

Solution:

As you see from the figure above, the interaction between the query and index goes through the GSI client sitting inside each query node. Each GSI client gives the illusion of a single logical index (**`customer`.`ih`**) on top of eight physical index partitions.

The GSI client takes all of the index scan request and then using the predicate, tries to see if it can identify which of index partitions has the data needed for the query. This is the process of partition pruning (aka partition elimination). For the hash based partitioning scheme, equality and IN clause predicates get the benefit of partition pruning. All other expressions use the scatter-gather method. After the logical elimination, GSI client sends the request to the remaining nodes, gets the result, merges the result and sends the result back to query. The big benefit of this is that queries can be written without worrying about the manual partitioning expression.

Example query below does not even have a predicate on the hash key, state. The below query does not get the benefit of partition elimination. Therefore, the GSI client issues scan to every index partition in parallel and then merges the result from each of the index scan. The big benefit of this is that queries can be written without worrying about the manual partitioning expression to match the partial index expression and still use the full capacity of the cluster resources.

```
CREATE INDEX ih1 ON customer(name, zip, status)
PARTITION BY HASH(state)
WHERE type = "cx" WITH {"num_partition":8}

SELECT *
FROM customer
WHERE type = "cx"
      and name = "acme"
      and zip = "94051";
```

Additional predicate on the hash key (state = “CA”) in the query below will benefit from partition pruning. For query processing, for simple queries with equality predicates on hash key, you get uniform distribution of the workload on these multiple partitions of the index. For complex queries including the grouping & aggregation we discussed above, the scans, partial aggregations are done in parallel, improving the query latency.

```
SELECT *
FROM customer
WHERE type = "cx"
     and name = "acme"
     and zip = "94051"
     and state = "CA";
```

You can create indexes by hashing on one or more keys, each of which could be an expression. Here are some examples.

```
CREATE INDEX idx1 ON customer(name) PARTITION BY HASH(META().id);
CREATE INDEX idx2 ON customer(name) PARTITION BY HASH(name, zip);
CREATE INDEX idx3 ON customer(name)
                      PARTITION BY HASH(SUBSTR(name, 5, 10));
CREATE INDEX idx3 ON customer(name)
     PARTITION BY
     HASH(SUBSTR(META().id, POSITION(META().id, "::")+2), zip)
```

Performance

For majority of database features, performance is everything. Without a great performance, proven by good benchmarks, the features are simply pretty syntax diagrams!

Index partitioning gives you improved performance in two ways.

1. Scale out. The partitions are distributed into multiple nodes, increasing the CPU and memory availability of for the index scan.

2. Parallel scan. Right predicate giving queries the benefit of partition pruning. Even after the pruning process, scans of all the indexes are done in parallel.

3. Parallel grouping and aggregation. The DZone article Understanding Index Grouping and Aggregation in Couchbase N1QL Queries explains the core performance improvement of grouping and aggregation using indexes.

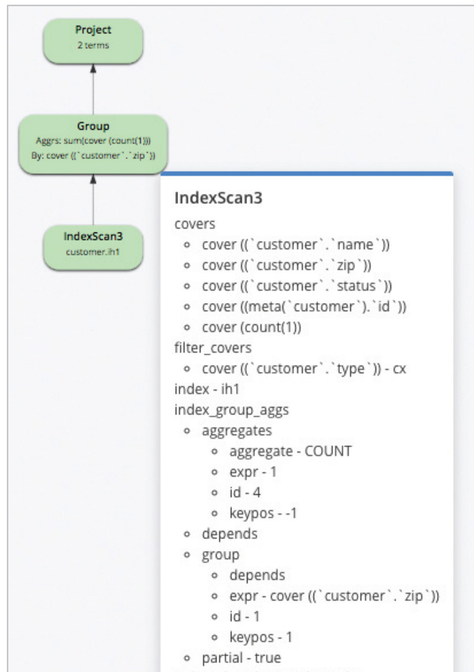
4. The parallelism of the index parallel scan (and grouping, aggregation) is determined by the max_parallelism parameter. This parameter can be set per query node and/or per query request.

Consider the following index and query:

```
CREATE INDEX ih1 ON customer(name, zip, status)
PARTITION BY HASH(state)
WHERE type = "cx" WITH {"num_partition":8}

select zip, count(1) zipcount
from customer
where type = "cx" and name is not missing
group by zip;
```

The index is partitioned by HASH(state), but state predicate is missing from the query. For this query, we cannot do partition pruning or create groups within individual scans of the index partitions. Therefore, it will need a merge phase after the partial aggregation with the query (not shown in the explain). Remember, these partial aggregations happen in parallel and therefore reduces the latency of the query.



Consider the following index and query:

```
CREATE INDEX ih2 ON customer(state, city, zip, status)
PARTITION BY HASH(zip)
WHERE type = "cx" WITH {"num_partition":8}
```

Example a:

```
select state, count(1) zipcount
from customer
where state is not missing
group by state, city, zip;
```

In the above example, the group by is on the leading keys (state, city, zip) of the index and hash key (zip) is part of the group by clause. This will help the query to scan the index and simply created the required groups

Example b:

```
select zip, count(1) zipcount
from customer
where type = "cx"
and city = "San Francisco"
and state = "CA"
group by zip;
```

In the above example, the group by is on the third key (zip) of the index and hash key (zip) is part of the group by clause. In the predicate clause (WHERE clause), there is single equality predicate on the leading index keys before the key zip (state and city). Therefore, we implicitly include the keys (state, city) in the group by without affecting the query result. This will help the query to scan the index and simply created the required groups.

Example c:

```
select zip, count(1) zipcount
from customer
where type = "cx"
and city like "San%"
and state = "CA"
group by zip;
```

In the above example, the group by is on the third key (zip) of the index and hash key (zip) is part of the group by clause. In the predicate clause (WHERE clause), there is range predicate on city. The index key (city) is before the hash key(zip). So, we create partial aggregates as part of the index scan and then then query will merge these partial aggregates to create the final resultset.

Summary

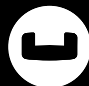
Index partition gives you increased capacity for your index, better queriability and higher performance for your queries. By exploiting the Couchbase scale-out architecture, indexes improve your capacity, queriability, performance and TCO.

References

1. Couchbase documentation:
   https://developer.couchbase.com/documentation/server/5.5/indexes/gsi-for-n1ql.html

2. Couchbase N1QL documentation:
   https://developer.couchbase.com/documentation/server/5.5/indexes/gsi-for-n1ql.html#

N1QL Auditing



AUDITING COUCHBASE N1QL STATEMENTS

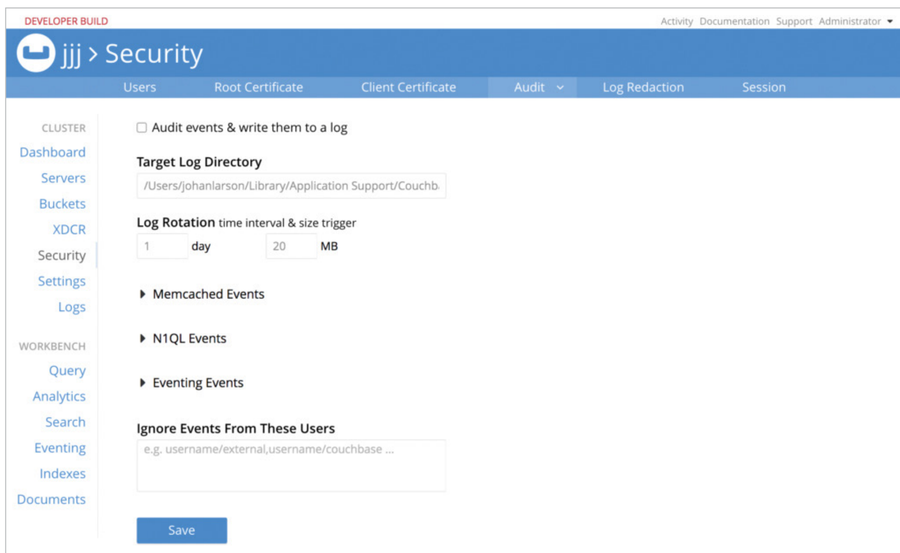
Author: Johan Larson, Senior Software Engineer, Couchbase R&D

Couchbase Server 5.5 includes the ability to keep a record of all N1QL actions taken by users. This is part of Couchbase's more general audit functionality, introduced in 5.0. Auditing is only available in Enterprise edition.

Auditing lets the administrators of the system track who is accessing what data in the system. This is important when the data being stored is sensitive in some way, such as information about users. Couchbase Server 5.5 supports auditing of N1QL statements, and lets the administrator specify what types of statements (SELECTs? INSERTs?) should actually be audited.

It is important to understand what Couchbase Server 5.5 does not do. In particular, it does not allow record-level auditing. If an UPDATE statement is run and modifies five records, the audit record will include the whole statement that ran, including any parameters passed in, and it will say that five records were updated. It will not say what specific records were updated, or what their values were before or after the operation. Fundamentally, N1QL auditing audits statements, not records.

To configure audit, log in to the Couchbase Admin console. Navigate to the Security tab (on the side) and to the Audit tab (at the top of the screen). You should now see a screen like this:

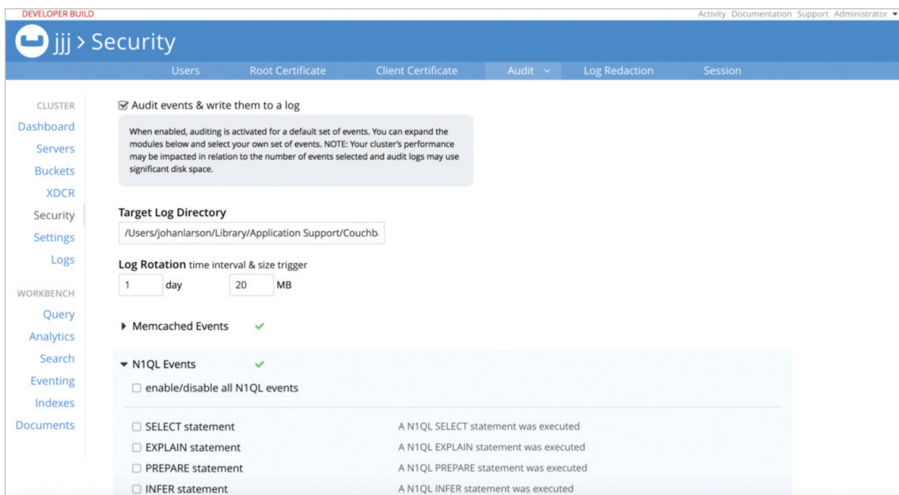


This tab lets you configure auditing in general. The checkbox at the top indicates whether auditing should be done at all. “Target Log Directory” shows where to put the audit log records. The records appear in a file named “audit.log” in the target log directory. The next set of text boxes control log rotation by size and time interval.

Next are three dropdowns for various types of events, giving you fine control over what sorts of activities should be logged. Generally speaking, audit only what you must. The actual throughput cost of auditing depends on how much is audited, and the type of statements being audited. Ten percent throughput loss due to auditing is a reasonable off-the-cuff estimate, but you should definitely test the actual effect before rolling out a new system.

Finally, you can whitelist users in the “Ignore Events From These Users” box. These are users who are trusted so completely their actions do not need to be logged. For example, you may have an automated script that inserts new data. You trust this script completely. Creating a whitelisted user and having the script use that user’s credentials may be useful to avoid generating too many audit records.

Toggle the “N1QL Events” dropdown, to see the types of events available for N1QL.



There are two general types. First are events corresponding to N1QL statement types. For example, you can choose to audit all INSERT events, or all DELETE events. It might for example be reasonable to audit all events that modify data (INSERT/DELETE/UPDATE/UPSERT), but ignore statements that only retrieve data (SELECT).

Second are events corresponding to APIs exposed by the query engine. The N1QL query engine makes a number of APIs available, typically for monitoring the system. Each of these API endpoints is a separate event type. For example, there is one for the /admin/stats endpoint, and another for the /admin/ping endpoint. You have separate control over whether to audit accesses to these APIs.

Plain Query

We'll start by auditing a simple SELECT statement.

Go to the "Buckets" page of the admin console, and create a bucket named "test" (no quotes). Memory quota 100 MB is fine for our purposes. Then go to the Query and create a primary index on the new bucket, to allow us to run N1QL queries on it.

```
create primary index on test
```

The go back to the audit configuration screen and select "Audit events & write them to a log" at the top, and the "SELECT statement" option under "N1QL Events". Then press "Save" at the bottom of the screen.

Then run a query like this.

```
curl http://localhost:8093/query/service -d "statement=select * from test" -u Adminis-
trator:password
```

And let's have a look at the audit log. The "Target Log Directory" field of the audit configuration screen has the directory where the audit log is stored. We'll use the "tail" command to show the last few records of the audit log in this directory. On Mac systems, this command works:

```
tail ~/Library/Application\ Support/Couchbase/var/lib/couchbase/logs/audit.log
```

You should see several long lines of JSON text. Each line is one audit record. The last one is the record for the statement we sent. Reformatted, it looks like this:

```
1.  {
2.    "timestamp": "2018-03-14T05:53:34.976-07:00",
3.    "real_userid": {
4.      "source": "local",
5.      "user": "Administrator"
6.    },
7.    "requestId": "d0554df3-fd99-40f5-b911-b3e4f0faf050",
8.    "statement": "select * from test",
9.    "isAdHoc": true,
```

```
10.    "userAgent": "curl\/7.43.0",
11.    "node": "127.0.0.1:8091",
12.    "status": "success",
13.    "metrics": {
14.      "elapsedTime": "822.147\u00b5s",
15.      "executionTime": "785.755\u00b5s",
16.      "resultCount": 0,
17.      "resultSize": 0
18.    },
19.    "id": 28672,
20.    "name": "SELECT statement",
21.    "description": "A N1QL SELECT statement was executed"
22.  }
```

Let's go through these field by field:

- "timestamp" shows the time from the query node.
- "real_userid" shows what user credential was supplied with the request. In this case it is the build-in user, "Administrator".
- "requestId" is the UUID the query engine generates for every request. These IDs are unique with very high probability.
- "statement" is the actual statement we executed.
- "isAdHoc" is true in this case, showing that we sent an actual statement for execution, rather than running a prepared statement.
- "userAgent" is the User-Agent string from the original request. This is useful for distinguishing whether the request came from an SDK, or the CBQ shell, or the Query WorkBench.
- "node" is the IP address from which the request was received.
- "status" shows what happened to the request. In this case, it succeeded.
- "metrics" is a set of statistics about the result. This matches the metrics that were sent with the result of the original request.
- "id" is the event type ID. The audit records for all SELECT queries have the same id, 28672.
- "name" is the short name of the event type. This will be the same for all SELECT queries.
- "description" is the long name of the event type. This is also the same for all SELECT queries.

Note that the audit record provides for only one user, although the query engine allows for multiple credentials per request. This is by design. N1QL allowed multiple credentials for queries back when our credentials were per-bucket, and multiple credentials were therefore necessary

for multi-bucket joins. But as of 5.0, with RBAC, multiple credentials are no longer necessary. We support them for backward compatibility, but the right way to handle such cases is to create users with credentials for multiple buckets, and use one such user for each query. If you insist on using multiple credentials for an audited query, the query will get audited, but there will be a separate audit record for every credential supplied. That's a bit awkward, so we strongly suggest updating the permissions model to use RBAC permissions in such cases.

Prepare Statement

Now let's consider a more sophisticated case, with a prepared statement. First, go back to the audit configuration screen, and turn on auditing of SELECT and PREPARE statements. Remember to hit "Save" at the bottom of the screen.

Now, we'll first prepare a statement. Here we are preparing a SELECT statement, with name "example". Note that the statement has an unnamed parameter.

```
curl http://localhost:8093/query/service -d "statement=prepare example as select * from
test where one=?" -u Administrator:password
```

Then, we'll execute the statement, supplying an argument for the statement. In this case, the statement will run, but return no results.

```
curl http://localhost:8093/query/service -d ‘prepared="example"&args=["bar"]’
```

Now let's have a look at the audit log again.

```
tail ~/Library/Application\ Support/Couchbase/var/lib/couchbase/logs/audit.log
```

The log will show two events, one for the PREPARE, and one for the SELECT executed from the prepared statement:

```
1.   {
2.     "timestamp": "2018-03-14T06:27:39.884-07:00",
3.     "real_userid": {
4.       "source": "local",
5.       "user": "Administrator"
6.     },
7.     "requestId": "9f76b8c2-ed9f-42f8-bc5c-31fb3326a661",
8.     "statement": "prepare example as select * from test where one=?",
9.     "isAdHoc": true,
10.    "userAgent": "curl\/7.43.0",
11.    "node": "127.0.0.1:8091",
12.    "status": "success",
13.    "metrics": {
```

```
14.       "elapsedTime": "6.591126ms",
15.       "executionTime": "6.515079ms",
16.       "resultCount": 1,
17.       "resultSize": 1279
18.     },
19.     "id": 28674,
20.     "name": "PREPARE statement",
21.     "description": "A N1QL PREPARE statement was executed"
22.   }
23.   {
24.     "timestamp": "2018-03-14T06:27:52.992-07:00",
25.     "real_userid": {
26.       "source": "internal",
27.       "user": "unknown"
28.     },
29.     "requestId": "56c5278b-5842-45a9-8549-5c7f52f109a7",
30.     "statement": "",
31.     "positionalArgs": [
32.       "\"bar\""
33.     ],
34.     "isAdHoc": false,
35.     "userAgent": "curl\/7.43.0",
36.     "node": "127.0.0.1:8091",
37.     "status": "success",
38.     "metrics": {
39.       "elapsedTime": "1.363373ms",
40.       "executionTime": "1.334763ms",
41.       "resultCount": 0,
42.       "resultSize": 0
43.     },
44.     "id": 28672,
45.     "name": "SELECT statement",
46.     "description": "A N1QL SELECT statement was executed"
47.   }
```

The fields of the audit records are similar to the earlier execution of a SELECT statements, but two fields bear notice:

- “positionalArgs” contains the argument supplied with the query.
- “isAdHoc” is in this case false, because the SELECT was executed from a prepared statement that was sent earlier.

API Request

Next, let's try auditing one of the query engine APIs. Go to the audit configuration page, and turn on the "/admin/ping API request" event type. Don't forget to save the configuration at the bottom of the page.

Now send a ping:

```
curl -v http://localhost:8093/admin/ping
```

Don't expect much, the "{}" at the bottom is the entire result:

```
1.   *   Trying ::1...
2.   * Connected to localhost (::1) port 8093 (#0)
3.   &gt; GET /admin/ping HTTP/1.1
4.   &gt; Host: localhost:8093
5.   &gt; User-Agent: curl/7.43.0
6.   &gt; Accept: */*
7.   &gt;
8.   &lt; HTTP/1.1 200 OK
9.   &lt; Date: Wed, 14 Mar 2018 13:54:24 GMT
10.  &lt; Content-Length: 2
11.  &lt; Content-Type: text/plain; charset=utf-8
12.  &lt;
13.  * Connection #0 to host localhost left intact
14.  {}
```

Then let's have a look at the audit log (again, using the location on Macs):

```
tail ~/Library/Application\ Support/Couchbase/var/lib/couchbase/logs/audit.log
```

The resulting audit log message, formatted, looks like this:

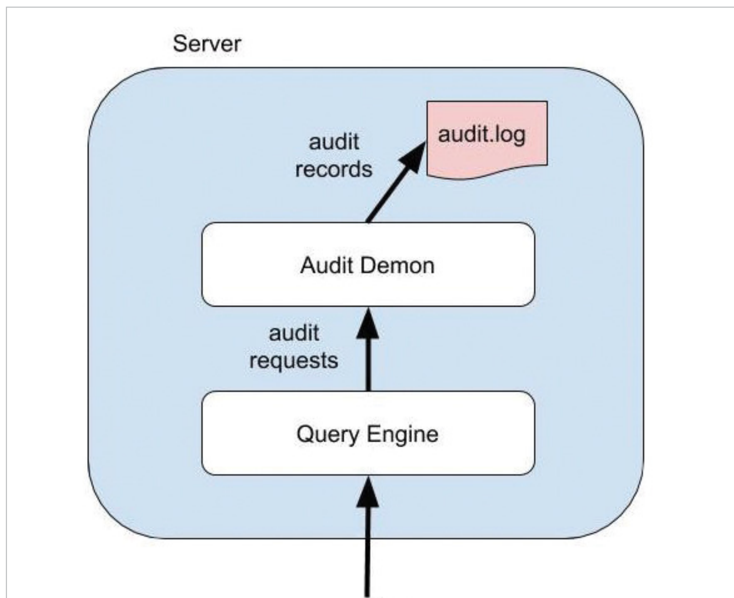
```
1.   {
2.     "timestamp": "2018-03-14T06:54:24.887-07:00",
3.     "real_userid": {
4.       "source": "internal",
5.       "user": "unknown"
6.     },
7.     "httpMethod": "GET",
8.     "httpResultCode": 200,
9.     "errorMessage": "",
10.    "id": 28697,
11.    "name": "/admin/ping API request",
12.    "description": "An HTTP request was made to the API at /admin/ping."
13.  }
```

Here “timestamp” and “real_userid” fields work as before, in the SELECT example. “httpMethod” is the type of HTTP request. “httpResultCode” and “errorMessage” indicate what happened with the request. “Id”, “name” and “description” are specific to the audit event; these fields will be identical for all audit records created for /admin/ping events.

Forward Filtering

(This is an advanced topic. You don’t need to know the material in this section to use N1QL auditing effectively. But a look under the hood may be of interest to advanced users.)

Auditing is controlled in each server by an executable called the audit demon. The audit demon creates all records in the audit log. In 5.0, the audit demon was responsible for all filtering of events; clients sent records for all auditable events, and the audit demon would create audit records in the log, or not, depending on the filtering configuration. Unfortunately, this would be very inefficient when auditing is highly filtered an clients are doing a lot of potentially auditable work. A client such as the query engine might generate millions of records only to have them thrown away by the audit demon when they arrived.



To alleviate this problem, in 5.5 Couchbase supports forward filtering. The query engine is aware of the current audit configuration, and sends only the currently audited records to the audit demon. It also sends a special audit record to indicate that it has received the new configuration and is aware of it.

This dual filtering is why you may see two types of configuration records in the audit log. A record like this indicates the audit demon has received a new configuration:

```
1.  {"rotate_size":20971520,"log_path":
    "/Users/johanlarson/Library/Application Support/Couchbase/var/lib/couchbase/logs
    ","rotate_interval":86400,
2.  "disabled_userids":[],"auditd_enabled":true,
3.  "disabled":[20485,20488,20489,20490,20491,28673,28675,28676,28677,28678,
    28679,28680,28681,28682,
4.  28683,28684,28685,28686,28687,28688,28689,28690,28691,28692,28693,28694,
    28695,28697,28698,28699,
5.  28700,28701,28702,32770,32771,32772,32780],
6.  "enabled:[20480,20482,20483,28672,28674,32768,32769,32773,32774,32775,32776,
    32777,32778,32779,32781,32782],
7.  "real_userid":{"source":"ns_server","user":"Administrator"},"sessionid":
    "8b3d16bffa8444ce596b64a78c0185f7",
8.  "remote":{"ip":"127.0.0.1","port":52153},
9.  "timestamp":"2018-03-14T06:25:30.370-07:00","id":8240,"name":"configured audit
    daemon",
10. "description":"loaded configuration file for audit daemon"}
```

And a record like this indicates that the query engine has received a new configuration:

```
1.  {"timestamp":"2018-03-14T06:25:30.427-07:00",
2.  "real_userid":{"source":"","user":""},"uuid":"26571424","id":28703,
3.  "name":"N1QL configuration","description":"States that N1QL is using audit
    configuration with specified uuid"}
```

Note the UUID that identifies the configuration. You can get this UUID from the configuration, like this:

```
curl http://localhost:8091/pools/default -u Administrator:password
```

Look for the “auditUid” field.

You can get the complete audit configuration like this:

```
curl http://localhost:8091/settings/audit -u Administrator:password
```

```
 1.  {"disabled":[20485,20488,20489,20490,20491,28673,28675,28676,28677,28678,
 2.  28679,28680,28681,28682,28683,28684,28685,28686,28687,28688,28689,
 3.  28690,28691,28692,28693,28694,28695,28698,28699,28700,28701,28702,
 4.  32770,32771,32772,32780],
 5.  "uid":"18635804","auditdEnabled":true,"disabledUsers":[],
 6.  "logPath":"/Users/johanlarson/Library/Application Support/Couchbase/var/lib/
     couchbase/logs",
 7.  "rotateInterval":86400,"rotateSize":20971520}
```

Loading the Audit Log

Couchbase Server currently only supports one destination for audit records: a file on the server. But sometimes it would be useful to get the audit records into the database itself. This is not difficult, since the audit records are JSON. But loading the log does require use of a utility, cbimport.

Assuming you have the audit log created in the standard location on a Mac, and you have created the "test" bucket, this incantation loads the audit.log file into the "test" bucket:

```
/Applications/Couchbase\ Server.app/Contents/Resources/couchbase-core/bin/cbimport json
-c http://localhost:8091 -u Administrator -p password -b test -g "#UUID#" -d file:///
Users/johanlarson/Library/Application\ Support/Couchbase/var/lib/couchbase/logs/audit.
log -f lines
```

That's rather a lot to take it, and you would need slightly different variations on other systems, so let's go through this step by step.

- **/Applications/Couchbase\ Server.app/Contents/Resources/couchbase-core/bin/cbimport** is the full path to the cbimport command on a Mac. For other systems, the utilities are located elsewhere. See this document.
- **-c http://localhost:809**1 is the URL of the server where Couchbase is running
- **-u Administrator -p password** is the username and password of the user we are uploading the data as (in this case the default administrator.)
- **-b test** is the name of the bucket we are uploading the data into.
- **-g "#UUID#"** is the type of key to generate for each document entered into the bucket. In this case, we are using a UUID, but there are many other options. Check the cbimport documentation for more information.
- **-d file:///Users/johanlarson/Library/Application\ Support/Couchbase/var/lib/couchbase/logs/audit.log is a file URL pointing to the location of the audit log.** Note the three forward slashes and the backslash to allow a space in the URL path. The logs, including the audit log, are placed in standard directories that vary from system to system. See this document for more information.

Once the audit records are in the system, you can query them just like any other data. Go to the Query WorkBench to try it out.

This query shows how many audit records you have:

```
select count(*) as num from test
```

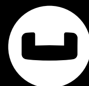
And this query breaks down the count by audit record type:

```
select name, count(*) as num from test group by name
```

Summary

- Requests to query engine are auditable as of 5.5 EE.
- Auditing in general supports filtering by event type and user whitelisting.
- Requests are marked as events by query type and API endpoint.
- Additional documentation about auditing of N1QL statements is available here.

X.509 for Query

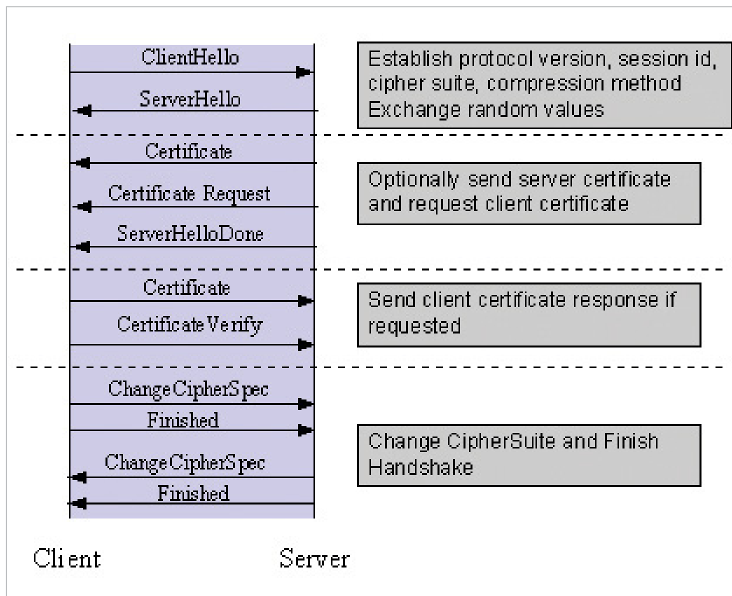


N1QL SUPPORT FOR X.509

Authors: Isha Kandaswamy, Senior Software Engineer, Couchbase R&D
Ajit Yagaty, Senior Software Engineer, Couchbase R&D

Couchbase Server 5.5 supports X.509 certificates to encrypt client server communications within a cluster. This new authentication method adds to the list of authentication methods Couchbase already supports today (SSL/TLS, password based built-in authentication for data buckets, and LDAP authentication for administrators)

With SSL authentication, a server certificate is exchanged between the server and the client which the client uses to verify the identity of the server. This starts with a client and server hello where the protocol version, session ID, SSL version number, Cipher settings and other session specific data is exchanged with the server. Once these values have been established, then the authentication step happens. Here the Client authenticates the server using the certificate sent by the server. The server in turn requests the client certificate which is used to identify the client. The server then uses its private key to decrypt the certificate. They agree to use specific ciphers. With this all future messages between client and server will be encrypted. Certificates are configured in .pem format.



X.509 certificate is the official standard for public key certificates and on which SSL/TLS relies. To enable SSL, a X.509 certificate needs to be installed on Couchbase Server.

The certificate authority (CA) issues digital certificates which certifies the ownership of a public key by the named subject of the certificate. There are 2 types of CA's, root CA and intermediate CA.

The root CA is the topmost CA. All certificates immediately below the root certificate inherit its trustworthiness, and can be used to secure systems. The root CA issues certificates to the intermediate CA. The intermediate CA generates *intermediate certificates* used to sign client certificates or the cluster certificate. This represents an n tier hierarchy (depending on how many intermediates there are).

Root CA signs-> Intermediate CA signs -> Cluster CA

In order for a certificate to be trusted, and often for a secure connection to be established at all, that certificate must have been issued by a CA that is included in the trusted store of the device that is connecting. Also the clients need to have a valid certificate signed by the same root CA.

For more detailed information on configuring X.509 please refer to the documentation - https://developer.couchbase.com/documentation/server/current/security/security-x509certsintro.html.

Certificate needs to be manually generated in a .pem format, signed by an intermediate certificate (ie CN=<host-name>), and then signed with root certificate and finally loaded into Couchbase. Service certificates (client-certs) are signed by the same chain of trust that terminate at root CA authority as Server certificates.

Once certificates are refreshed and the SSL config has been updated (post enabling certificate auth), any new connections coming into the server will make use of the new config. However existing connections will not be affected.

Deeper dive into supporting Client Certificate Authorization

There are three states in which the authorization operation can be performed - disable, enable and mandatory. When disabled, no client authorization is performed. This is the default mode. If the client presents a certificate and client authorization has been enabled, then it will be used and if the certificate cannot be authenticated then access will be denied. However, if the client does not have to present a certificate and if none is presented then the certificate based authentication will be bypassed. If the system is mandated to have a certificate (status setting as mandatory) then the client needs to present a valid certificate in order to gain access to Couchbase buckets and Admin UI.

Another setting that the Couchbase server uses is prefixes. Prefixes are used to extract the user name from the client certificate. A prefix entry is an array of {path,prefix,delimiter} triples. The user can specify up to 10 such triplets.

The following table explains the intent of each field in a triple:

Field	Description
path	This denotes the field in the X.509 certificate that will be used to extract the username from. If the *State* setting is either *enable* or *mandatory* then this field must be set by the user. It can take the following values: 1. **subject.cn:** Refers to the *commonName* field in subject section of the certificate. 2. **san.uri:** Refers to the *URI* field in *Subject Alternate Names* section of the certificate. 3. **san.email:** Refers to the *email* field in *Subject Alternate Names* section of the certificate. 4. **san.dnsname:** Refers to the *dns* field in *Subject Alternate Names* section of the certificate.
prefix	This is an optional field. This denotes the prefix to be ignored from the username read from the *Path*. If this field is specified and the username doesn't have the prefix then we move on to the next triple. If it's empty then we use the entire value found in *Path*.
delimiter	This is an optional field. Multiple characters can be configured as the delimiters. After the prefix removal, we pick the characters from the value until a delimiter is encountered and that would be returned as the username. If there is no match against any of delimiters then we return the entire string as the username.

Each triple is processed until a match is found. A "match" is when we find a name in the client certificate that satisfies path and prefix. Delimiter does not count for matches. If the delimiter is not found we take up to the end of the string.

Sample example of the client cert authorization settings -

```
{
  "state" : "enable",
  "prefixes" : [
    { "path" : "subject.cn", "prefix" : "www.cb-", "delimiter" : "." },
    { "path" : "san.dnsname", "prefix" : "us.", "delimiter" : "." },
    { "path" : "san.uri", "prefix" : "www.", "delimiter" : "." }
  ]
}
```

X.509 Client certificate authorization behavior

Support for authorization based on client certificates in ns server has also been added into 5.5. The client certificate will be generated with the username encoded into one of the fields in the certificate. The server will receive an HTTP request, and check for the appropriate request header to assign basic token authentication. If the client certificate authorization status is set to enable or mandatory, the server tries to get the certificate from the request object. If client certificate is missing and if authorization state is *enable* then server falls back to regular basic-auth. But if the client cert authorization state is *mandatory* then server fail the request.

Once the server gets the client certificate from the request, then the prefix from the config is used to extract the username. The server reads the value of the field from the certificate identified by *path* and then uses the *prefix* and *delimiter* settings to extract the username from the value read. Once the username extraction is successful then that is used as a local identity and the server checks to see if the identity has the required permissions to process request. If not, then the request fails.

If the client certificate presented is incorrect then the request will be rejected at the TLS level. Typically the error string returned will be “bad certificate” and TLS layer provides an alert number associated with it. If the client certificate is correct (TLS authentication works) but the user identity encoded within the certificate doesn’t match any of the configured prefixes or if the username extracted doesn’t match any of the configured users then an HTTP Error code 401 will be returned.

This is why the prefixes need to be set correctly. When the prefixes are checked by the server, the prefix portion of the value is discarded and the substring found until the delimiter is found in the value is returned as the username. If both are undefined then we return the value as the username. If prefix is defined but it’s not found in the value then we move on to the next triple. Some of the fields (specifically pertaining to Subject Alternate Names) may have multiple values defined. In such a case, we pick the first value that matches the prefix and delimiter.

If none of the triples match then the request will be failed.

Running a query with Client Certificate Authorization

To setup client cert authorization for Couchbase server, we first need to enable TLS on the server.

Then we need to generate client certificates. (Please refer to Couchbase Server 5.1 documentation on X.509 - https://developer.couchbase.com/documentation/server/current/security/security-x509certsintro.html) So create the client, client/int, client/client directories under the SSLCA folder. Once done, generate an intermediate certificate. This intermediate certificate will be used to sign the client certificate. This intermediate certificate will in turn be signed by the root certificate (root/ca.pem). Then the client certificate is generated using an openSSL configuration file. (See examples here- https://www.openssl.org/docs/man1.1.1/man1/req.html)

Once this step is completed, we need to add a user in Couchbase Server (whose username has been encoded in the client certificate), assign required permissions to this user and set the client cert authorization settings. This would be used to extract the username from the client certificate. (See above for an example)

Execute the following REST API to config the client cert auth.

```
curl -X POST -u Administrator:password -d@client_cert_auth.json
http://localhost:8091/settings/clientCertAuth
```

Each request will take 3 file paths typically deployed to the local node machines -

1. Client key (let's call it client.key)
2. Root CA certificate (let's call it ca.pem)
3. Certificate chain file based on the supported CA hierarchies (chain.pem)

A sample N1QL query that uses client-cert authorization -

```
curl --cacert ./root/ca.pem --cert-type PEM --key-type PEM
--cert ./client/client/chain.pem --key ./client/client/client.key
https://localhost:18093/query/service -d "statement=select * from
system:keyspaces"
```

COUCHBASE N1QL

Enabling Engagement

START A REVOLUTION

query.couchbase.com