SQL++ demo

Prerequisites

Create a free Couchbase Capella account.

Create a Project:

You have no projects, w

Create Pr

Create Project

Project Name *

Workshop

You can rename this at any time.

Create Project

Then, click on the project name and click `Create cluster` button. That opens a cluster creation dialog, simply hit the blue `Create cluster` button:

Create Cluster

1. Cluster Option

Choose an option to automatically configure

Single Node

REQUIRES CREDIT CARD OR ACTIVATION ID

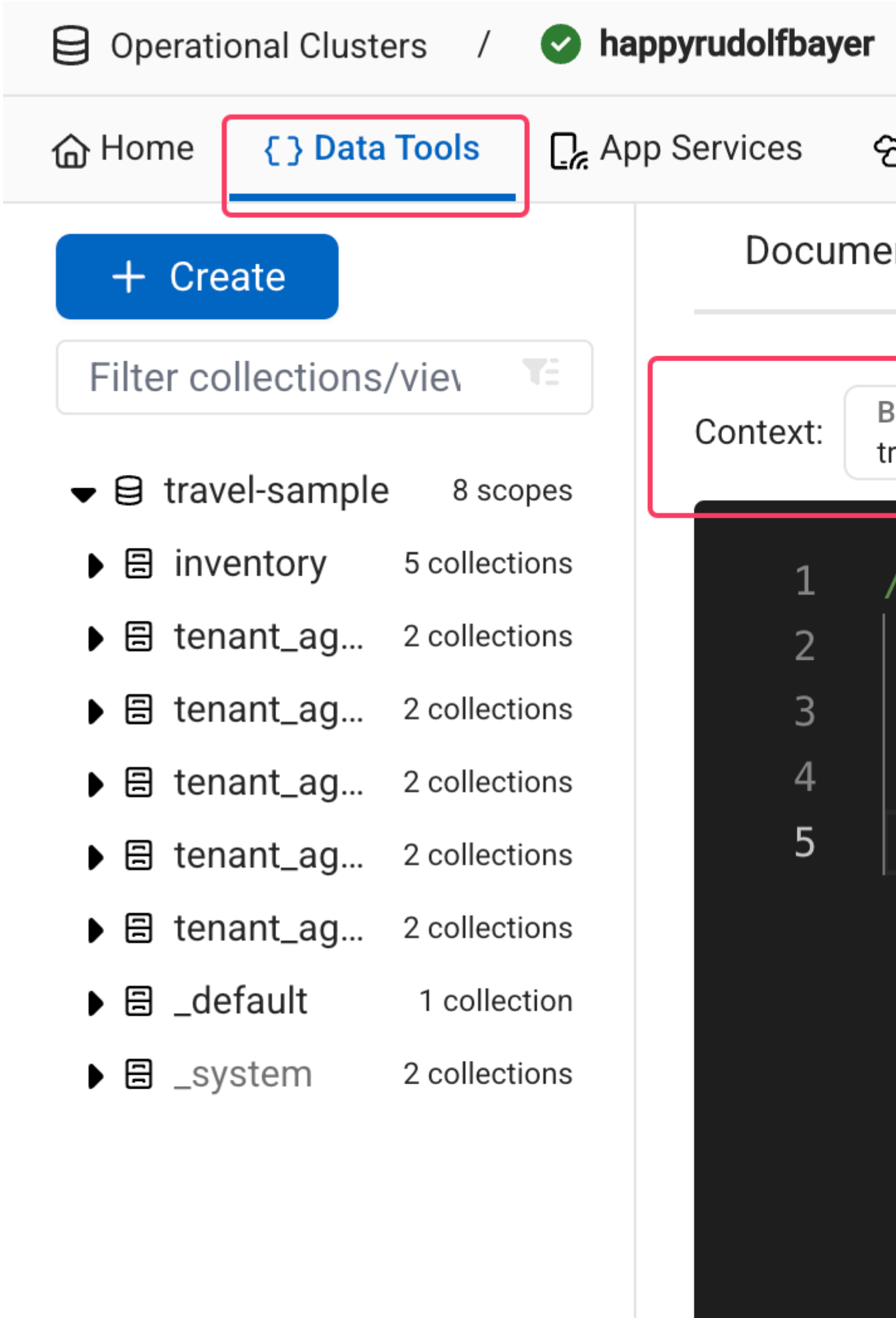
Get the lowest cost and streamlined options. E
more about Capella features.

2. Cluster Details and Cloud Servic

Give your cluster a name and description.

After 5 minutes your cluster is ready to be used.

Click on the cluster name and then go to the `Data Tools` -> `Query`. Set the context to the bucket `travel-sample` and the scope `inventory`:


Operational Clusters / happyrudolfbayer
Home
Data Tools
App Services
Create
Filter collections/viev
travel-sample 8 scopes
inventory 5 collections
tenant_ag... 2 collections
tenant_ag... 2 collections
tenant_ag... 2 collections
tenant_ag... 2 collections
tenant_ag... 2 collections
_default 1 collection
_system 2 collections
Context:
1
2
3
4
5

Introduction

Couchbase, being a JSON based data platform, is organising the data differently than RDBMS:

Relational Model	Couch
Server	Clus
Database	Buc
Schema	**Sco**
Table	**Colle**
Row	Document (JS

The language is very similar, so let's have a look what's similar to ANSI SQL and what's different.

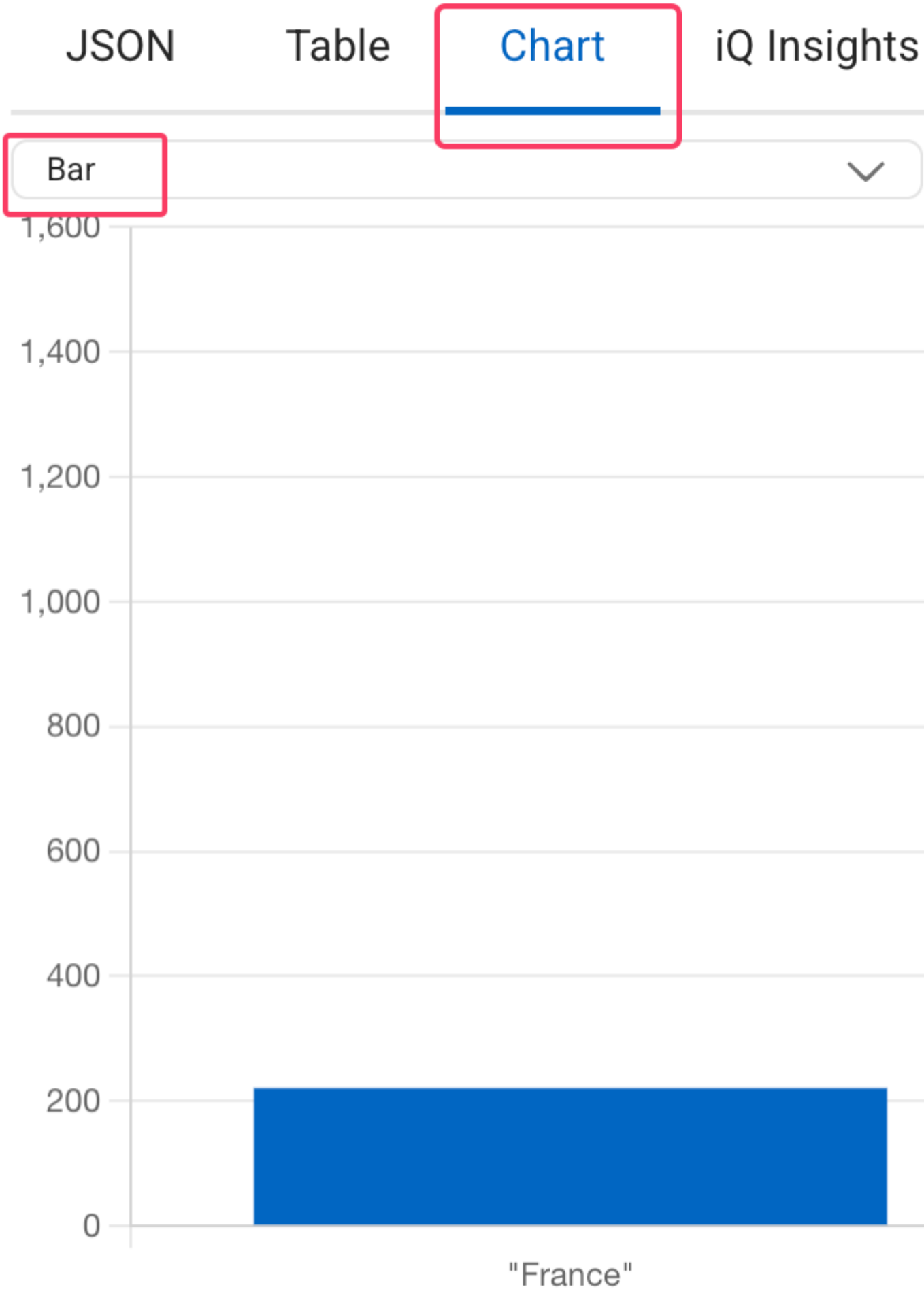
Select the name, IATA code, and ICAO code for airlines located in the United States, limiting the result to 5 entries.

`sql SELECT name, iata, icao FROM airline WHERE country = "United States" LIMIT 5;` That looks really familiar, because there's no difference between SQL and SQL++: we're `SELECT`-ing some fields `FROM` the collection (remember, that is equal to a table in RDBMS world) `airline`.
`WHERE` and `LIMIT` are exactly the same.

Count the number of airports in each country, ordering the results by the airport count in descending order and limiting to the top 5 countries.

`sql SELECT country, COUNT(*) AS airport_count FROM airport GROUP BY country ORDER BY airport_count DESC` `SELECT`, `FROM`, `GROUP BY`, `ORDER BY` - we're used to those from SQL already. No difference.

But let's see how to visualise the data we're receiving as the result. Click `Chart` and select `Bar` as the chart type. Choose `country` as X-axis and `airport_count` as Y-axis:


JSON
Table
Chart
iQ Insights
Bar
1,600
1,400
1,200
1,000
800
600
400
200
0
"France"

In the route keyspace, flatten the schedule array to get details of the flights on Monday (1).

`sql SELECT route.sourceairport, route.destinationairport, sched.flight, sched.utc FROM route UNNEST schedule sched WHERE sched.day = 1 LIMIT 3;` **Explaination**

The route documents look like this (simplified): `json { "type": "route", "sourceairport": "JFK", "destinationairport": "LAX", "schedule": [ { "flight": "AA100", "day": 1, "utc": "10:00" }, { "flight": "AA101", "day": 2, "utc": "12:00" } ] }` Notice that the schedule field is an array of objects—each object represents a flight on a specific day and time.

What does `UNNEST` do here?

- **UNNEST** schedule sched takes the schedule array from each route document and flattens it.
- For each element in the `schedule` array, it creates a new row in the result, with sched representing each schedule entry.

Summary

- **UNNEST** is used to flatten the schedule array so you can work with each flight schedule as a separate row.
- This makes it easy to filter, select, and display individual schedule entries, not just the parent route document.
- **UNNEST** lets you treat each item in the schedule array as its own row in your query results, making it easy to filter and select specific flights.

List only airports in Toulouse which have routes starting from them, and nest details of the routes.

`sql SELECT * FROM airport a INNER NEST route r ON a.faa = r.sourceairport WHERE a.city = "Toulouse" ORDER BY a.airportname;` **Explaination**

`INNER NEST` is a SQL++ join operation that combines documents from two different datasets based on a specified condition.
Unlike a regular `JOIN`, `INNER NEST` embeds the matching documents from the second dataset into an array within the first dataset.

Context: *airport* and *route* collections in the `travel-sample` bucket.

- *airport*: information about airports, including their FAA code (faa), city, and airport name.
- *route*: information about flight routes, including the source airport (sourceairport) and destination airport.

What `INNER NEST route r ON a.faa = r.sourceairport` does:

- This is where the magic happens. It finds all route documents (r) where the sourceairport matches the faa code of the airport document (a).
- Instead of creating a flat, joined result (like a regular JOIN), it nests the matching route documents into an array within the airport document.

Calculate the number of airlines per country, showing the distribution of airlines across different countries.

```
sql SELECT a.name, a.country, COUNT(*) OVER (PARTITION BY
a.country) AS country_airline_count FROM airline a ORDER BY
country_airline_count DESC LIMIT 10;
```

Explanation

The line of interest here is `COUNT(*) OVER (PARTITION BY a.country) AS country_airline_count`.
This is a `window function`. Here's what it does:

- For each airline, it counts **how many airlines are in the same country**.
- `PARTITION BY a.country` means the count is calculated separately for each country.
- The result is a new column, *country*airline*count*, showing the total number of airlines in that airline's country.

Key Point: What Does the `Window Function` Do?

For each airline, it counts the total number of airlines in that country. The result is not grouped (like with `GROUP BY`), so you still see each airline as a separate row, but with the country's total airline count attached.

Retrieve airport names and geo-coordinates, filtering based on latitude and longitude ranges.

```
sql SELECT ap.name, ap.geo.lat, ap.geo.lon FROM airport ap WHERE
ap.geo.lat BETWEEN 40 AND 50 AND ap.geo.lon BETWEEN -80 AND -70
LIMIT 5;
```

Explanation

Airport documents have a nested structure for geographical information:

```
json { "type": "airport", "faa": "JFK", "airportname": "John F.
Kennedy International Airport", "geo": { "lat": 40.6413, "lon":
-73.7781 }, "city": "New York", "country": "United States" }
```
Notice the geo field, which is an object containing `lat` (latitude) and `lon` (longitude).

The line of interest is `WHERE ap.geo.lat BETWEEN 40 AND 50 AND ap.geo.lon BETWEEN -80 AND -70`:

- Filters the airport documents based on their latitude and longitude.
- `ap.geo.lat BETWEEN 40 AND 50`: Selects airports with latitude between 40 and 50 degrees.
- `ap.geo.lon BETWEEN -80 AND -70`: Selects airports with longitude between -80 and -70 degrees.

Key Point: Accessing Nested Fields

In SQL++, you use dot notation (.) to access fields within nested objects.
`ap.geo.lat` means "go to the ap document, then go to the geo field, and then get the lat field."

Join airline and route on the IATA code, groups by airline name, and counts the number of routes per airline.

```
sql SELECT a.name AS airline, COUNT(r.airline) AS route_count
FROM `travel-sample`.inventory.airline a JOIN `travel-
sample`.inventory.route r ON r.airline = a.iata WHERE a.callsign
IS NOT MISSING GROUP BY a.name ORDER BY route_count DESC LIMIT 5;
```

Explaination

- **SELECT**: `a.name AS airline` - returns the airline name
- **FROM**: `airline a` - references the airline documents in the inventory scope, aliased as `a`.
- **JOIN**: joins with `route r` (aliased as `r`). Join condition: `r.airline = a.iata` - links routes to airlines using the airline's IATA code.
- **WHERE**: `a.callsign IS NOT MISSING` - filters to only include airlines that have a callsign field (excludes null/missing values)
- **COUNT**: `COUNT (r.airline) AS route_count` - counts how many routes each airline operates
- **GROUP BY**: groups results by airline name to aggregate route counts per airline
- **ORDER BY** and **LIMIT**: sorts by route count in descending order and returns only the top 5 airlines

Note the execution time (`ELAPSED`), it's over a second! That's too much, let's have a look where the time has been spent.
Click on `Plan` and zoom in to see the orange bubbles (we're not interested in the green ones):

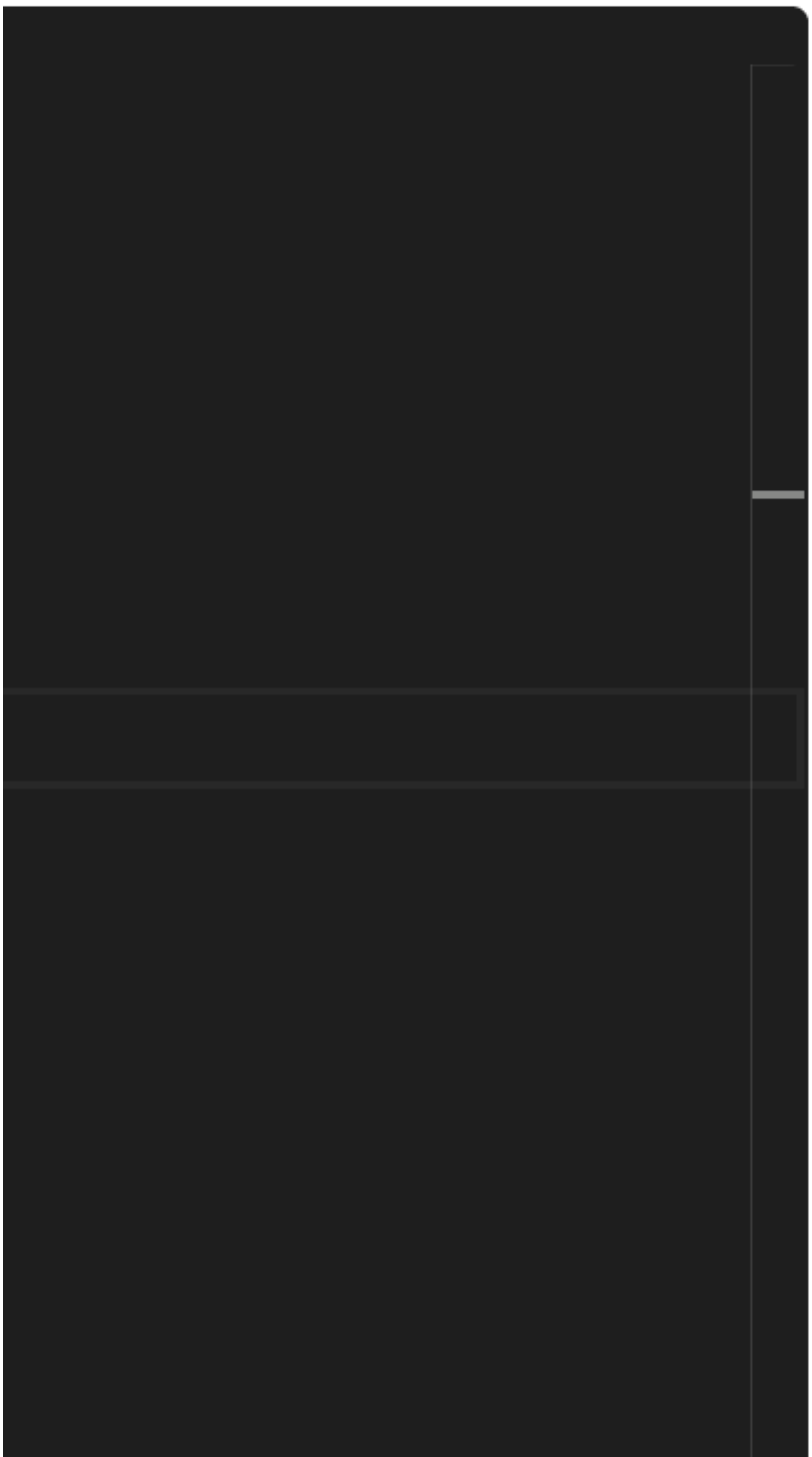
Documents Query Indexes Se

Context: Bucket travel-sample

```
1  SELECT a.name AS airline, C
2  FROM `travel-sample`.invent
3  JOIN `travel-sample`.invent
4  WHERE a.callsign IS NOT MIS
5  GROUP BY a.name
6  ORDER BY route_count DESC
7  LIMIT 5;
```

Note, the most time is lost in the `Fetch` phase. The system is using a primary index, which isn't a good idea. Let's do something about it!
Click the blue `Index Advice` button on the right. All it takes to improve the situation is to `Build suggested` index:

Recent Queries (1/1) +



Index Advic

Main Query A

⚠ The inde
this que

Current Inc

CREATE PRI
airline_prima
le`.`inventory

CREATE PRI
route_primar
e`.`inventory

Suggested

CREATE IND
ON `default`
airline`(`calls

CREATE IND
travel-sampl
`)

CREATE IND
ault`:`travel-s
airline` DESC

After it's built, re-run the query and observe the improved execution time:

Documents Query Indexes Sea

Context: Bucket travel-sample

```
1  SELECT a.name AS airline, CO
2  FROM `travel-sample`.invento
3  JOIN `travel-sample`.invento
4  WHERE a.callsign IS NOT MISS
5  GROUP BY a.name
6  ORDER BY route_count DESC
7  LIMIT 5;
```

Use pattern matching to find airlines whose names start with "A" or contain "Air" with a "B" prefix.

```
sql SELECT a.name, a.icao FROM `travel-sample`.inventory.airline
a WHERE (a.name LIKE "A%" OR REGEXP_CONTAINS(a.name,
"^[Bb].*[Aa]ir")) ORDER BY a.name LIMIT 10;
```

Explanation

- **FROM**: `FROM airline a` - references airline documents in the inventory scope, aliased as `a`.
- **WHERE** clause with *compound* conditions: `a.name LIKE "A%"` - finds airlines whose names start with the letter "A"
  OR
  `REGEXP_CONTAINS(a.name, "^[Bb].*[Aa]ir")`: uses regex to find airlines whose names either start with *"B"* or *"b"* (`^[Bb]`), have any characters in between (`.*`), end with *"Air"* or *"air"* (`[Aa]ir`)
- **SELECT**: `a.name` - returns the airline name, `a.icao` - returns the airline's ICAO code
- **ORDER BY** & **LIMIT**: sorts results alphabetically by airline name and returns only the first 10 matches

Categorize airports by region based on their country and counts the number of airports in each region.

```
sql SELECT ap.name, ap.country, CASE WHEN ap.country IN ["United
States", "Canada", "Mexico"] THEN "North America" WHEN ap.country
IN ["United Kingdom", "France", "Germany", "Spain", "Italy"] THEN
"Europe" WHEN ap.country IN ["China", "Japan", "India"] THEN
"Asia" ELSE "Other Regions" END AS region, COUNT(*) AS
airport_count FROM `travel-sample`.inventory.airport ap GROUP BY
ap.name, ap.country, region ORDER BY airport_count DESC LIMIT 10;
```

Explanation

- **FROM**: `airport ap` - references airport collection in the inventory scope, aliased as `ap`
- **SELECT**
  - `ap.name`: returns the airport name
  - `ap.country`: returns the country of the airport
  - **CASE ... END AS region** creates a new field called "region" based on the airport's country:
    - If the country is in *"United States", "Canada", "Mexico"*, the region is *"North America"*
    - If the country is in *"United Kingdom", "France", "Germany", "Spain", "Italy"*, the region is *"Europe"*
    - If the country is in *"China", "Japan", "India"*, the region is *"Asia"*
    - Otherwise, the region is *"Other Regions"*
    - **COUNT(*) AS airport_count**: counts the number of airports for each combination of airport name, country, and region
- **GROUP BY** & **LIMIT**: `ORDER BY airport_count DESC` sorts the results by the airport count in descending order, so the airport names with the most occurrences are listed first. `LIMIT 10` returns only the top 10 documents.

Calculate the distance between each hotel and New York City using geo-coordinates.

```
sql SELECT h.name AS hotel_name, h.address.city AS city,
h.geo.lat AS latitude, h.geo.lon AS longitude,
ROUND( DEGREES(ACOS( SIN(RADIANS(h.geo.lat)) *
SIN(RADIANS(40.7128)) + COS(RADIANS(h.geo.lat)) *
COS(RADIANS(40.7128)) * COS(RADIANS(h.geo.lon - (-74.0060))) )) *
69.09 ) AS miles_from_nyc FROM `travel-sample`.inventory.hotel h
WHERE h.geo IS NOT MISSING AND h.geo.lat IS NOT MISSING AND
h.geo.lon IS NOT MISSING ORDER BY miles_from_nyc LIMIT 10;
```

Explaination

- **FROM**:
  References hotel documents in the `inventory` scope: `travel-sample.inventory.hotel h` (aliased as `h`).

- **SELECT**:

  - `h.name AS hotel_name`: returns the hotel name.
  - `h.address.city AS city`: returns the city from the nested address field.
  - `h.geo.lat AS latitude`: returns the latitude from the nested geo field.
  - `h.geo.lon AS longitude`: returns the longitude from the nested geo field.
  - `ROUND(DEGREES(ACOS(...)) * 69.09) AS miles_from_nyc`: calculates the distance from New York City (latitude 40.7128, longitude -74.0060) using the Haversine formula. The result is rounded and converted to miles.

- **WHERE**: filters out hotels missing geo-coordinates:

  - `h.geo IS NOT MISSING`
  - `h.geo.lat IS NOT MISSING`
  - `h.geo.lon IS NOT MISSING`

- **ORDER BY & LIMIT**:

  - `ORDER BY miles_from_nyc`: Sorts results by distance from NYC, closest first.
  - `LIMIT 10`: Returns only the 10 closest hotels.

The CTE (TopAirports) filters countries with more than 50 airports. The main query joins these countries to airlines and aggregates airline names per country.

```
sql WITH TopAirports AS ( SELECT ap.country, COUNT(*) AS
airport_count FROM `travel-sample`.inventory.airport ap GROUP BY
ap.country HAVING COUNT(*) > 50 ) SELECT ta.country,
ta.airport_count, ARRAY_AGG(a.name) AS airlines_in_country FROM
TopAirports ta JOIN `travel-sample`.inventory.airline a ON
ta.country = a.country GROUP BY ta.country, ta.airport_count
ORDER BY ta.airport_count DESC LIMIT 5;
```

Explanation

- **WITH Clause (CTE)** creates a temporary result set called `TopAirports`:
  - `SELECT ap.country, COUNT(*) AS airport_count`: counts airports per country
  - `FROM travel-sample.inventory.airport ap`: references airport documents
  - `GROUP BY ap.country`: groups results by country
  - `HAVING COUNT(*) > 50`: filters to only include countries with more than 50 airports
- **FROM** uses the CTE as the primary data source: `FROM TopAirports ta` (aliased as `ta`)
- **JOIN** connects countries with their airlines: `JOIN travel-sample.inventory.airline a ON ta.country = a.country` links countries from the CTE with airline documents based on matching country names.
- **SELECT**:
  - `ta.country`: returns the country name from the CTE
  - `ta.airport_count`: returns the airport count from the CTE
  - `ARRAY_AGG(a.name) AS airlines_in_country`: aggregates all airline names into an array for each country
- **GROUP BY** groups results by country and airport count. `GROUP BY ta.country, ta.airport_count`: this allows `ARRAY_AGG()` to collect all airlines per country.
- **ORDER BY & LIMIT**:
  - `ORDER BY ta.airport_count DESC`: Sorts countries by airport count, highest first
  - `LIMIT 5`: Returns only the top 5 countries with the most airports

Graph traversal: a direct flight from LAX to JFK.

```
sql WITH FlightPath AS ( SELECT [source.faa, destination.faa] AS route, destination.faa AS lastStop, r.airline, r.schedule, 1 AS depth FROM route AS r JOIN airport source ON r.sourceairport = source.faa JOIN airport destination ON r.destinationairport = destination.faa WHERE source.faa = "LAX" ) SELECT fp.route, fp.airline, ARRAY s FOR s IN fp.schedule END AS schedule FROM FlightPath AS fp WHERE fp.lastStop = "JFK" AND fp.depth = 1;
```

Explanation

This query finds all direct (non-stop) flight routes from Los Angeles International Airport (LAX) to John F. Kennedy International Airport (JFK) using a graph-style traversal pattern.
The query demonstrates the SQL++ ability to:

- Use CTEs to model graph traversal patterns (flight paths)
- Join collections multiple times for source and destination relationships
- Build and filter arrays to represent paths

- Filter for direct connections using a "depth" field
- Aggregate and return schedule data as arrays

- **WITH Clause (`FlightPath` CTE)**: defines a temporary result set representing all direct flight paths starting from LAX.

    - `SELECT [source.faa, destination.faa] AS route`: creates an array representing the route from the source airport to the destination airport using their FAA codes.
    - `destination.faa AS lastStop`: stores the FAA code of the destination airport as the last stop in the route.
    - `r.airline`: includes the airline operating the route.
    - `r.schedule`: includes the schedule information for the route.
    - `1 AS depth`: sets the depth to 1, indicating a direct (single-leg) flight.
    - `FROM route AS r`: uses the `route` collection as the main data source.
    - `JOIN airport source ON r.sourceairport = source.faa`: joins the `airport` collection to get details about the source airport.
    - `JOIN airport destination ON r.destinationairport = destination.faa`: joins the `airport` collection again to get details about the destination airport.
    - `WHERE source.faa = "LAX"`: filters to only include routes that start at LAX.

- **Main SELECT**:

    - `fp.route`: returns the array representing the route (e.g., `["LAX", "JFK"]`).
    - `fp.airline`: returns the airline operating the route.
    - `ARRAY s FOR s IN fp.schedule END AS schedule`: returns the schedule for the route as an array.

- **WHERE**:

    - `fp.lastStop = "JFK"`: filters to only include routes where the destination is JFK.
    - `fp.depth = 1`: ensures only direct flights (no layovers) are included.

Graph traversal: a flight from LAX to JFK over MIA.

```sql WITH RECURSIVE FlightPath AS ( SELECT [source.faa, destination.faa] AS route, destination.faa AS lastStop, [route.airline] AS airlines, [route.schedule] AS schedules, 1 AS depth FROM route JOIN airport source ON route.sourceairport = source.faa JOIN airport destination ON route.destinationairport = destination.faa WHERE source.faa = "LAX"

UNION ALL

SELECT
    ARRAY_APPEND(fp.route, destination.faa) AS route,
    destination.faa AS lastStop,
    ARRAY_APPEND(fp.airlines, route.airline) AS airlines,
    ARRAY_APPEND(fp.schedules, route.schedule) AS schedules,
    fp.depth + 1 AS depth
FROM FlightPath fp
JOIN route ON fp.lastStop = route.sourceairport
JOIN airport destination ON route.destinationairport = destination.faa
WHERE destination.faa != "LAX" AND fp.depth < 2
```

) OPTIONS {"levels": 2} SELECT route, airlines, schedules FROM FlightPath WHERE route[1] = "MIA" AND lastStop = "JFK" AND depth = 2; ```

Explaination

This query uses a recursive Common Table Expression (CTE) to find all two-leg flight paths from Los Angeles International Airport (LAX) to John F. Kennedy International Airport (JFK) that connect through Miami International Airport (MIA). It demonstrates SQL++ ability to:

- Use recursive CTEs for graph traversal and multi-hop pathfinding
- Build and extend arrays to represent routes, airlines, and schedules
- Join collections multiple times to follow connections between airports
- Filter for specific intermediate stops and path lengths
- Model real-world graph problems (like flight connections) in a document database

- **WITH RECURSIVE Clause (`FlightPath` CTE)**: defines a recursive structure to build flight paths step by step.
    - **Base Case (First Leg):**
        - `SELECT [source.faa, destination.faa] AS route`: starts the route as an array with the source and destination FAA codes.
        - `destination.faa AS lastStop`: sets the destination as the current last stop.
        - `[route.airline] AS airlines`: starts an array with the airline for the first leg.
        - `[route.schedule] AS schedules`: starts an array with the schedule for the first leg.
        - `1 AS depth`: indicates this is the first leg of the journey.
        - `FROM route ... WHERE source.faa = "LAX"`: finds all routes departing from LAX.
    - **Recursive Case (Second Leg)**:
        - `ARRAY_APPEND(fp.route, destination.faa) AS route`: extends the route array by adding the next destination.
        - `destination.faa AS lastStop`: updates the last stop to the new destination.
        - `ARRAY_APPEND(fp.airlines, route.airline) AS airlines`: adds the airline for the new leg to the airlines array.
        - `ARRAY_APPEND(fp.schedules, route.schedule) AS schedules`: adds the schedule for the new leg to the schedules array.
        - `fp.depth + 1 AS depth`: increments the depth to indicate the number of legs.
        - `FROM FlightPath fp ... WHERE destination.faa != "LAX" AND fp.depth < 2`: joins onward flights from the current last stop, avoids cycles back to LAX, and limits the path to two legs.
- **OPTIONS {"levels": 2}**: limits the recursion to two levels (i.e., two flight legs).
- **Main SELECT Clause:**
    - `route`: returns the full route as an array of airport codes (e.g., `["LAX", "MIA", "JFK"]`).
    - `airlines`: returns an array of airlines for each leg.

    - `schedules`: returns an array of schedules for each leg.

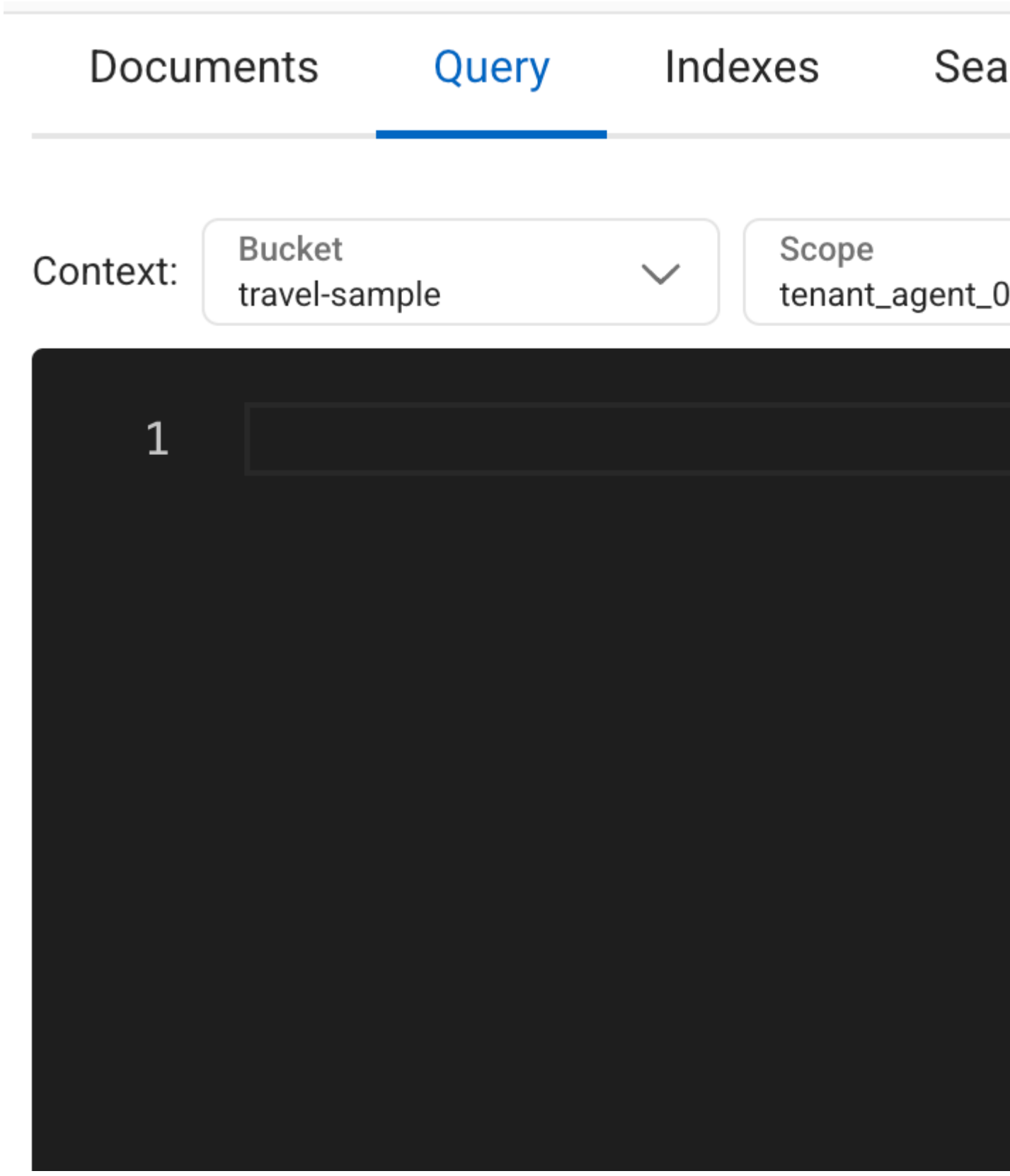
- **WHERE Clause:**

    - `route[1] = "MIA"`: ensures the first stop after LAX is MIA (Miami).
    - `lastStop = "JFK"`: ensures the final destination is JFK (New York).
    - `depth = 2`: ensures only two-leg journeys are included.

Transactions

Did you know that Couchbase supports ACID compliant multi-document transactions?
Let's have a look.
Before we run one, we need to change some query settings, as our Couchbase Capella instance is a one-node cluster.

Click `Options` in the query editor:



And then set

- `Transaction timeout` to `120 seconds`
- `Scan Consistency` to `not_bounded`
- Add a *Named Parameter* `durability_level` with the value `"none"`, don't forgtet the value's quotes!

Hit Save

Query Options

- [x] Collect query timings
- [x] Use Cost-Based Optimizer
- [ ] Don't save query history

Max Parallelism

Query Timeout (sec)
600

Transaction Timeout (sec)
120

Scan Consistency
not_bounded

Positional Parameters

Named Parameters

name 0
durability_level

val
"no

Now we're good to go. I suggest a simple transaction:

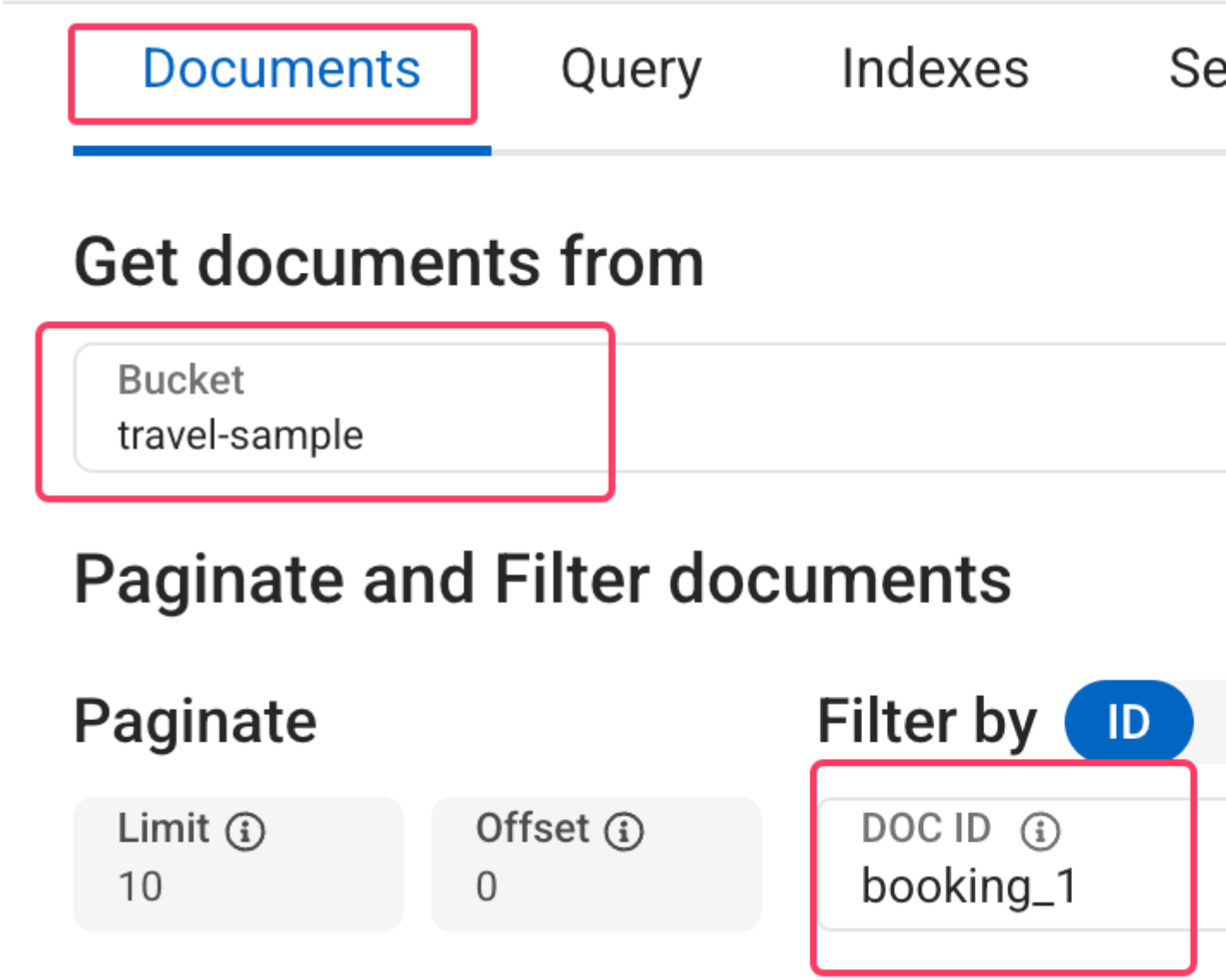
```sql BEGIN TRANSACTION;

INSERT INTO `travel-sample`.tenant*agent*00.bookings (KEY, VALUE) VALUES ("booking*1", { "customer*id": "cust*123", "hotel*id": "hotel*456", "check*in*date": "2023-10-01", "check*out_date": "2023-10-05", "status": "confirmed" });

COMMIT; ```

**Verification**

Once finished, let's switch to the `Documents` tab, set the context to the bucket `travel-sample`, the scope `tenant_agent_00` and the collection `booking`.
Fetch the document by its key - simply type `booking_1` into the `DOC ID` field and hit `Get Documents`:




Click the document's key to open.

**Explaination**
```

This transaction demonstrates SQL++'s ability to:

- Use ACID transactions for data consistency
- Insert documents with explicit key-value pairs
- Work with multi-tenant data structures (tenant-specific scopes)
- Handle JSON document creation with nested field structures
- Ensure data integrity through transaction boundaries

- **BEGIN TRANSACTION:**
  Starts a new transaction to ensure atomicity and consistency.
  All operations within the transaction will either all succeed or all fail together.

- **INSERT Statement:**

  - **Target Collection:**
    `travel-sample.tenant_agent_00.bookings`: Inserts into the `bookings` collection within the `tenant_agent_00` scope of the `travel-sample` bucket.

  - **Document Structure:**
    Uses `(KEY, VALUE)` syntax to explicitly specify both the document key and content:

    - **KEY:** `"booking_1"` - the unique document identifier
    - **VALUE:** JSON object containing the booking details:
      - `"customer_id": "cust_123"`: reference to the customer making the booking
      - `"hotel_id": "hotel_456"`: reference to the booked hotel
      - `"check_in_date": "2023-10-01"`: guest arrival date
      - `"check_out_date": "2023-10-05"`: guest departure date
      - `"status": "confirmed"`: current booking status

- **COMMIT:**
  Finalizes the transaction, making all changes permanent and visible to other operations.
  If any error occurred during the transaction, this would trigger a rollback instead.

Time Series Querying

For this exercise you will need to download the regular Time Series dataset time*series*regular.json (that can be found in this repository). The dataset contains daily values for minimum and maximum temperature in 2024 for several locations within Munich. The data in this dataset were already pre-converted into Time Series documents ready for querying in Couchbase (each JSON document contains data per location per month).

Import the Time Series dataset into Capella

In Capella UI: use the Import tool to import the regular Time Series dataset:

1. Go to the `Data Tools` tab → `Import` tab.
2. In the Import tab, select `Load from your browser` and choose the file 'time*series*regular.json' from your computer (please download it from this repository).
3. In the `Choose your target` step, click on `+ Create new target collection` and proceed to create a new bucket with `time_series` as the New

Bucket Name, `time` as the New Scope Name and `weather` as the New Collection Name.

Create a Bucket, Scope, and Collec

Buckets are the top-level containers
Add **scopes** to a bucket to group rel
Use a **collection** as the final contain

1 Bucket

New | Existing

Create a bucket with all default settings

New Bucket Name *

time_series

You cannot change the bucket name later

Memory Quota (MiB)

100

2 Scope

Use system generated _default fo

4. Click `Create`.
5. Select `Field` and `cbmid` as the Field name to use the value from the cbmid field inside of imported data as document keys.
6. Click `Import`.

In Capella UI: switch from the `Import` tab to the `Documents` tab and check that the documents were imported correctly.

Documents | Query | Indexes | Sea

Get documents from

Bucket
time_series

Paginate and Filter documents

Paginate

Limit
10

Offset
0

Filter by

ID | ID

DOC ID

10 documents | limit 10 | offset 0

DOC ID
temp:munich:2024:01Arnulf
temp:munich:2024:01Erhardt

Query the regular Time Series data using SQL++

In Capella UI: switch from the `Documents` tab to the `Query` tab and set the Query Context: `time_series` as the Bucket and `time` as the Scope. In Capella Query Workbench: create a Secondary index (GSI) for the imported Time Series data in order to query it later.

```
sql CREATE INDEX idx_temp ON weather(location, ts_end, ts_start);
```

Show the daily low and high temperatures for the time period from Jan, 1st 2024 till Jan 10th 2024 for the 'Olympia' location.

In Capella Query Workbench:

```
sql -- Define the start and end range for the query WITH
range_start AS (1704067200000), -- Start timestamp in
milliseconds (01.01.2024) range_end AS (1704927600000) -- End
timestamp in milliseconds (11.01.2024) -- Select the required
fields from the weather collection SELECT
MILLIS_TO_TZ(t._t,"UTC") AS day, -- Convert timestamp to UTC
t._v0 AS low, -- Low temperature t._v1 AS high -- High
temperature FROM weather AS d -- Alias `for` the weather
collection UNNEST _timeseries(d, {"ts_ranges": [[range_start,
range_end]]}) AS t -- Unnest the time series data with the
specified range WHERE d.location = 'Olympia' -- Filter by
location AND (d.ts_start <= range_end -- Ensure the data's start
timestamp is within the range AND d.ts_end >= range_start) --
Ensure the data's end timestamp is within the range -- Order by
timestamp ORDER BY t._t;
```

Explaination

- This query uses a CTE (common table expression) to store the date-time range.
- For each time point, the _TIMESERIES function calculates the date-time stamp `_t` and returns the values `_v0` and `_v1`.
- The query adds aliases to the data returned by the _TIMESERIES function and converts the date-time stamp to a readable date-time string.