The Couchbase Blog
Couchbase, the NoSQL Database

Optimizing Database Pagination using Couchbase N1QL.

Keshav Murthy on November 2, 2017

Background:

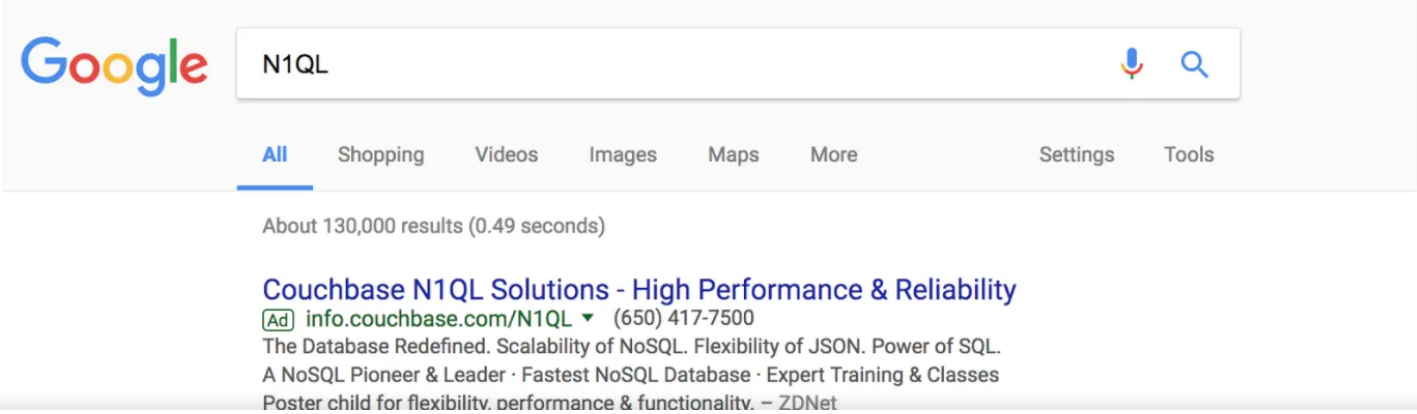
How does Google do it? When you google something or anything, it gives you back top relevant results, tells you an approximate number of documents for your topic — all under a second. Here are some high-level pointers: https://www.google.com/search/howsearchworks/algorithms/

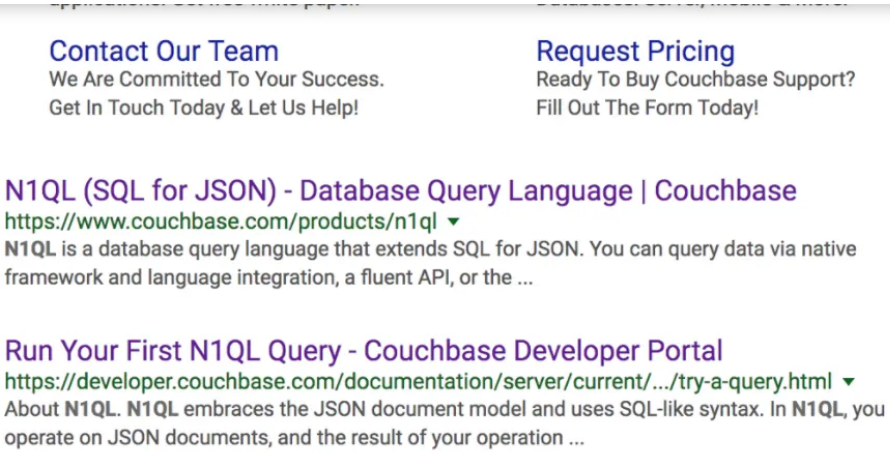
Enterprise applications have the same need, albeit with more complex search, search, sort, and pagination criteria.

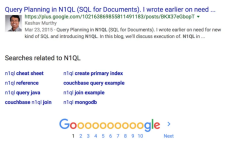
Let's look at the Google pagination behavior to understand the basic pagination features. Then, we'll discuss implementing pagination in an enterprise application, step by step.

Google Pagination:

Google for "N1QL".







The page you get back, the result of the search has the following information.

1. Number of pages matched: 130,000
2. Search was executed in 0.49 seconds
3. The page includes advertisements. In this case, an ad from Couchbase. Naturally.
4. The first page of resultset: Links to 12 pages and few lines from each page.
5. Search related to “N1QL”. Few suggestions to related to searches
6. Links to next 10 pages of results and the link to the NEXT page.

Database Pagination

Pagination’s tasks is to retrieve and show **subset of the resultset.** The subset is determined by the pagination specification (how many rows per page) and the sort order of the query issued by the application. In database pagination, the application tries to exploit the characteristics and optimizations provided by the database management.

Let’s look at each of the pagination features we saw with Google and explore how you can implement and optimize queries in Couchbase.

In the following sections, we’ll focus on database pagination using Couchbase.

1. Counting the total results

3. Fetching the first page
4. Creating the links to next and other pages
5. Fetching the next or any other page.

We won't cover the Google ad selection or related search suggestions. They're distinct topics by themselves.

Note this article uses new features like index collation (ASC and DESC specification for each index key), offset pushdown and other optimizations in Couchbase 5.0.

Section 1. Counting the total results

Google returned the following:

About 130,000 results (0.49 seconds)

COUNT: Estimated number of pages 130,000

TIME: Amount of time it took to do the search. In this case, 0.49 seconds

In database pagination, both of these can be useful.

COUNT is useful to determine the number of next and previous links you need to generate when you render the results in the UI. Your pagination query itself may not return the total number of results because the optimizer will try to use the indexes and other techniques to limit the number of documents processes. That'll prevent the query from knowing the possible total number of documents in the resultset.

If the query has ORDER BY, it can sortCount in some cases. This tells you the total number of documents we sort, even though we return just one document.

When the query exploits index ordering to avoid the sort to evaluate an ORDER BY clause, the sortCount is unavailable.

```
cbq> select * from `travel-sample` where faa > "4AB" ORDER BY airportname OFFSET 1 LIMIT 1;
{
    "requestID": "709fe2fb-d124-4b0c-b2b4-1c235d3c6f12",
    "signature": {
        "*": "*"
    },
    "results": [
        {
            "travel-sample": {
```

```
                "faa": "ABI",
                "geo": {
                    "alt": 1791,
                    "lat": 32.411319,
                    "lon": -99.681897
                },
                "icao": "KABI",
                "id": 3718,
                "type": "airport",
                "tz": "America/Chicago"
            }
        }
    ],
    "status": "success",
    "metrics": {
        "elapsedTime": "40.111996ms",
        "executionTime": "40.087977ms",
        "resultCount": 1,
        "resultSize": 500,
        "sortCount": 1659
    }
}
```

This query below exploits the index on the field, faa to get the data in sorted order and push down the pagination (OFFSET 50 LIMIT 10) clause to the index scan. Therefore, sortCount is missing in the resultset.

```
cbq> select * from `travel-sample` where faa > "4AB" ORDER BY faa OFFSET 50 LIMIT 10;
{
    "requestID": "5bc38dd1-7285-41a5-80e3-0f1da23df178",
    "signature": {
        "*": "*"
    },
    "results": [
        {
            "travel-sample": {
                "airportname": "Andrews Afb",
                "city": "Camp Springs",
                "country": "United States",
                "faa": "ADW",
                "geo": {
                    "alt": 280,
                    "lat": 38.810806,
                    "lon": -76.867028
                },
                "icao": "KADW",
                "id": 3552,
                "type": "airport",
                "tz": "America/New_York"
            }
        },
...
    "status": "success",
    "metrics": {
        "elapsedTime": "4.167044ms",
        "executionTime": "4.143152ms",
        "resultCount": 10,
        "resultSize": 5033
    }
}
```

The IndexCountScan2 method (or IndexCountScan prior to 5.0) simply does the index scan and count the number of qualified documents and avoids all the data transfer from the indexer to query engine.

```
cbq> SELECT COUNT(faa) FROM `travel-sample` where  faa > "4AB";
{
    "requestID": "78a3aeae-4dd1-468c-a01c-38610fd87cf4",
    "signature": {
        "$1": "number"
    },
    "results": [
        {
            "$1": 1659
        }
    ],
    "status": "success",
    "metrics": {
        "elapsedTime": "2.945555ms",
        "executionTime": "2.920307ms",
        "resultCount": 1,
        "resultSize": 34
    }
}
```

Here's the query plan for this query to get COUNT.

```
cbq> EXPLAIN SELECT COUNT(faa) FROM `travel-sample` where  faa > "4AB";

                    {
                         "#operator": "IndexCountScan2",
                         "covers": [
                             "cover ((`travel-sample`.`faa`))",
                             "cover ((meta(`travel-sample`).`id`))"
                         ],
                         "index": "def_faa",
                         "index_id": "460bd5dad1c6c95d",
                         "keyspace": "travel-sample",
                         "namespace": "default",
                         "spans": [
                             {
                                 "exact": true,
                                 "range": [
                                     {
                                         "inclusion": 0,
                                         "low": "\"4AB\""
                                     }
                                 ]
                             }
```

Section 2. Timings and other metrics

Every query result has the basic metrics on the query execution.

```
    "metrics": {
        "elapsedTime": "40.111996ms",
        "executionTime": "40.087977ms",
```

elapsedTime is the duration of clock time server spent after receiving the query. This includes any wait time. **executionTime** is strictly the time spent executing the query. resultCount is the number of documents returned. resultSize is the number of bytes in the resultset. sortCount is the number of documents sorted before the pagination.

If the query does not include OFFSET or LIMIT, resultCount is the total number of documents in the resultset. When we need to sort the intermediate data, As mentioned before, sortCount will be missing if there is no sort performed to evaluate the ORDER BY.

Section 3. first page of the resultset

Let's focus on the first page first.

```
SELECT *
FROM `travel-sample` t
WHERE type = "hotel"
AND country = "United Kingdom"
AND ARRAY_LENGTH(public_likes) > 3
ORDER BY ARRAY_LENGTH(public_likes), ratings DESC
OFFSET 0
LIMIT 20;


CREATE INDEX idx_hotel_ctry_likes ON
`travel-sample`(country, ARRAY_LENGTH(public_likes))
WHERE type = "hotel"
```

Query using this index runs in about 30 milliseconds.

```
    "status": "success",
    "metrics": {
        "elapsedTime": "30.125957ms",
        "executionTime": "30.110732ms",
        "resultCount": 20,
        "resultSize": 181449,
        "sortCount": 238
    }
}
```

This index has the three predicates in the index. While the index does all the filtering, the query still has to get the complete result set, sort them and then project just the first page. In my machine, this runs in 30 milliseconds. I'd like to reduce this further so we can run lots of queries concurrently.

```
DROP INDEX `travel-sample`.idx_hotel_ctry_likes;
CREATE INDEX idx_hotel_ctry_likes_ratings ON `travel-sample`
        (country, ARRAY_LENGTH(public_likes), ratings DESC)
              WHERE type = "hotel"
```

Run the query again.

```
        "executionTime": "9.452752ms",
        "resultCount": 20,
        "resultSize": 181449
    }
```

For this query and index, the explain will show that LIMIT 20 has been pushed down to index scan.

```
          {
            "#operator": "IndexScan2",
            "index": "idx_hotel_ctry_likes_ratings",
            "index_id": "f7de95817c4dc84b",
            "index_projection": {
              "primary_key": true
            },
            "keyspace": "travel-sample",
            "limit": "20",
            "namespace": "default",
            "spans": [
              {
                "exact": true,
                "range": [
                  {
                    "high": "\"United Kingdom\"",
                    "inclusion": 3,
                    "low": "\"United Kingdom\""
                  },
                  {
                    "inclusion": 0,
                    "low": "3"
                  }
                ]
              }
            ],
```

This query runs under 10 milliseconds, by avoiding the full fetch and sort. The fastest way to sort is to avoid the sort itself.

Section 4: Creating the links to next and other pages

When the first page is rendered, Google also gives you link to the next page and other 10 subsequent pages. As we discussed in section 1, the total count of potential results can be obtained in multiple ways. Once you have the count, simply create the links with respective OFFSETs required for each page. To get the next page, we simply set the OFFSET to 20. To get each subsequent page or any random page, you simply calculate the OFFSET with (page# * the number of documents in a page). Of course, this OFFSET should be less than the total number of potential documents in the result set. We discussed getting this total count in section 1.

Example:

Second page: OFFSET 20 LIMIT 20;

Eight page: OFFSET 160 LIMIT 20;

Section 5: Fetching the next or any other page.

In the previous section, we discussed creating the links with correct pagination parameters. Once you have the first query, calculate the OFFSETs for subsequent pages, you have everything you need for issuing queries for all subsequent queries.

Retrieve the second page via the following query:

```
SELECT *
FROM `travel-sample` t
WHERE type = 'hotel'
AND country = 'United Kingdom'
AND ARRAY_LENGTH(public_likes) > 3
ORDER BY ARRAY_LENGTH(public_likes), ratings DESC
OFFSET 20
LIMIT  20;
```

You simply get each subsequent page (or a random page) by changing the OFFSET.

Starting with Couchbase 5.0, when the index can evaluate the complete predicate and the ORDER BY clause, both OFFSET and LIMIT are pushed down to index scan. This will make the index scan efficient by returning the only LIMITed number of rows after skipping the qualified rows as specified by OFFSET.

You should see a query plan like below:

```
            {
              "#operator": "IndexScan2",
              "index": "idx_hotel_ctry_likes_ratings",
              "index_id": "f7de95817c4dc84b",
              "index_projection": {
                "primary_key": true
              },
              "keyspace": "travel-sample",
              "limit": "20",
              "namespace": "default",
              "offset": "20",
              "spans": [
                {
                  "exact": true,
                  "range": [
                    {
                      "high": "\"United Kingdom\"",
                      "inclusion": 3,
                      "low": "\"United Kingdom\""
                    },
                    {
                      "inclusion": 0,
                      "low": "3"
```

```
27              ],
```

Even when you have to create an optimal index (like this query), the index still has to traverse qualified entries from OFFSET 0 to OFFSET NN to identify the index entries to return. When the offset value is large, this can be expensive. When the query does not have an appropriate index, the offset processing is even more expensive.

Conclusion

Even though we've optimized N1QL query processing and index scans for optimizations, you can still optimize these queries further when your use cases mainly "fetch next". This is a common and important scenario. Marks Winand and Lukas Eder have discussed keyset pagination method, which improves the performance even further. Their articles are in the reference section.

In the next article, I'll discuss the implementation of keyset pagination in Couchbase N1QL.

References:

1. https://www.slideshare.net/MarkusWinand/p2d2-pagination-done-the-postgresql-way
2. http://use-the-index-luke.com/sql/partial-results/fetch-next-page
3. https://blog.jooq.org/2013/10/26/faster-sql-paging-with-jooq-using-the-seek-method/

Posted in: Application Design, N1QL / Query

Tagged in: application, database, Index, JSON, N1QL, nosql, pagination, performance, SQL

Posted by Keshav Murthy

Keshav Murthy is a Vice President at Couchbase R&D. Previously, he was at MapR, IBM, Informix, Sybase, with more than 20 years of 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

Next →
N1QL Enhancements in Cou…

All Posts Website

Leave a reply

Comment

Notify me of follow-up comments by email.

Notify me of new posts by email.

Post Comment

Search our blog

Search here…