N1QL Tuning Guide

Couchbase
NoEQUAL

Notice and Disclaimer

The recommendations, best practice guides, tuning examples (together "Best Practices") as well as sample code, scripts (together "Sample Code", collectively with the Best Practices is "Content") contained herein are the property of Couchbase, Inc. ("Couchbase") and are provided for illustrative and instructional purposes only. The user of the Content acknowledges and accepts that the Content is not supported by any license agreement between Couchbase and the user.

The Content may not be reproduced, disseminated, sold, sub-licensed, assigned, rented leased, distributed or otherwise published, in whole or in part without prior written permission from Couchbase.

The user of the Source Code assumes the entire risk of any use it may make or permit to be made of the Source Code and is solely responsible for adequate protection and backup of its data. Couchbase reserves the right to make changes to the Source Code or Best Practices at any time without prior notice. ALWAYS thoroughly evaluate Sample Code using test data to ensure proper operation and confirm the Sample Code causes no adverse effects prior to use on live or production data.

Couchbase hereby reserves all rights in the Content under the copyright laws of the United States and applicable international laws, treaties, and conventions.

THE CONTENT HEREIN IS PROVIDED "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. WITHOUT LIMITING ANY OF THE FOREGOING AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL Couchbase OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) SUSTAINED BY YOU OR A THIRD PARTY, HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT ARISING IN ANY WAY OUT OF THE USE OF THE CONTENT, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. The foregoing shall not exclude or limit any liability that may not by applicable law be excluded or limited.

Use of or access to Couchbase products or services requires a separate license from Couchbase.

N1QL Tuning Guide

This document discusses goals for index and query tuning, as well as how to identify slowly performing and high resource consuming N1QL statements. We explain how to review and interpret an explain plan and provide tuning suggestions.

Sections

The performance of any system follows physics. The basic two rules can be (loosely) stated as:

1. Quantity: Less work is more performance.
2. Quality: Faster work is more performance.

Query processing is no different and it also tries to optimize both these factors in various forms and scenarios to bring efficiency. Each optimization is different and results in a different amount of performance benefit.

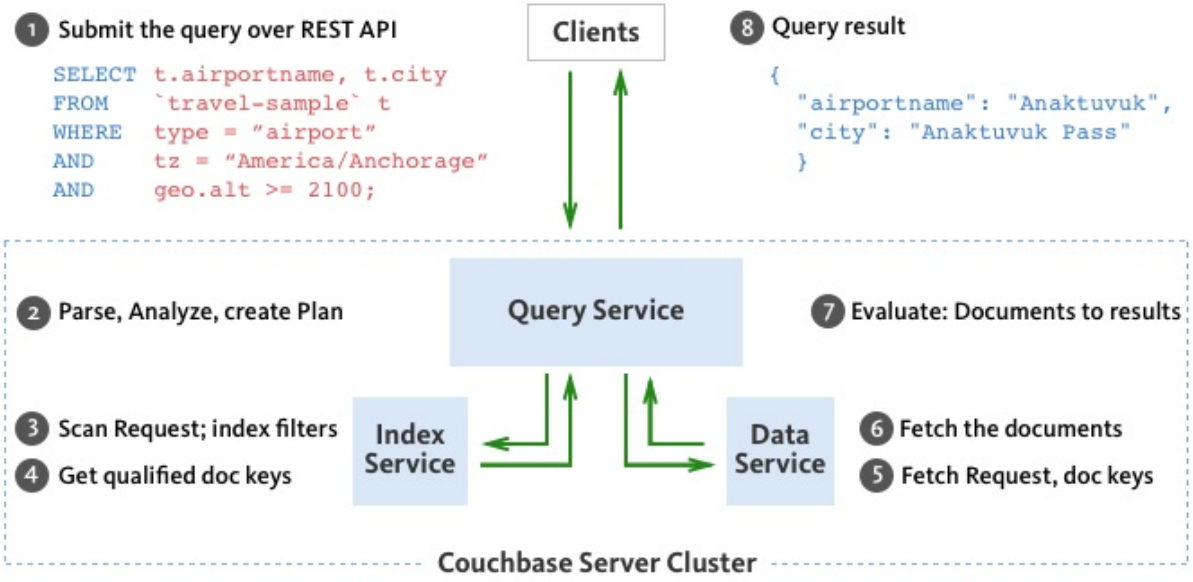
Tuning is iterative and involves the following basic steps:

1. Identifying the slowly performing or high resource consumption N1QL statements that are responsible for a large share of the application workload and system resources. Generally tuning the slower and most frequently used N1QL queries will yield the highest results. Additionally, depending on your response and SLA needs you will need to identify and tune specific queries. As in many scenarios generally, the Pareto principle applies to query tuning as well - 80% of your workload/performance problems are probably caused by 20% of your queries - focus and tune that 20% of your queries
2. Verify that the execution plans produced by the query optimizer for these statements are reasonable and expected. Note: Couchbase currently is a RULE based optimizer and not a COST based optimizer so key or index cardinality do not impact the choice of the index or creation of the overall query plan
3. Implement corrective actions to generate better execution plans for poorly performing SQL statements

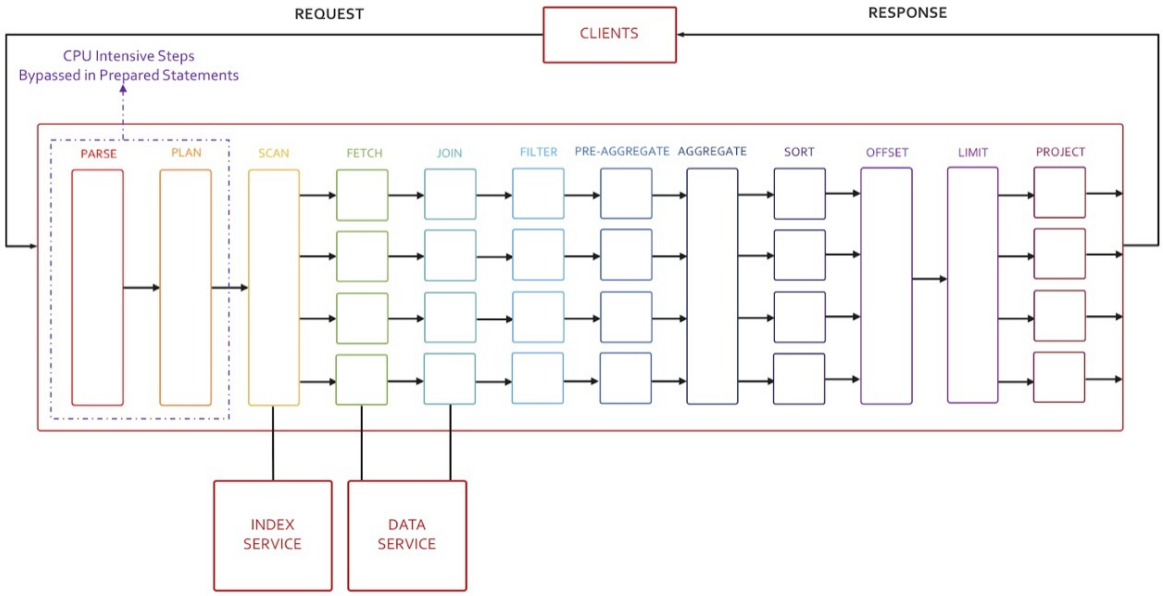
The previous steps are repeated until the query performance reaches a satisfactory level or no more statements can be tuned.

Understanding Query Workflow and Optimization

Applications and their drivers submit the N1QL query to one of the available query nodes. The Query node parses and 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 indices, the query node works with index and data nodes to retrieve and perform the select-join-project operations. Since Couchbase is a clustered database, you scale out data, index and query nodes to fit your performance and availability goals.



The figure below shows all the possible phases a query goes through during execution to return the results. Not all queries need to go through every phase. Some go through many of these phases multiple times. The Optimizer decides phases each query should execute. For example, the Sort phase will be skipped when there is no `ORDER BY` clause in the query; the scan-fetch-join phase executes multiple times to perform multiple joins. Some operations like query parsing and planning is done serially, while other operations like fetch, join, sort can be done in parallel.



The N1QL Optimizer analyzes the query and available access path options for each keyspace (bucket) in the query. For each query block, the planner needs to first select the access path for each bucket, determine the both join order and type.

Access Path Selection

1. **KeyScan access:** When specific document IDs (keys) are available, the Keyscan access method retrieves those documents. Any filters on that keyspace are applied to those documents. The Keyscan can be used when a keyspace is queried by specifying the document keys (USE KEYS modifier) or during join processing. The Keyscan is commonly used to retrieve qualifying documents for the inner keyspace during join processing.
2. **Index Count Scan:** Queries with a single projection of a `COUNT()` aggregate and do not contain any `JOIN` statements. The chosen index needs to be covered with a single range or equality predicates that can be to push down to the indexer. The argument passed to `COUNT()` needs to be constant or leading key.
3. **Covering Secondary Scan** Each satisfied index with the most number of index keys is examined for query coverage. Shortest covering index will be used.
4. **Regular Secondary Scan access:** A qualifying secondary index scan is used to first filter the keyspace and to determine the qualifying documents IDs. It then retrieves the qualified documents from the data store, if necessary. If the selected index has all the data to answer the query, N1QL avoids fetching the document altogether — this method is called the covering index scan. This is highly performant. Your secondary indexes should help queries choose cover index scans as much as possible. Indexes with the most number of matching index keys are used. When more than one index is qualified, an IntersectScan is used.
5. **UNNEST Scan:** Only array indexes are considered. And only queries with UNNEST clauses are considered.
6. **PrimaryScan access:** This method is chosen when documents IDs are not given and no qualifying secondary indexes are available for this keyspace. This access method is quite expensive and should be avoided especially in a production environment.

Index Selection

Before discussing how the N1QL Query Planner performs index selection, it is important to define the contract between N1QL Index Selection and the user. The contract does not change from release to release:

- The index chosen by N1QL will satisfy the query. That is N1QL will not choose an index whose definition can lead to wrong results. (Bugs outside of index definition and selection do not count)
- If there are one or more indexes that satisfy the query and are online, N1QL will choose at least one such index. That is, if an index scan can be performed, N1QL will not perform a full/primary scan.
- N1QL does not promise to choose the "best" index or combination of indexes to satisfy the query. This is an optimization problem, and no database offers such a guarantee (excluding marketing claims)

Prior to proceeding with secondary scans, the N1QL planner will resolve qualified indexes on the keyspace based on the query predicates. The following algorithm is used to select the indexes for a given query:

- **Online indexes:** Only online indexes are considered. That means when an index is being built (pending), or is only defined but not built, it is disqualified and isn't chosen.
- **Preferred indexes:** If a query has a `USE INDEX` clause, only those indexes are evaluated. If the preferred indexes are not qualified, other online indexes are considered.
- **Satisfying index filters:** For partial/filtered indexes, the N1QL Query Planner considers only those

indexes whose filter (`WHERE`) is broad enough to satisfy the query. The filter does not need to match the query predicate exactly; the filter just needs to be a superset of the query predicate.
- **Satisfying index keys:** Indexes whose leading keys satisfy query predicate are selected. This is the common way to select indexes with B-TREE indexes.
- **Longest satisfying keys:** Finally, among the indexes with satisfying keys, some redundancy is eliminated by keeping the longest satisfying index keys in the index key order. For example: An index with satisfying keys (a, b, c) is retained instead of an index with satisfying keys (a, b). Note that satisfying keys refers only to those keys at the beginning of the index that is used in the query predicate. An index can have additional keys after its satisfying keys.

Once the index selection is done the following scan methods are considered in the order.

1. **IndexCountScan:** Queries with a single projection of COUNT aggregate, NO JOINs, or GROUP BY are considered. The chosen index needs to be covered with a single exact range for the given predicate, and the argument to COUNT needs to be constant or leading key.
2. **Covering secondary scan:** Each satisfied index with the most number of index keys is examined for query coverage, and the shortest covering index will be used. For an index to cover the query, we should be able to run the complete query just using the data in the index. In other words, the index needs to have both keys in the predicate as well as the keys referenced in other clauses, e.g., projection, subquery, order by, etc.
3. **Regular secondary scan:** Indexes with the most number of matching index keys are used. When more than one index is qualified, IntersectScan is used. To avoid IntersectScan, provide a hint with `USE INDEX``.
4. **UNNEST Scan:** Only array indexes with an index key matching the predicates are used for UNNEST scan.
5. **Regular primary scan:** If a primary scan is selected, and no primary index available, the query errors out.

Understanding Index Scans

FILTER, JOIN, and PROJECT are fundamental operations of database query processing. 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 spans in the query plan and passed to Indexer. Spans simply express the predicates in terms of data ranges. Where each range has a start value, an end value, and specifies whether to include the start or the end value.

- A "High" field in the range indicates the end value. If "High" is missing, then there is no upper bound.
- A "Low" field in the range indicates the start value. If "Low" is missing, the scan starts with `MISSING` .
- Inclusion indicates if the values of the High and Low fields are included.

Inclusion #	Meaning	Description
0	NEITHER	Neither High nor Low fields are included
1	LOW	Only Low fields are included

2	HIGH	Only High fields are included
3	BOTH	Both High and Low fields are included

Example: Equality Predicate

```
SELECT meta().id FROM `travel-sample` WHERE id = 10
```

Span Range for	Low	High	Inclusion
ID = 10	10	10	3 (BOTH)

Example: Inclusive One-Sided Range Predicate

```
SELECT meta().id FROM `travel-sample` WHERE id >= 10
```

Span Range for	Low	High	Inclusion
ID >= 10	10	Unbounded	1 (LOW)

Example: Exclusive One-Sided Range Predicate

```
SELECT meta().id FROM `travel-sample` WHERE id > 10
```

Span Range for	Low	High	Inclusion
ID > 10	10	Unbounded	0 (NEITHER)

Example: AND Predicate

```
SELECT meta().id FROM `travel-sample` WHERE id >=10 AND id < 25
```

Span Range for	Low	High	Inclusion
ID >= 10 AND ID < 25	10	25	1 (LOW)

Example: OR Predicate

```
SELECT meta().id FROM `travel-sample` WHERE id = 10 OR id = 20
```

The predicate produces two independent ranges and both of them are pushed to index scan. Duplicate ranges are eliminated, but overlaps are not eliminated.

Span Range for	Low	High	Inclusion
ID = 10	10	10	3 (BOTH)
ID = 20	20	20	3 (BOTH)

When you analyze the explain plan, correlate the predicates in the explain to the spans. Ensure the most optimal index is selected and the spans have the expected range for all the index keys. More keys in each span will make the query more efficient. Further explanation and many more detailed examples of index scans can be found here: https://docs.couchbase.com/server/5.5/performance/index-scans.html

Identifying the Top Slow Queries

The top slow queries can be identified by querying the system catalog using the following select statement (available on version Couchbase 4.5 and above)

```
select * from system:completed_requests
```

The `system:completed_requests` catalog maintains a list of the most recent (4000 by default) completed requests that have run longer than a predefined threshold of time (>=1000ms by default). This information provides a general insight into the health and performance of the query engine and the cluster.

You can configure the `system:completed_requests` catalog by specifying the parameters as command-line options for the cbq-engine.

- `completed-threshold` : Sets the minimum request duration after which requests are added to the `system:completed_requests` catalog. The default value is 1000ms. Specify 0 to log all requests and -1 to not log any requests to the catalog.
  - To specify a different value, use: `cbq-engine -completed-threshold=500`
- `completed-limit` : Sets the number of most recent requests to be tracked in the `system:completed_requests` catalog. The default value is 4000. Specify 0 to not track any requests and -1 to set no limit.
  - To specify a different value, use: `cbq-engine -completed-limit=1000`

You can also set these parameters through the Admin API settings endpoint:

```
curl -X POST \
  -u Administrator:password \
  -d '{ "completed-threshold": 500, "completed-limit": 2000 }' \
  http://localhost:8093/admin/settings
```

system:completed_requests properties

```
[{
  "clientContextID": "MYAPP-23fce132-050b-4ca3-9369-745b579cfad4",
  "elapsedTime": "1.149392493s",
  "errorCount": 0,
  "node": "127.0.0.1:8091",
  "phaseCounts": {
    "fetch": 35,
    "primaryScan": 35,
```

```
    "sort": 2
  },
  "phaseOperators": {
    "authorize": 1,
    "fetch": 4,
    "primaryScan": 4,
    "sort": 1
  },
  "remoteAddr": "127.0.0.1:37149",
  "requestId": "1fd6b7e9-8021-4872-98a8-a07908107674",
  "requestTime": "2019-02-08 00:41:12.722504817 +0000 UTC",
  "resultCount": 2,
  "resultSize": 381,
  "scanConsistency": "unbounded",
  "serviceTime": "1.149185373s",
  "state": "completed",
  "statement": "SELECT type, rendition, score, segment FROM `api` WHERE type
 = 'linearSegment' AND rendition = 'master:c32ae827:162800' ORDER BY score D
ESC LIMIT 1",
  "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_2)",
  "users": "Administrator"
}]
```


Property	Description
clientContextID	The opaque ID or context provided by the client.
elapsedTime	The time taken from when the request was acknowledged by the service to when the request was completed. It includes the time taken by the service to schedule the request.
errorCount	Total number of errors encountered while executing the query.
executionTime	The duration of the query from when it started executing to when it completed.
node	IP address and port of the query engine node in the Couchbase Cluster.
phaseCounts	Count of documents processed at selective phases involved in the query execution, such as authorize, indexscan, fetch, parse, plan, run etc.
phaseOperators	Indicates the number of each kind of query operators involved in different phases of the query processing. For instance, this example, one non covering index path was taken, which involves 1 indxeScan and 1 fetch operators. A join would have probably involved 2 fetches (1 per keyspace) A union select would have twice as many operator counts (1 per each branch of the union). This is in essence the count of all the operators in the `executionTimings` field.
remoteAddr	IP address and port number of the client application, from where the query is received.

requestId	Unique request ID internally generated for the query.
requestTime	Timestamp when the query is received.
resultCount	Total number of documents returned in the query result.
resultSize	Total number of bytes returned in the query result.
scanConsistency	The value of the query setting Scan Consistency used for the query.
serviceTime	Total amount of calendar time taken to complete the query.
state	The state of the query execution, such as completed, in progress, cancelled.
statement	The N1QL query statement being executed.
userAgent	Name of the client application or program that issued the query.
users	Username with whose privileges the query is run.

The `system:completed_requests` catalog can be queried against just like any other keyspace (bucket) in Couchbase.

Longest Running Queries

First target and tune the queries that take the most amount of time.

```
SELECT statement,
    DURATION_TO_STR(avgServiceTime) AS avgServiceTime,
    COUNT(1) AS queries
FROM system:completed_requests
WHERE UPPER(statement) NOT LIKE 'INFER %'
    AND UPPER(statement) NOT LIKE 'CREATE INDEX%'
    AND UPPER(statement) NOT LIKE '% SYSTEM:%'
GROUP BY statement
LETTING avgServiceTime = AVG(STR_TO_DURATION(serviceTime))
ORDER BY avgServiceTime DESC
```

Most Frequent Queries

Secondly, target the queries which occur most frequently.

```
SELECT statement,
    COUNT(1) AS queries
FROM system:completed_requests
WHERE UPPER(statement) NOT LIKE 'INFER %'
    AND UPPER(statement) NOT LIKE 'CREATE INDEX%'
    AND UPPER(statement) NOT LIKE '% SYSTEM:%'
GROUP BY statement
LETTING queries = COUNT(1)
ORDER BY queries DESC
```

Largest Result Size Queries

```
SELECT statement,
    (avgResultSize) AS avgResultSizeBytes,
    (avgResultSize / 1000) AS avgResultSizeKB,
    (avgResultSize / 1000 / 1000) AS avgResultSizeMB,
    COUNT(1) AS queries
FROM system:completed_requests
WHERE UPPER(statement) NOT LIKE 'INFER %'
    AND UPPER(statement) NOT LIKE 'CREATE INDEX%'
    AND UPPER(statement) NOT LIKE '% SYSTEM:%'
GROUP BY statement
LETTING avgResultSize = AVG(resultSize)
ORDER BY avgResultSize DESC
```

Largest Result Count Queries

```
SELECT statement,
    avgResultCount,
    COUNT(1) AS queries
FROM system:completed_requests
WHERE UPPER(statement) NOT LIKE 'INFER %'
    AND UPPER(statement) NOT LIKE 'CREATE INDEX%'
    AND UPPER(statement) NOT LIKE '% SYSTEM:%'
GROUP BY statement
LETTING avgResultCount = AVG(resultCount)
ORDER BY avgResultCount DESC
```

Queries using a Primary Index

```
SELECT *
FROM system:completed_requests
WHERE phaseCounts.`primaryScan` IS NOT MISSING
    AND UPPER(statement) NOT LIKE '% SYSTEM:%'
ORDER BY resultCount DESC
```

Queries that are Not very Selective

```
SELECT statement,
    diff
FROM system:completed_requests
WHERE phaseCounts.`indexScan` > resultCount
LETTING diff = AVG(phaseCounts.`indexScan` - resultCount)
ORDER BY diff DESC
```

Queries Not Using a Covering Index

```
SELECT *
FROM system:completed_requests
WHERE phaseCounts.`indexScan` IS NOT MISSING
    AND phaseCounts.`fetch` IS NOT MISSING
ORDER BY resultCount DESC
```

Understanding an Explain Plan

Plans are built from algebras using a visitor pattern. A separate planner/optimizer is used for index selection. You can view the explain plan but prefacing a query with the `EXPLAIN` keyword, clicking the "Explain" button in query workbench or by executing a query and viewing the "Plan Text". The table below describes the various attributes you see in the explain plan and how to interpret them. Reference the Operators Appendix for a complete list of all operators.

Attributes

Attribute	Description
phaseTimes	Cumulative execution times for various phases involved in the query execution, such as authorize, indexscan, fetch, parse, plan, run etc.
phaseCounts	Count of documents processed at selective phases involved in the query execution, such as authorise, indexscan, fetch, parse, plan, run etc.
phaseOperators	Indicates the number of each kind of query operators involved in different phases of the query processing. For instance, this example, one non-covering index path was taken, which involves 1 indxeScan and 1 fetch operators. A join would have probably involved 2 fetches (1 per keyspace) A union select would have twice as many operator counts (1 per each branch of the union). This is, in essence, the count of all the operators in the `executionTimings` field.
#operator	Name of the operator.
#stats	These values will be dynamic, depending on the documents processed by various phases up to this moment in time.
#itemsIn	Number of input documents to the operator.
#itemsOut	Number of output documents after the operator processing.

#phaseSwitches	Number of switches between executing, waiting for services, or waiting for the goroutine scheduler.
	• execTime - Time spent executing the operator code inside N1QL query engine.
	• kernTime - Time spent waiting to be scheduled for CPU time.
	• servTime - Time spent waiting for another service, such as index or data.
	  ◦ For index scan, it is time spent waiting for GSI/indexer
	  ◦ For fetch, it is time spent waiting on the KV store

These statistics (`kernTime` , `servTime` , and `execTime`) can be very helpful in troubleshooting query performance issues, for example as:

- A high `servTime` for a low number of items processed is an indication that the indexer or KV store is stressed.
- A high `kernTime` means there is a downstream issue in the query plan or the query server having many requests to process (so the scheduled waiting time will be more for CPU time).

When tuning (or writing) a N1QL statement the goal is to drive from the query that has the most selective filter. This means that there are fewer documents/keys are passed to the next step. If the next step is a join, then this means that fewer documents are joined. Check to see whether the access paths are optimal.

When examining the optimizer execution plan, look for the following:

- The driving query/subquery has the best filter.
- The join order in each step returns the fewest number of rows to the next step (that is, the join order should reflect, where possible, going to the best not-yet-used filters).
- Consider the predicates in the N1QL statement and the number of documents being returned.
- Ensure that the right indexes are being used. Telltale signs of poor performance are that the index you wouldn't expect is being used e.g. Primary Index or if the index can be covering is not being used. In general, a smaller more restrictive index will more performant than a primary index.
- Determine why an index is not used for selective predicates
- Ensure that the SPANS are correctly being leveraged for an index. Even if the index is being used by the optimizer if you are not able to push out appropriate SPANS to the optimizer the query performance will suffer

---

Understanding Cardinality and Selectivity

Cardinality and selectivity play a crucial role in index tuning and optimization as they can provide measurable insights into your data set and effectiveness of the index, pointing you to where specific optimizations can be made.

Cardinality refers to the individual uniqueness of values in a specific index key. Each index key (document property) emitted into the index will have varying degrees of cardinality. Cardinality can be broken down into roughly 3 different types:

- *High-Cardinality*: Refers to values that are unique or very uncommon within the index key. Examples include fields such as GUIDs, IDs, email addresses, and usernames.
- *Normal-Cardinality*: Refers to values that are somewhat uncommon but not necessarily unique within the index key. Examples include: first / middle / last name, zip codes. There are last names / surnames that very well may be unique in the data set, however, if you were to examine all of the distinct values you'll find groupings of certain values (i.e. Jones).
- *Low-Cardinality*: Refers to values that are common within the data set and have very few possible values. Examples include status, gender, and booleans. Fields that have little uniqueness and are common across the index, examples are status, gender, and booleans.

Selectivity is the measure of variation in unique values in a given data set and it is represented as a number between `0 - 1` or `0 - 100%` . The formula to calculate selectivity can be represented as follows:

```
selectivity = cardinality/(number of records) * 100
```

or more simply stated:

```
# of Distinct Values / Total # of Records = Selectivity
```

Cardinality and Selectivity can be applied to any "data set" such as an index, query or bucket. In general for database indexes, the higher cardinality -> better selectivity -> faster scans -> increased performance. Consider the table below:

	Name	Breed	Gender	Origin Country
1	Oakley	German Shepherd	M	Germany
2	Zeus	Doberman Pinscher	M	Germany
3	Darby	Doberman Pinscher	F	Germany
4	Rocky	Bulldog	M	United Kingdom
5	Lucy	Labrador Retriever	F	Canada
6	Buddy	Golden Retriever	M	United Kingdom
7	Molly	Pug	F	China
8	Sadie	Labrador Retriever	F	Canada
9	Max	Boxer	M	Germany
10	Simba	Great Dane	M	Germany
Cardinality	10	6	2	4
Selectivity	100%	60%	20%	40%

Examples

Using the `travel-sample` bucket, we'll calculate two selectivity values for some of the sample indexes:

- *Projection Selectivity*: This is a measure of the # of documents in the bucket that match the index filter/ `WHERE` predicate and contain the leading field. This is often referred to as "index segmentation".
- *Index Selectivity*: This is a measure of the number of unique values in the index compared to the total # of entries in the index.

> For optimum performance, you will want a relatively low percentage of Projection Selectivity as this means the index is smaller, and a higher value for Index selectivity as this means there is a lot of uniqueness within the index.

Initially, we need to get the total # of documents in the bucket, as we will reuse this value in all of our calculations:

```
SELECT RAW COUNT(1)
FROM `travel-sample`
```

```
[
  31591
]
```

Example 1: `def_type` index

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

Determine the total number of records in the index, this query will push the `COUNT()` down to the indexer, and we trigger the use of the index by referencing the first field in the index. If needed you could optionally specify a `USE INDEX()` statement to ensure the index is used:

```
SELECT COUNT(1)
FROM `travel-sample`
WHERE type IS NOT MISSING
```

```
[
  31591
]
```

Next, we need to determine the total number of possible unique values in the index:

```
SELECT RAW COUNT(DISTINCT type)
FROM `travel-sample`
WHERE type IS NOT MISSING
```

```
[
  5
]
```

Description	Formula	Selectivity
Projection Selectivity	(31591 / 31591) * 100	100%
Index Selectivity	(5 / 31591) * 100	0.015%

Example 2: `def_faa` index

```
CREATE INDEX `def_faa` ON `travel-sample`(`faa`)
```

Determine the total number of records in the index:

```
SELECT RAW COUNT(1)
FROM `travel-sample`
WHERE faa IS NOT MISSING
```

```
[
  1968
]
```

Next, we need to determine the total number of possible unique values in the index:

```
SELECT RAW COUNT(DISTINCT faa)
FROM `travel-sample`
WHERE faa IS NOT MISSING
```

```
[
  1708
]
```

Description	Formula	Selectivity
Projection Selectivity	(1968 / 31591) * 100	6.23%
Index Selectivity	(1708 / 1968) * 100	86.79%

Example 3: `def_country` index

```
CREATE INDEX `def_country` ON `travel-sample`(`country`, `type`)
```

Determine the total number of records in the index:

```
SELECT RAW COUNT(1)
FROM `travel-sample`
WHERE country IS NOT MISSING
```

```
[
  7567
]
```

Next, we need to determine the total number of possible unique values in the index. For this example, however, there are two index keys `country` and `type`. The selectivity depends on how the index will be used and when optimizing it is important to understand how the cardinality of one key can affect the other.

Example 3.a

```
SELECT *
FROM `travel-sample`
WHERE country = 'United States'
```

```
SELECT RAW COUNT(DISTINCT country)
FROM `travel-sample`
WHERE country IS NOT MISSING
```

```
[
  3
]
```

Example 3.b

```
SELECT *
FROM `travel-sample`
WHERE country = 'United States' AND type = 'landmark'
```

When both keys are used, the selectivity can be described in two ways, the first is the total uniqueness of both keys when combined together:

```
SELECT RAW COUNT(DISTINCT country || type)
FROM `travel-sample`
WHERE country IS NOT MISSING
```

```
[
  12
]
```

Example 3.c

The second is # of unique keys for the second index key which matches the previous index key:

```
SELECT RAW COUNT(1)
FROM `travel-sample`
WHERE country = 'United States'
```

```
[
  3948
]
```

```
SELECT RAW COUNT(DISTINCT type)
FROM `travel-sample`
WHERE country = 'United States'
```

```
[
  4
]
```

Description	Formula	Selectivity
Projection Selectivity	(7567 / 31591) * 100	23.95%
Index Selectivity (3.a)	(3 / 7567) * 100	0.039%
Index Selectivity (3.b)	(12 / 7567) * 100	0.16%
Index Selectivity (3.c)	(4 / 3948) * 100	0.10%

Summary

Both cardinality and selectivity can affect the performance of IndexScans, and you should always consider their implications as it relates to your access patterns and query predicates. Having a solid understanding of cardinality and selectivity as it relates to your data set can provide solid guidance in the tuning and determining the order of index keys within the index.

Understanding Covering Indexes and TTLs

When using covering indexes, there are some important considerations in-terms of how your N1QL queries are constructed to ensure they do NOT return stale data from your indexes(GSIs). To better understand this concept, it's important to have a basic understanding of document expiration and how it works.

Example Use Case

To prove the importance of this concept, let's consider a 3-Legged OAuth grant flow scenario that uses a covering index and also has documents with TTLs set to 10 minutes. After then 10 minute expiry, the document(s) with this TTL will expire and no longer be available for use.

Example model using a Document Key of: `temp:code:7zk5ZDczMzRlNDEwYLj`

```
{
    "scopes": [
        "account.read",
```

```
        "account.update",
        "groups.read"
    ],
    "expiry": 1571668070320,
    "userID": "34200980012",
    "docType": "tempCode",
    "email": "user@yourdomain.com",
    "roledID": "1"
}
```

Example Index

```
CREATE INDEX idx_temp_code ON bucket_name(
  email, userid, scopes, expiry, META().expiration
)
WHERE docType = "tempCode"
```

The example query below returns documents after the bucket TTL has expired and yields stale data.

Example N1QL Query with Unexpected Results

```
SELECT meta().expiration, email, scopes, userid
FROM bucket_name
WHERE docType="tempCode" AND email="user@yourdomain.com"
```

Why does this happen?

When a document's expiration is reached(i.e. TTL expires), it is deleted when one of the following occurs:

- expiry pager runs(default every 60 minutes)
- compaction runs(default 30% fragmentation)
- attempt is made to access the document(this only applies to KV operations)

The issue in the case of covering indexes, is that N1QL does not currently use the underlying capabilities of the Subdoc API when a query is executed, so the metadata associated with the document is not taken into consideration during the phases of query execution. Therefore, we have to ensure the queries encapsulate the appropriate logic and provide the expected results.

The solution is simple and to obtain accurate results, all we need to do is modify our queries to have an additional condition. So, to solve for this situation, we simply add a condition in the WHERE clause to reference the `meta().expiration` and make sure it is greater than the current time(i.e. `NOW_MILLIS()`).

Example N1QL Query with Expected Results

```
SELECT meta().expiration,email,scopes,userid
FROM bucket_name
WHERE docType="tempCode" AND email="user@yourdomain.com"
```

```
  AND TOSTRING(META().expiration) > SPLIT(TOSTRING(NOW_MILLIS() / 1000), "."
)[0]
```

Tuning Tips and Advice

Tip 1: Use USE KEYS

If you know the Document Id/Key of the document, you should leverage the USE KEYS clause. This bypasses the Index Service (hence bypassing network, scans, index results and processing) - it's the closest thing you have on the N1QL side comparable to a KV fetch. The optimizer will use a KeyScan when you use the USE KEYS instead of an Index or Primary Scan. When the key is known and returning the entire document is required, always preference `USE KEYS` over a `META().id` index scan.

```
SELECT * FROM `travel-sample` USE KEYS ["landmark_37588"];
```

You can specify multiple values in `USE KEYS` if you are querying for multiple documents.

```
SELECT * FROM `travel-sample` USE KEYS ["landmark_37588", "landmark_37603" ]
;
```

JOIN operations are also done using the document keys.

```
SELECT * FROM ORDERS o INNER JOIN CUSTOMER c ON KEYS o.id;
SELECT * FROM ORDERS o USE KEYS ["ord:382"] INNER JOIN CUSTOMER c ON KEYS o.
id;
```

As this query is performing a KV GET() via N1QL, it would be even more performant to bypass N1QL altogether and issue a KV GET() directly via the SDK. `bucket.get("landmark_37588")`

> Note that when selecting specific field(s), a covered index scan may be faster than performing the data service fetch when working with large documents or a long list of keys.

Tip 2: Do not Index Values that are an EQUALITY predicate of the Index

Remove any index keys/expressions that are listed in both the index and the indexes `WHERE` statement as an equality predicate. These values would have zero cardinality and do not need to be indexed as they would result in slower IndexScans, they simply need to prevent documents who do not satisfy the condition as true from being added to the index. The query/index services are intelligent enough to understand a query and automatically cover values that are present as an equality predicate in the index.

Consider the following query:

```
SELECT userId, firstName, lastName
FROM ecommerce
WHERE docType = "user" AND username = "johnsmith21"
```

Now consider the following indexes, all of which will satisfy the query above:

```
CREATE INDEX idx_usernames ON ecommerce(docType, username)
```

This index emits both `docType` and `username` into the index. Not only is `username` more unique, but you would expect that this index only contained just "user" documents. However, this index would contain an entry for every single document where the document had a `docType` property, regardless of whether or not it has a `username` property as only the leading key needs to qualify. Think of the leading key having a `WHERE docType IS NOT MISSING` statement.

```
CREATE INDEX idx_usernames ON ecommerce(username, docType)
```

This index would successfully limit the indexes scope to just documents that contained a `username` property, but again this is not guaranteed. we could filter the index by adding a `WHERE docType = "user"` predicate:

```
CREATE INDEX idx_usernames ON ecommerce(username, docType)
WHERE docType = "user"
```

However, by having the `docType` emitted into the index when it will only be a single value, it is just wasted bytes.

```
CREATE INDEX idx_usernames ON ecommerce(username)
WHERE docType = "user"
```

This is the best choice of all of the indexes above, as it will filter out any documents that do not satisfy `WHERE docType = "user"` and only index the remaining documents that contain a `username` property.

Tip 3: Every Index should be Filtered i.e contain a WHERE clause

Often times referred to as "Partial Index", filtered indexes are an index on a subset of documents in the keyspace which are relevant to the query being executed. The result is a smaller index, which results in faster scans, yielding faster response times.

```
CREATE INDEX idx_cx3 ON `travel-sample` (state, city, name.lastname)
WHERE type = 'hotel'

CREATE INDEX idx_cx4 ON customer (state, city, name.lastname)
WHERE type = 'hotel' and country  = 'United States' AND ratings > 2
```

Tip 4: Index Key Order and Predicate Types

The order of index keys, as well as the cardinality (uniqueness) of the values for a specific key/expression, can have a dramatic affect on query performance. Index keys should be first ordered based on the queries predicate types in the following order:

1. EQUALITY
2. IN
3. LESS THAN
4. BETWEEN
5. GREATER THAN
6. Array predicates
7. Look to add additional fields for the index to cover the query

For keys who share the same predicate type, cardinality comes into play. As a general rule of thumb, order keys from left to right based on highest cardinality (most unique) to lowest cardinality (least unique) when they share the same predicate type. Note that the query predicates do not need to be listed in the order in which they are listed in the index, the query planner determines this automatically.

Consider the following query and index:

```
SELECT cid, address
FROM customer
WHERE type = 'premium'
  AND state = 'CA'
  AND zipcode IN [29482, 29284, 29482, 28472]
  AND salary < 50000
  AND age > 45
```

```
CREATE INDEX idx_orders ON customer(state, zipcode, salary, age, address, ci
d)
WHERE type = 'premium'
```

Even though `zipcode` is more unique than `state`, it is being queried using an `IN` statement which is the equivalent of an `OR`. It is more efficient to reduce the index entries first by `state` then by `zipcode`, however if our primary access pattern was `zipcode = 29482`, then we would want to list it first.

Tip 5: Index to Avoid Sorting

Each index stores data pre-sorted by the index keys, matching the keys in the ORDER BY and leading N keys will avoid sorting. When exploiting the index order, the index keys should be added after all predicates and before any additional values that may be used to cover the query.

Take a scenario where you want to retrieve the order history for a given user, consider the following index and query:

```
CREATE INDEX `idx_order_history` ON `ecommerce` (
  userId, orderDate, orderTotal, orderId
)
WHERE docType = "order"
```

```
SELECT orderId, orderDate, orderTotal
FROM ecommerce
WHERE docType = "order" AND userId = 123
ORDER BY orderDate DESC
```

If you examine the `EXPLAIN` plan for the query near the bottom you will see:

```
{
    "#operator": "Order",
    "sort_terms": [{
        "desc": true,
        "expr": "cover ((`customer`.`orderDate`))"
    }]
}
```

The Order operator takes all of the documents from the previous operator (IndexScan) and must loop over all of the records and sort them in-memory based on the `ORDER BY` statement. Data is pre-sorted `ASC` by default, with the previous index the `ASC` is implied but would look like:

```
CREATE INDEX `idx_order_history2` ON `customer` (
  userId ASC, orderDate ASC, orderTotal ASC, orderId ASC
)
WHERE docType = "order"
```

Knowing that we want our result ordered by the `orderDate DESC`, we can inform the indexer to store the data in the order in which we will use it:

```
CREATE INDEX `idx_order_history_sorted` ON `ecommerce` (
  userId, orderDate DESC, orderTotal, orderId
)
WHERE docType = "order"
```

Now if we issue an `EXPLAIN` on the query, you will see the "Order" operator is missing as it is not needed since the result is already in the appropriate order.

Tip 6: Use Covering Indexes

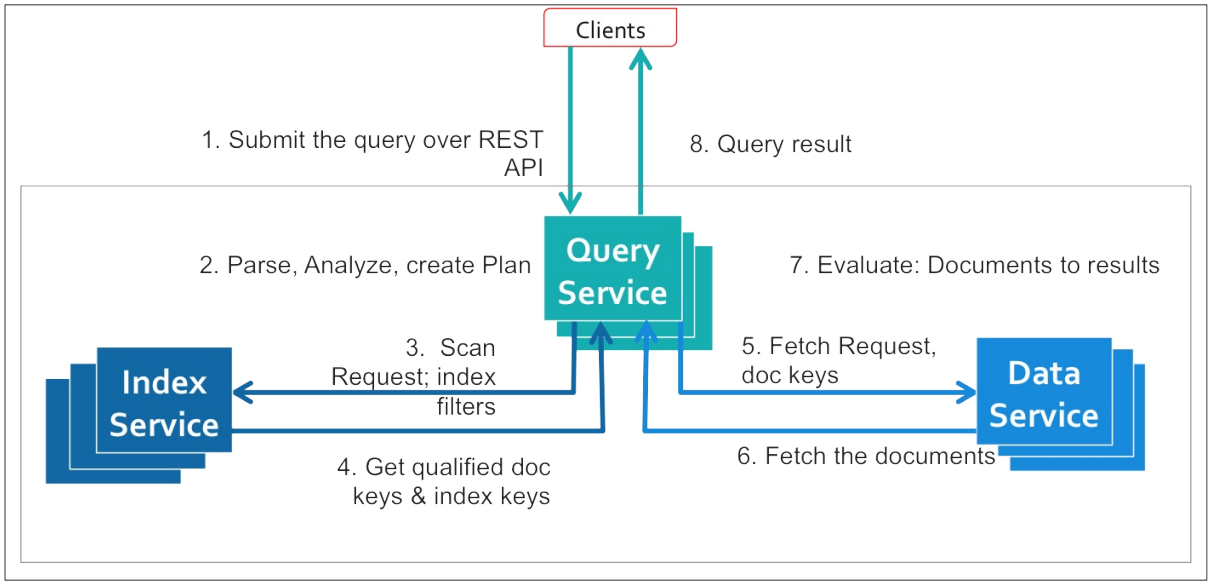
Covering Indexes are indexes which contain all of the query predicates (`WHERE`), all of the returning values (`SELECT ...`) and any other processed attributes. When the index contains all of these values, it "covers" the query and the Query service does not need to go-to the Data service to obtain values for those fields. Covering indexes make queries efficient since it bypasses the "FETCH" from the Data service saving a significant amount of data transfer and processing. Both the Final Project and Filtering can be "covered" if an index is created appropriately.

Consider the following index:

```
CREATE INDEX idx_cx3 ON customer(state, city, name.lastname)
WHERE status = 'premium'
```

The following diagram illustrates the query execution workflow for a query that is not "covered":

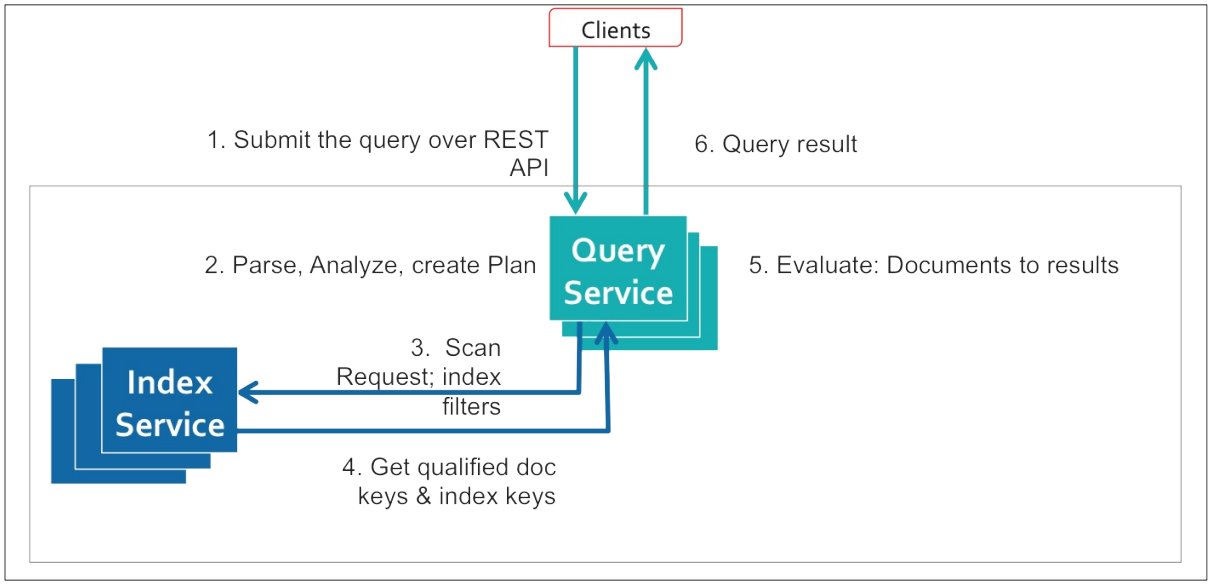
```
SELECT *
FROM customer
WHERE state = 'CA' AND status = 'premium'
```



When performing a `SELECT *` , a data service fetch will always be required, and the query cannot be covered.

The following diagram illustrates the query execution workflow where the query is covered:

```
SELECT status, state, city
FROM customer
WHERE state = 'CA' AND status = 'premium'
```



As you can see in the second diagram, a well-designed query that uses a covering index avoids the additional steps to fetch the data from the data service. This results in a considerable performance improvement.

We can verify that a query is "covered" by reviewing the explain plan:

```
{
    ...
    "~children": [{
            "#operator": "IndexScan3",
            "covers": [
                "cover ((`customer`.`state`))",
                "cover ((`customer`.`city`))",
                "cover (((`customer`.`name`).`lastname`))",
                "cover ((meta(`customer`).`id`))"
            ],
            "filter_covers": {
                "cover ((`customer`.`status`))": "premium"
            },
            ...
        }
    ]
}
```

It is important to note that a "FETCH" operation is not necessarily a bad thing, and every query does not always need to be "covered". Covering indexes offer the performance benefit of having all of the data in a single place and avoid a scatter-gather (i.e. fetches), but a covering index could potentially be fairly large and should be properly sized. A query which performs an IndexScan and returns a few records then performs a fetch, might be perfectly acceptable and meet SLAs. However, a query that returns tens or hundreds of thousands of rows and has to perform a fetch, could have a dramatic effect on system performance.

Tip 7: Never Create a Primary Index in Production

Unlike relational databases, Couchbase and N1QL do not require a primary index as long as the query has a viable secondary index to access the data.

A primary index scan is analogous to a full table scan in relational database systems. However, where a table scan stops at the "table", a primary index scan will perform the equivalent of a full database scan. N1QL will retrieve every document ID in the entire keyspace from the primary index, then fetch each document in the bucket and finally performing any predicate-join-project processing.

When a primary index is present, even though it may never be "intended" to be used, it acts as a fallback for any query who does not have a satisfying index at execution time. Think failure scenarios, if the queries qualifying index(es) go offline for whatever reason (i.e. node failure) and a primary index exists, it will be used, which could cause unintended and unexpected side-effects.

Tip 8: Avoid "docType" Only Index

A keyspace (bucket) in Couchbase is a logical namespace for one or more types of documents. Standard practice in Couchbase is to ensure that each document has a common `docType`, `type`, `_class`, etc. property that identifies the model/purpose of the document. This property is not only a good practice it allows for efficient indexing (i.e. Partial Index).

While filtering on the `docType` is strongly encouraged as it creates a partial index on a subset of documents, creating an index on the `docType` as a leading or stand-alone index key is inefficient and has extremely low cardinality.

```
CREATE INDEX `idx_docType` ON `bucket` (`docType`)
```

Indexes like this can inadvertently cause slow-performing queries and unexpected explain plans. For example, it increases the likelihood of an IntersectScan. More importantly though, if you've established the use of a `docType` property on each of your documents, more than likely developers writing N1QL statements will add that as a predicate to each of their queries, i.e.:

```
SELECT *
FROM bucket
WHERE docType = "user" AND username = "johnsmith32"
```

Just as a primary index can be inadvertently used as a fallback when expected indexes are not found and cause potentially a high number of "FETCH" operations, a `docType` index can do exactly the same thing and this is why it should be avoided.

Tip 9: Partition Indexes

When an index is created, the entire index only exists on a single node (replicas excluded). If the index grows in size and can no longer fit into memory in the case of MOI indexes, or the resident ratio drops low in the case of Standard GSI, you have to either add more resources to the node or you can partition the index.

The process of index partitioning involves distributing a single large index across multiple index nodes in the cluster.

Prior to Couchbase Server 5.5 this was achieved by creating multiple smaller range partitioned or partial indexes. The following query is our example:

```
SELECT userId, firstName, lastName
FROM ecommerce
WHERE docType = 'user' AND LOWER(username) = 'johnsmith32'
```

The following is the initial index that is used to cover the query:

```
CREATE INDEX idx_users ON ecommerce(LOWER(username), userId, firstName, last
Name)
WHERE docType = 'user'
```

Over time the index grows in size and might not be performing as expected, can no longer fit on a single node, etc. and needs to be partitioned across multiple nodes:

```
CREATE INDEX idx_users_AtoM ON ecommerce(LOWER(username), userId, firstName,
 lastName)
WHERE docType = 'user' AND LOWER(username) >= 'a' AND LOWER(username) < 'n'
WITH { "nodes": ["index_host1"]}

CREATE INDEX idx_users_NtoZ ON ecommerce(LOWER(username), userId, firstName,
 lastName)
WHERE docType = 'user' AND LOWER(username) >= 'n' AND LOWER(username) < '['
WITH { "nodes": ["index_host2"]}
```

The same query above works without any changes. Indexes can be partitioned in an infinite number of ways. As of Couchbase Server 5.5+ you can define a single index, the partitioning strategy/keys as well as the number of partitions (16 by default) to create and we'll manage the distribution across the cluster automatically for you. Index Partitioning can be implemented by specifying the [PARTITION BY] clause.

```
CREATE INDEX idx_users ON ecommerce(LOWER(username), userId, firstName, last
Name)
WHERE docType = 'user'
PARTITION BY HASH(LOWER(username))
WITH { "num_partitions": 10 }
```

Index partitioning has the benefit of "Partition Elimination", where the query planner understands that `WHERE ... LOWER(username) = 'johnsmith32'`, is an equality and it knows which of the 10 partitions that indexed value lives on, and will scan just that single partition. Be sure to evaluate the key/expressions that you're using for partitioning as it can have an impact on the scans.

Tip 10: Avoid the use of SELECT *

Many developers or frameworks will use `SELECT * FROM ...` as much simpler and less work than typing out individual property `SELECT orderId, orderDate, orderTotal FROM ...`. However, in general, this is not a wise thing to do regardless of the database being used, and in some situations, it can have serious performance implications.

Using `SELECT *` returns all of the properties for each document. Oftentimes a query does not actually need all of the properties, only a select few. This causes unnecessary I/O, resulting in a larger payload size that must be transferred across the wire back to the application, where a smaller payload is more performant.

The most important reason to not use `SELECT *` is you are limiting what the query optimizer can do to pick more appropriate indexes (i.e. a covering index). When `SELECT *` is used, no matter what, 100% of the time a "FETCH" operation will be performed and it is impossible to further optimize the query without changing the underlying code / N1QL statement.

Consider the following scenario, where you want to retrieve all of the airlines for a given country and use the `id`, `name` and `callsign` properties. Initially, our index might look like:

```
CREATE INDEX `idx_airline_country` ON `travel-sample` (country)
WHERE type = 'airline'
```

Now compare the following queries:

```
SELECT *
FROM `travel-sample`
WHERE type = 'airline' AND country = 'United States'
```

```
SELECT id, name, callsign
FROM `travel-sample`
WHERE type = 'airline' AND country = 'United States'
```

Both queries ultimately achieve the result and based on our initial index both queries would perform a "FETCH" operation, because of `*` in query one, and because query two contains properties that are not in the index. Hypothetically let's say that this query is performed 10 times per second, the index may be perfectly fine and meet SLAs.

You have appropriate monitoring and profiling in place. Over time our application becomes more and more popular, and our query is now being performed 1,000 times per second. Originally, both queries returned 127 records, and a "FETCH" was being performed where `127 docs * 10qps = 1,270 get ops/sec`, but is now `127 docs * 1000qps = 127,000 get ops/sec`. This is causing system performance issues and you realize that improvements need to be made and the query needs a covering index, the profile of the second query tells you exactly what fields need to be covered and you create the following index in production:

```
CREATE INDEX `idx_airline_country_cvr` ON `travel-sample` (country, id, name
, callsign)
```

```
WHERE type = 'airline'
```

If you were using the second query, the number of gets/second would go from `127,000` to `0` and you were able to achieve this without a single code change or deployment to the application. However, if you were using the first query this index would have not offered any performance benefit, the application code would have to be updated and then deployed to see the performance gains.

Tip 11: Avoid the use of USE INDEX

When specified the USE INDEX clause *hints* to the query optimizer that the index(es) listed should be preferred. The query optimizer is a rule-based optimizer, and will generally pick the most appropriate index to satisfy a query. Additionally, with every release of Couchbase, there are improvements made to the optimizer, making it more efficient.

If a `USE INDEX(...)` is specified it couples code and operations. Meaning the index that is specified, cannot be dropped without a code change, as the code is expecting the index to be there and if it is not this would result in an error. Moreover, you cannot optimize the index that is referenced without dropping it first which would result in a period of downtime while the index is being built and the creation of a more optimized index wouldn't yield any benefit either as it would not be preferred.

The use of the `USE INDEX()` statement can be beneficial with the elimination of IntersectScans but should be done with caution for the reasons listed above.

> The `USE INDEX(...)` clause can accept a comma-delimited list of indexes.

Tip 12: Pushdown Pagination to the Index

Optimizing pagination queries is usually the most critical part of tuning. Exploiting index ordering is even more beneficial when paginating. Both OFFSET and LIMIT are attempted to be pushed down to the indexer whenever possible, this depends on a few different factors:

- If the whole predicate (`WHERE`) can be pushed down to a single index (i.e. all index keys exist)
- An IntersectScan is not being performed
- If an `ORDER BY` clause is used, the index keys must be in the same order
- There is no `JOIN` clause

If all of these are *true* then the `LIMIT` and `OFFSET` are pushed to the indexer, otherwise they are applied by the query service after all IndexScans are performed. Take the following index for example:

```
CREATE INDEX idx_products ON ecommerce (
  productCategory, productName, productPrice, productId
)
WHERE docType = 'product'
```

This query retrieves all of the products sorted by the `productName` property and exploits the index order so `LIMIT` and `OFFSET` are pushed down to the indexer.

```
SELECT productId, productName, productPrice
FROM ecommerce
WHERE docType = 'product' AND productCategory = "Electronics"
ORDER BY productName ASC
LIMIT 100
OFFSET 300
```

This can be verified by examining the `EXPLAIN` plan of the query, in the `IndexScan3` #operator will show both of the "limit" and "offset" properties, indicating that they were pushed down to the indexer.

```
{
    "#operator": "IndexScan3",
  ...
    "index": "idx_products",
    "limit": "100",
    "offset": "300",
  ...
}
```

Now if we adjust the query to sort by `productPrice DESC` .

```
SELECT productId, productName, productPrice
FROM ecommerce
WHERE docType = 'product' AND productCategory = "Electronics"
ORDER BY productPrice DESC
LIMIT 100
OFFSET 300
```

The query is still covered, as before but now the "Limit" and "Offset" are pushed to the bottom of the Sequence and are the last thing to happen before the final projection:

```
[
    ...
    {
        "#operator": "Order",
        "limit": "100",
        "offset": "300",
        "sort_terms": [{
            "desc": true,
            "expr": "cover ((`customer`.`productPrice`))"
        }]
    }, {
        "#operator": "Offset",
        "expr": "300"
```

```
    }, {
        "#operator": "Limit",
        "expr": "100"
    }
]
```

`LIMIT` and `OFFSET` are generally the go to for database pagination and Couchbase Server has many optimizations that make these operations really fast. There is one drawback to limit/offset pagination in any database and that is the greater the offset, the longer the initial index scan has to traverse the index before the limit can be implied. Oftentimes this is negligible, however, if you want to squeeze every last bit of performance out of pagination there is another approach called KeySet Pagination.

Tip 13: Use Query Bindings

It is the responsibility of the application to sanitize and inspect dynamic/input data before sending it to the database. If an application uses this input to dynamically construct a query, it is opening the database to SQL injection attacks.

```
def airports_in_city(city):
  query_string = "SELECT airportname FROM `travel-sample` WHERE city="
  query_string += '"' + city + '"'
  return cb.n1ql_query(query_string)
```

This is insecure as any value or N1QL statement could be passed as `city`. N1QL allows the use of placeholders to declare dynamic query parameters. Query parameters (named or positional) allow your application to securely use dynamic query arguments for your application.

Implement named or positional parameters for all dynamic query arguments.

```
def airports_in_city(city):
  query_string = "SELECT airportname FROM `travel-sample` WHERE city=$1"
  query = N1QLQuery(query_string, city)
  return cb.n1ql_query(query)
```

Not only is this more secure, but it also simplifies query profiling and tuning, while the same statement is issued with different parameters, it can be profiled as the same.

Tip 14: Combine Indexes with Shared/Common Index Keys

It can be an easy habit to get into of optimizing every query and have a 1:1 ratio for the query to index. This is not necessary and can be avoided by expecting common leading keys of various indexes and combining multiple indexes into a single index that can service multiple queries. Take the following queries an example:

```
SELECT orderId, orderDate, orderTotal
FROM ecommerce
WHERE docType = 'orders'
```

```
  AND billing.country = 'US'
```

```
CREATE INDEX idx_orders_country ON ecommerce (billing.country)
WHERE docType = 'order'
```

```
SELECT orderId, orderDate, orderTotal
FROM ecommerce
WHERE docType = 'orders'
  AND billing.country = 'US'
  AND billing.state = 'CA'
```

```
CREATE INDEX idx_orders_state_country ON ecommerce (
  billing.state, billing.country
)
WHERE docType = 'order'
```

```
SELECT orderId, orderDate, orderTotal
FROM ecommerce
WHERE docType = 'orders'
  AND billing.country = 'US'
  AND billing.state = 'CA'
  AND orderTotal >= 1000
```

```
CREATE INDEX idx_orders_country ON ecommerce (
  billing.state, billing.country, orderTotal
)
WHERE docType = 'order'
```

All of these indexes can be combined into a single index. It should be pointed out in this example, `billing.country` is the first index key, as the query that uses it expects `billing.country` to be the leading key of the index, this may or may not have an effect on SLAs and should be tested as `billing.state` has a higher cardinality than `billing.country` but may be negligible.

```
CREATE INDEX idx_orders_country ON ecommerce (
  billing.country, billing.state, orderTotal
)
WHERE docType = 'order'
```

Tip 15: Use Prepared Statements

When a N1QL statement is sent to the server, the Query service will inspect and parse the string, determining which indexes to query, ultimately defining a Query Plan to optimally satisfy the statement. The computation for the plan adds some additional processing time and overhead for the query.

Often-used queries can be prepared so that the computed plan is generated only once. Subsequent queries using the same query string will use the pre-generated plan instead, saving on the overhead and processing of the plan each time. Parameterized queries are considered the same query for caching and planning purposes, even if the supplied parameters are different.

There are two approaches to implementing prepared statements, choose one that best fits your environment.

SDK Prepared Statements

This method of prepared statements sets the `adhoc = false` option for a given query. The SDK will internally prepare the statement and store the plan in an internal cache specific to that SDK instance. After the statement has been initially prepared the first time, subsequent calls to the same statement will pass the prepared plan to the Query service, eliminating the inspection, parsing, and planning steps and start executing immediately.

```
query = N1QLQuery("SELECT airportname FROM `travel-sample` WHERE country=$1"
, "USA")
q.adhoc = False
```

Named Prepared statements

This method of prepared statements is similar to the previous option but instead of the SDK managing the planning of the queries, it would be managed by the application or through an external process.

First, the query has to be prepared by using a PREPARE Statement

```
PREPARE unique_name_for_query FROM
SELECT airportname FROM `travel-sample` WHERE country=$1
```

Once the query has been prepared, the query plan is cached in the Query service and can be executed by using an EXECUTE Statement.

```
query = N1QLQuery("EXECUTE unique_name_for_query", "USA")
```

The tradeoff is there is a single cached plan that can be used by many clients, however, the lifecycle of the plan must be maintained. For example, if the plan doesn't exist an error will be thrown, that would need to be trapped and then re-prepare the query and execute again.

Tip 16: Avoid IntersectScans

An IntersectScan is when two or more indexes are used to satisfy a query. This results in two or more separate IndexScan operations, each returning back qualifying results (i.e. `meta().id`) and then intersecting the scans together only returning results that are present in both. Consider the following query and indexes:

```
SELECT *
FROM `travel-sample`
WHERE type = "landmark" AND activity = "drink" AND country = "France"
```

```
CREATE INDEX `idx_landmark_activity` ON `travel-sample` (activity)
WHERE type = "landmark"

CREATE INDEX `idx_landmark_country` ON `travel-sample` (country)
WHERE type = "landmark"
```

If you execute this query it will run in ~100ms, examining the plan text shows that an IntersectScan is performed, and an IndexScan is run on each of the above indexes. The scan on `idx_landmark_country` returns 388 results and the scan on `idx_landmark_activity` returns 287 results, both of these are passed to the IntersectScan operator for a total of 675 records and then filters that down to 388 results.

```
{
    "~children": [{
        "#operator": "IntersectScan",
        "#stats": {
            "#itemsIn": 675,
            "#itemsOut": 388,
      ...
        },
        "scans": [{
                "#operator": "IndexScan3",
                "#stats": {
                    "#itemsOut": 388,
          ...
                },
                "index": "idx_landmark_country",
            },
            {
                "#operator": "IndexScan3",
                "#stats": {
                    "#itemsOut": 287,
                },
                "index": "idx_landmark_activity",
            }
        ]
    }]
}
```

Alternatively, if a composite index is used instead, this will result in a single IndexScan operation and be more performant. The same query using the index below will execute in ~17ms.

```
CREATE INDEX `idx_landmark_activity_country` ON `travel-sample` (activity, c
ountry)
WHERE type = "landmark"
```

In general, a single wide index (composite index) which meets the criteria will be more performant than intersections on multiple singular indexes. Only consider intersection when the predicate usage is non-deterministic.

Tip 17: Avoid LIKE Statements

Oftentimes we need to find partial matches within a given index key and use a `LIKE` statement. This is useful and convenient syntax, however, this performs a range scan that depending on the use of `%` can be a range of the entire index and result in a lot of extra processing. For this example, we'll use the following index and base query:

```
CREATE INDEX idx_landmark_names ON `travel-sample` (name)
WHERE type = "landmark"
```

```
SELECT name
FROM `travel-sample`
WHERE type = "landmark"
  AND name LIKE '%Theater%'
```

Execution Time: ~190ms

Now let's examine the various spans associated with the `LIKE` statement:

Statement	Low	High	Inclusion
'%Theater%'	""	"[]"	1
'Theater%'	"Theater"	"Theates"	1
'%Theater'	""	"[]"	1

Clearly, the second option `'Theater%'` offers the more performant range scan as it is a targeted subset of the index key instead of the entire index. Optimizing your `LIKE` queries to only match on the righthand side offers some performance benefit, but this is not always a possibility.

A powerful feature of N1QL is that it can index individual array elements, not just individual scalar properties. While `name` is a simple string, we can use any of the available string functions to convert the string into an array (SPLIT, TOKENS, SUFFIXES, etc.) For this example, we'll use the `SUFFIXES()` function.

```
SELECT RAW SUFFIXES("Clay Theater")
```

```
[
  [
    "Clay Theater",
    "lay Theater",
    "ay Theater",
    "y Theater",
    " Theater",
    "Theater",
    "heater",
    "eater",
    "ater",
    "ter",
    "er",
    "r"
  ]
]
```

As this returns all possible suffixes of a given string as an array, we can effectively index this array and now remove the left-hand `%` from our query providing a more efficient range scan.

```
CREATE INDEX idx_landmark_names_suffixes ON `travel-sample` (
  DISTINCT ARRAY v
    FOR v IN SUFFIXES(LOWER(name))
  END
)
WHERE type = 'landmark'
```

```
SELECT name
FROM `travel-sample`
WHERE type = "landmark"
  AND ANY v IN SUFFIXES(LOWER(name))
    SATISFIES v LIKE 'theater%'
  END
```

Execution Time: ~17ms

This query is now 11 times faster than the original. You should always consider the size of an array index, as in this case the larger the string the larger the array for each item and the larger the index size. Reference

Tip 18: Consider Array Indexes as an alternative to OR Statements

Many times we need to write a query that can satisfy "this OR that" and return results from either the left-hand or right-hand side of the `OR`. Lets take a common example where a user needs to login to an application with their "username" OR "email".

```
SELECT userId, pwd, firstName, lastName
FROM ecommerce
WHERE docType = 'user'
  AND (
    username = 'johns'
    OR
    email = 'johns'
  )
```

And we'll start with the following indexes:

```
CREATE INDEX `idx_username` ON `ecommerce` (username)
WHERE docType = 'user'

CREATE INDEX `idx_email` ON `ecommerce` (email)
WHERE docType = 'user'
```

Viewing the explain plan of this query shows that a `UnionScan` is performed using two separate `IndexScan` operations on the `idx_username` and `idx_email`. A simple approach, in this case, would be to have the application construct two separate queries and inspect the input prior to issuing the query as an email pattern is trivial to validate, which would eliminate the `UnionScan` and result in a single `IndexScan`. This might not always be possible, so next, you might drop the previous indexes and attempt to create a single index to cover both `username` and `email` address:

```
CREATE INDEX `idx_username_email` ON `ecommerce` (username, email)
WHERE docType = 'user'
```

But the query would fail, because a `UnionScan` is still attempted, it is just two scans of the same index. It would need to be rewritten as follows for the same index to use:

```
SELECT userId, pwd, firstName, lastName
FROM ecommerce
WHERE docType = 'user'
  AND (
    username = 'johns'
    OR
    username IS NOT MISSING AND email = 'johns'
  )
```

This is not an optimum approach either. While we only maintain a single index it results in a `UnionScan` and the range on the first index key is scanned completely.

Array indexes are very powerful and provide optimized execution of queries when array elements are used, something that is not possible with traditional databases. In this case, our data is not an array, but we can index it as one by creating a functional array.

```
CREATE INDEX idx_username_email_arr ON `ecommerce` (
  DISTINCT ARRAY v
    FOR v IN [LOWER(username), LOWER(email)]
  END
)
WHERE docType = 'user'
```

```
SELECT userId, pwd, firstName, lastName
FROM ecommerce
WHERE docType = 'user'
  AND ANY v IN [LOWER(username), LOWER(email)]
    SATISFIES v = 'johns'
  END
```

The explain plan verifies that only a single `DistinctScan` is performed against our index. This query would only expect a single result but is performing a "FETCH" as the fields `userId, pwd, firstName, lastName` are not in the index. If we wanted to squeeze every last bit of performance we could cover the query with the index:

```
CREATE INDEX idx_username_email_arr_cvr ON `ecommerce` (
  DISTINCT ARRAY v
    FOR v IN [LOWER(username), LOWER(email)]
  END,
  userId, username, pwd, firstName, lastName
)
WHERE docType = 'user'
```

Tip 19: Favor Equality Predicates over Ranges

Equality predicates are preferred and more performant than range scans, as a range is bounded by a low and high value, and the performance of the scan depends on how wide the range scan is. As an example, almost every application works with dates in some form or fashion, and are typically stored in either ISO-8601 (i.e. "2019-01-15T10:42:23Z") or Epoch time (i.e. "1547548943000") formats. Consider the following query and index to find all of the orders on a specific day:

```
CREATE INDEX idx_orderDate ON `ecommerce` (orderDate)
WHERE docType = "order"
```

```
SELECT orderId, orderDate, orderTotal
```

```
FROM ecommerce
WHERE docType = "order"
 AND orderDate >= "2019-01-15T00:00:00"
 AND orderDate < "2019-01-16"
```

This results in a range scan:

```
{
 "range": [{
   "high": "\"2019-01-16\"",
   "inclusion": 1,
   "low": "\"2019-01-15T00:00:00\""
 }]
}
```

However, if we create a functional index we can use a single equality predicate i.e. `WHERE orderDate = "2019-01-15"`

```
CREATE INDEX idx_orderDate_date ON `ecommerce` (SPLIT(orderDate, "T")[0])
WHERE docType = "order"
```

```
SELECT orderId, orderDate, orderTotal
FROM ecommerce
WHERE docType = "order"
 AND SPLIT(orderDate, "T")[0] = "2019-01-15"
```

Tip 20: Implement Index Replication

Indexes can (and should) be replicated across cluster-nodes. This ensures:

- *High Availability* (failover): If one Index-Service node is lost, the other continues to provide access to replicated indexes.
- *High Performance*: If original and replica copies are available, incoming queries are load-balanced automatically across them. This is contrary to the data service, where document replicas are "passive", index replicas are "active".

Prior to Couchbase Server 5.0, the only way to have "replica" indexes was to have the same index definition, but with a different name and manually place the indexes on different hosts.

```
CREATE INDEX productName_index1 ON bucket_name(productName, ProductID)
WHERE type="product"
WITH { "nodes": ["host1"] }

CREATE INDEX productName_index2 ON bucket_name(productName, ProductID)
WHERE type="product"
```

```
WITH { "nodes": ["host2"] }
```

As Couchbase Server 5.0+ index replicas are specified by using the `WITH` clause, simply specify the `num_replica` value.

```
CREATE INDEX productName_index1 ON bucket_name(productName, ProductID)
WHERE type="product"
WITH { "num_replica": 2 };
```

The only requirement is that the number of nodes in the cluster running the Index service is greater than or equal to `{num_replica} + 1`. Replicas can also be created by specifying the destination nodes of the index:

```
CREATE INDEX productName_index1 ON bucket_name(productName, ProductID)
WHERE type="product"
WITH { "nodes": ["node1:8091", "node2:8091", "node3:8091"] }
```

Additionally, both `num_replica` and `nodes` can be specified as long as `num_replica` is equal to the length of the `nodes` array + 1.

```
CREATE INDEX productName_index1 ON bucket_name(productName, ProductID)
WHERE type="product"
WITH { "num_replica": 2, "nodes": ["node1:8091", "node2:8091", "node3:8091"]
 }
```

Whenever a `CREATE INDEX` statement is issued, the default number of index replicas to create is `0`. This value can be changed, such that anytime a `CREATE INDEX` is performed there is no need to specify the WITH clause and replicas will be created automatically.

```
curl \
  -u Administrator:password \
  -d "{\"indexer.settings.num_replica\": 2 }"\
  http://localhost:9102/settings
```

Tip 21: Defer Index Builds to share DCP stream

When a `CREATE INDEX` statement is issued, each document in the keyspace must be projected against the index and by default, this is a synchronous operation. Meaning it will block until the index is built 100%, at which point in time the index will be updated asynchronously by any future mutations. This can be cumbersome and time-consuming, especially when managing many indexes.

In Couchbase, there can only be one index build process going on at a time. However, that does not mean that there can only be one index being built at a time. N1QL allows you to define the index but defer the actual building of the index to a later point in time, this is done using the `WITH { "defer_build": true }`.

```
CREATE INDEX `def_sourceairport` ON `travel-sample`(`sourceairport`)
WITH { "defer_build":true }

CREATE INDEX `def_city_state` ON `travel-sample`(`city`, `state`)
WITH { "defer_build":true }
```

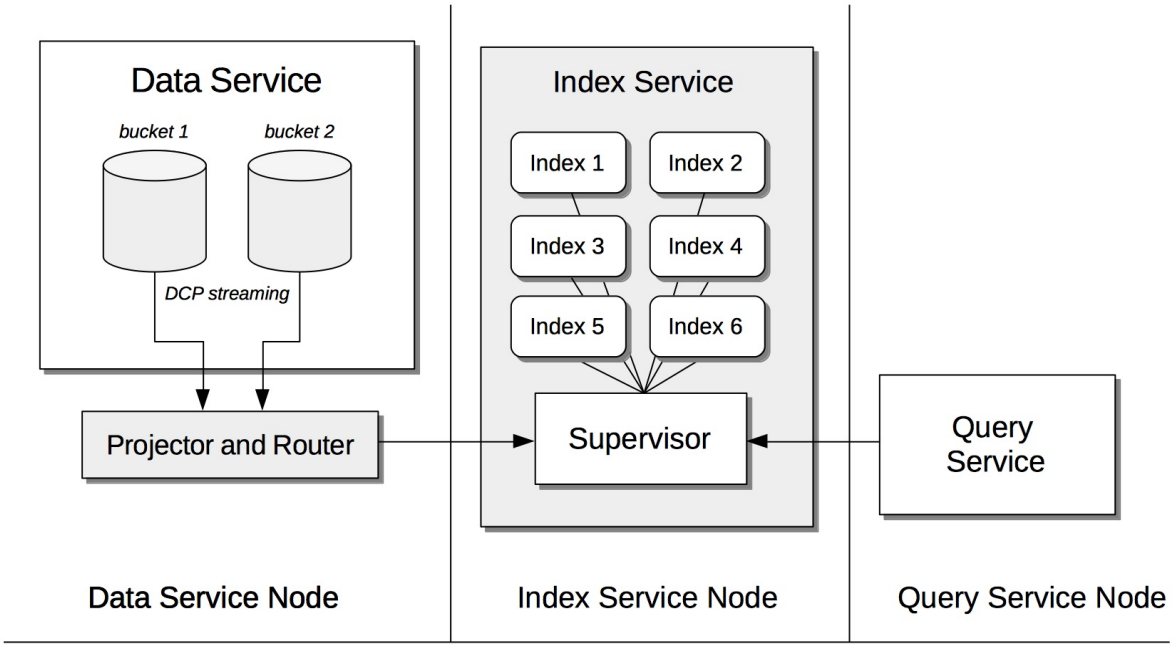
The indexes are in a "Created" state and not eligible to be used until they are built. To build multiple indexes at the same time a BUILD INDEX is used.

```
BUILD INDEX ON `travel-sample` (def_sourceairport, def_city_state)
```

The `BUILD INDEX` is asynchronous by default and has the primary benefit of allowing each index build to share the same DCP stream, and each document that is projected for the index build only has to be retrieved once instead of once per index.

Tip 22: Consider the Projection Selectivity of the Index

The Data, Index and Query service all work together to perform N1QL queries and manage indexes. While the bulk of index processing resides with the Index Service, there are two components of indexing that actually reside within the Data Service. These are the "Projector" and "Router" processes, which are responsible for projecting every data mutation against each index on the bucket and communicating those mutations to the "Supervisor" process on the Index Service.



This is roughly the # of documents that would qualify for the index divided by the total # of documents in the bucket.

For example, a bucket with `1,000,000` documents, and `100` of those documents were "config" documents, if we created an index with the predicate of `WHERE docType = "config"` the Projection Selectivity is 0.10%. Conversely, if you created an index on just `docType` for example and every

document in the bucket has `docType` the Projection Selectivity is 100%.

Both of these are on the extreme ends of the spectrum, the first will only satisfy projection 0.10% of the time and there would be many wasted CPU cycles on unnecessary projection and you may want to consider an alternative access pattern. Additionally, having a Projection Selectivity of 100% is also not performant as every mutation meets projection and results in an update to the index. There is not a set number, as it will be specific to each data set, but understand the implications of unnecessary projection.

Tip 23: Combine Multiples Scans Using `CASE` Expressions

Often, it is necessary to calculate different aggregates on various sets of documents. Usually, you achieve this goal with multiple scans on the data, but it is easy to calculate all the aggregates with a single scan. Eliminating n-1 scans can greatly improve performance.

You can combine multiple scans into one scan by moving the `WHERE` condition of each scan into a CASE expression, which filters the data for the aggregation. For each aggregation, there could be another field that retrieves the data.

The following example asks for the hotels which have free internet or free breakfast or free parking. You can obtain this result by executing three separate queries:

```
SELECT COUNT(*) FROM `travel-sample` WHERE type="hotel" and free_internet=tr
ue
SELECT COUNT(*) FROM `travel-sample` WHERE type="hotel" and free_breakfast=t
rue
SELECT COUNT(*) FROM `travel-sample` WHERE type="hotel" and free_parking=true
```

Total time = 119.23 + 121.63 + 118.71 = 359.57ms

However, it is more efficient to run the entire query in a single statement. Each number is calculated as one field. The count uses a filter with the `CASE` expression to count only the rows where the condition is valid. For example:

```
SELECT COUNT(CASE WHEN free_internet=true THEN 1 ELSE null END) cnt_free_int
ernet,
  COUNT(CASE WHEN free_breakfast=true THEN 1 ELSE null END) cnt_free_breakfa
st,
  COUNT(CASE WHEN free_parking=true THEN 1 ELSE null END) cnt_free_parking
FROM `travel-sample`
WHERE type="hotel"
```

The query takes 250 ms, shaving off an entire 100ms off the total time taken, not to mention the network calls, processing etc. that it would take while querying it from the application side.

This is a very simple example, larger datasets will show larger variances. There could be ranges involved, the aggregation functions could be different, etc.

Tip 24: Make your Documents Index Friendly

Different access paths can determine the optimum structure of your documents. When you are leveraging N1QL, you will want to ensure your documents are "index friendly". Ensure all documents have a consistent `docType` (or equivalent) attribute. This allows for efficient filtering during indexing, querying or both.

```
{
    "docType": "user",
    ...
}
```

If you're performing some processing on an array in the document, instead of performing that processing in the query consider storing a pre-computed value in another attribute (e.g. total, average, score, etc.)

```
{
    ...
    "scores": [
        78,
        97,
        23,
        ...,
        43
    ],
    "scoreTotal": 2392,
    "scoreAvg": 77.6,
    "scoreMin": 23,
    "scoreMax": 99
}
```

If an attribute name is dynamic or unknown, it's OK for KV access but not practical from an indexing standpoint. Below the first example is simply not practical to index or search if that object had every single language listed in there, you would need one index per language.

Bad for Index/Querying:

```
{
    "Greetings": {
        "English": "Good Morning",
        "Spanish": "Buenos días",
        "German": "Guten Morgen",
        "French": "Bonjour"
    }
}
```

Good for Index/Querying:

```
{
    "Greetings": [{
            "Language": "English",
            "Greeting": "Good Morning"
        },
        {
            "Language": "Spanish",
            "Greeting": "Buenos días"
        },
        {
            "Language": "German",
            "Greeting": "Guten Morgen"
        }, {
            "Language": "French",
            "Greeting": "Bonjour"
        }
    ]
}
```

Tip 25: USE INFER to understand your dataset

Couchbase buckets are a logical namespace made up of one or more types of documents/models. Understanding these models and their underlying structures can provide additional opportunities for performance and tuning. N1QL supports the INFER statement, that is statistical in nature. It analyzes a bucket through sampling and returns results in JSON Schema format.

```
INFER `travel-sample` WITH {
  "sample_size": 1000,
  "num_sample_values": 5,
  "similarity_metric": 0.6
}
```

For each identified attribute, the statement returns the following details:

Attribute	Description
#docs	Specifies the number of documents in the sample that contain this attribute.
%docs	Specifies the percentage of documents in the sample that contain this attribute.
minitems	If the data type is an array, specifies the minimum number of elements (array size).
maxitems	If the data type is an array, specifies the maximum number of elements (array size).
samples	Displays a list of sample values for the attribute found in the sample population.
type	Specifies the identified data type of the attribute.

Tip 26: META() properties such as CAS and Expiration can be Indexed

CAS or Compare-and-Swap values are used as a form of optimistic locking for document mutations. The CAS value can be returned from a N1QL query by calling the `META()` function and referencing the `META().cas` value.

If you are leveraging CAS operations and attempting to cover your queries, this value will need to be added to the index.

Tip 27: Beware of the Scan Consistency Level

N1QL indexes are updated asynchronously after a mutation has occurred on the data service. While key-value operations against the data service are strongly consistent, and while index updates are extremely fast (usually <1ms), because of the asynchronous nature they are "eventually consistent".

N1QL allows you to specify the Scan Consistency level to use for a query. There are three possible values:

- **Not Bounded:** Is the default and the fastest. It says to return the results that are currently in the index, regardless or not if there are items in the queue waiting to be indexed.
- **Request Plus:** Is the opposite and slowest of the scan levels. At query time it will wait for all indexes to catch up to their highest sequence numbers. The benefit of this slower scan level is it allows you to Read Your Own Write (RYOW) but could take some time if the system is under heavy write load.
- **Statement Plus / At Plus:** Is unique compared to the other two scan levels and requires `mutationTokens` to be enabled before it can be used. This causes a few extra bytes of information to be sent back to the SDK whenever a mutation occurs. The mutation tokens are then passed to the N1QL query and the query will wait for at least those tokens to be indexed prior to proceeding.

```
// enable mutation tokens
CouchbaseEnvironment env = DefaultCouchbaseEnvironment
    .builder()
    .mutationTokensEnabled(true)
    .build();
// mutate a document and save the mutation
JsonDocument written = bucket.upsert(JsonDocument.create("mydoc", JsonObject
.empty()));
// written.mutationToken() ==
//  "mt{vbID=55, vbUUID=166779084590420, seqno=488, bucket=travel-sample}"
// pass the mutation to the query
bucket.query(
    N1qlQuery.simple("select count(*) as cnt from `travel-sample`",
    N1qlParams.build().consistentWith(written))
);
```

There is a price to pay for using a query scan consistency such as Request+ or Statement+. See if you can achieve the desired result by key-value Access, especially if it's a single document or a handful of documents which can be obtained via a multi-get.

Tip 28: Use IN instead of WITHIN

The IN operator specifies the search depth to only include the current-level of the array on which it is operating against. Whereas the WITHIN operator specifies the search depth to include the current level of the array it's operating on and *all* of the children and descendant arrays indefinitely.

The use of `WITHIN` can have unexpected performance consequences if used incorrectly, by recursively iterating through all arrays of a given property.

Tip 29: Cancel Long Running or Problematic Requests

Active N1QL requests are stored in the `system:active_requests` catalog, if the result of the query is greater than `1s` it will be stored in the `system:completed_requests` catalog, otherwise it is discarded. If you need to cancel any active requests, you can issue a `DELETE` statement against the `system:active_requests` keyspace effectively canceling the query.

```
DELETE
FROM system:active_request
WHERE requestId = "..."
```

Tip 30: Cleanup system:completed_requests

The `system:completed_requests` is extremely useful with identifying slow performing and resource intensive queries. As you continue to iterate through each of the queries and optimize them, you no longer want to see that same query in `system:completed_requests`. You can delete records from the `system:completed_requests` catalog, just as you would with any other keyspace.

```
DELETE
FROM system:completed_requests
WHERE requestId = "..."

DELETE
FROM system:completed_requests
WHERE statement = "SELECT DISTINCT type\nFROM `travel-sample`"
```

Tip 31: Initially Design Queries / Indexes on an Empty Bucket

Designing, building and iterating on indexes against a bucket with a large number of documents can be time-consuming. When initially designing queries and indexes, if possible, execute them against an empty bucket until you're satisfied with the explain plan.

Tip 32: Remove Unused Indexes

Proactive monitoring is critical to any application, not only should you be actively monitoring indexes and their usage to identify potential growth needs, you should also be monitoring and identifying indexes which are not used and can be dropped.

Tip 33: Set clientContextID option in the SDK

The `clientContextID` is a user-defined identifier query option that can be sent to the query service from the SDK. This is not used by Couchbase for anything, however, it is stored in the `system:active_requests` and `system:completed_requests`. This can be a true application requestId/genesisId generated by the application, or some type of application identifier that can be later used for debugging or tracking down specific queries related to a given application or request.

Appendix: Operators

#operator	Usage	Description
PrimaryScan	Scan	Scans a primary index v1 (Pre CB 5.50)
PrimaryScan3	Scan	Scans a primary index v2 (CB 5.50+)
ParentScan	Scan	Used for `UNNEST`. Treats the parent object as the result of a scan.
IndexScan	Scan	Scans a secondary index v1 (CB pre 5.0)
IndexScan2	Scan	Scans a secondary index v2 (CB 5.0)
IndexScan3	Scan	Scans a secondary index v3 (CB 5.50+)
KeyScan	Scan	Does not perform a scan. Directly treats the provided keys as a scan.
ValueScan	Scan	Used for the VALUES clause of `INSERT` and `UPSERT` statements. Treats the provided values as the result of a scan.
DummyScan	Scan	Used for `SELECT`s with no `FROM` clause. Provides a single empty object as the result of a scan.
CountScan	Scan	Used when the query has no predicate i.e. `SELECT COUNT(*) FROM bucket-name`. Treats the bucket size as the result of a scan, without actually performing a full scan of the bucket.
IndexCountScan	Used when the query has predicates and the predicate can be pushed to the indexer. Count is performed by Indexer. (v1 Pre CB 5.0)	
IndexCountScan2	Scan	Used when the query has predicates and the predicate can be pushed to the indexer. Count is performed by Indexer. (v2 CB 5.0)
IndexCountDistinctScan2	Scan	Used when the query has predicates and the predicate can be pushed to the indexer. Count of Distinct values is performed by Indexer. (v2 CB 5.0)
IntersectScan	Scan	A container that scans its child scanners and intersects the results. Used for scanning multiple secondary indexes

		concurrently for a single query. Intersect of document keys are done
OrderedIntersectScan	Scan	Same as IntersectScan, First scan in the order is maintained (First Scan Index order is exposed becaue it matches query `ORDER BY`)
UnionScan	Scan	`OR` predicate can use multiple indexes. Each Index perform IndexScan document keys are merged as `UNION`
DistinctScan	Scan	Eliminates Duplicate document keys (Indexer can produce duplicate document keys. Array Indexing, Overlap `OR` clauses)
ExpressionScan	Scan	Source (`FROM` clause) is not a key space. It is Expression. Expression Scan will be performed from the in memory data.
Fetch	Fetch	Obtain documents from the data service based on a key
DummyFetch	Fetch	No Fetch Operations performed. Act like dummy fetch
Join	Join	Used for Look-up Join. `a JOIN b ON KEYS`
IndexJoin	Join	Used for Index Join. `a JOIN b ON KEY b.xxx FOR a`
NestedLoopJoin	Join	ANSI JOIN (`a JOIN b ON a.xx = b.yy`). The Join is performed nested loop every row of a, index scan/fetch is performed on b
HashJoin	Join	ANSI JOIN (`a JOIN b ON a.xx = b.yy`). The Join is performed Hash JOIN, In memory hash table will be constructed on a or b. Then a or b scanned/fetched and look up done in memory hash table
Nest	Join	Join operation between a parent and a child with a nested array where parent is repeated for each child array item. Same as Join (`a NEST b ON KEYS a.xxx`)
IndexNest	Join	Same as Index JOIN (`a NEST b ON KEY b.xxx FOR a`)
NestedLoopNest	Join	Same as Nested LOOP JOIN (`a NETST b ON a.xx = b.yy`)
HashNest	Join	Same as Hash JOIN (`a NEST b ON a.xx = b.yy`)
		Grouping operation between a parent and

Unnest	Join	a child array where child array is embedded into the parent.
Let	Let + Letting	Let and Letting varaibles evaluation
InferKeyspace	Infer	INFER statement
Filter	Filter	Apply a filter expression e.g. `WHERE X= <value>`
InitialGroup	Group	Initial phase. (Can be executed in parallel with IntermediateGroup)
IntermediateGroup	Group	Cumulate intermediate results. This phase can be chained.
FinalGroup	Group	Compute final aggregate results.
InitialProject	Project	Reduce the stream size to the fields involved in the query processing
FinalProject	Project	Final Shaping of the result into the JSON for the requested fields
IndexCountProject	Project	Project the output of IndexCountScan/2 IndexCountDistinctScan2 operators
Distinct	Distinct	Indicates that duplicates are being filtered from the result.
UnionAll	Set Operator	Combine the results of two queries. For `UNION` , we perform `UNION ALL` followed by DISTINCT.
IntersectAll	Set Operator	Intesect the all of the result objects
ExceptAll	Set Operator	Except all of the result objects (i.e Present on LEFT side query but not present on right side query)
Order	Order	Orders the results based on 1 or more keys ASC or DESC
Offset	Paging	Start returning items from a specified item count
Limit	Paging	Limit the number of items returned to N
SendInsert	Insert	When an insert statement is explained
SendUpsert	Upsert	When an upsert statement is explained
SendDelete	Delete	When an delete statement is explained
Clone	Update	Used for UPDATE. Clones data values so that UPDATEs read original values and mutate a clone.
Set	Update	Used for UPDATE
Unset	Update	Used for UPDATE
SendUpdate	Update	When an update statement is explained
Merge	Merge	Merge Statement
Alias	Framework	Alias of Keyspace

Authorize	Framework	Privilege validation (i.e permission RBAC validation on all objects in the query)
Parallel	Framework	A container that executes multiple copies of its child operator in parallel. Used for all data-parallelism.
Sequence	Framework	A container that chains its children into a sequence. Used for all execution pipelining.
Discard	Framework	Discard results
Stream	Framework	Stream results out. Used for returning results.
Collect	Framework	Collect results into an array. Used for subqueries.
CreatePrimaryIndex	Index DDL	When a `Create PRIMARY INDEX` statement is explained
CreateIndex	Index DDL	When a `CREATE INDEX` statement is explained
DropIndex	Index DDL	When a `DROP INDEX` statement is explained
AlterIndex	Index DDL	When a `ALTER INDEX` statement is explained
BuildIndexes	Index DDL	When a `BUILD INDEX` statement is explained
GrantRole	Roles	GRANT statement
RevokeRole	Roles	Revoke Statement
Explain	Explain	EXPLAIN statement
Prepare	Prepare	Prepare Statement

Resources

- N1QL: A Practical Guide (2nd Edition)
- A Guide to N1QL in Couchbase 5.5
- Index Scans
- N1QL Monitoring
- Deep Dive into Couchbase N1QL Query Optimization
- Optimize N1QL Performance using Request Profiling
- Grouping and Aggregation in Couchbase
- Nitro: A Fast, Scalable In-Memory Storage Engine for NoSQL Global Secondary Index
- Query Tutorial
- Couchbase Training Online - N1QL