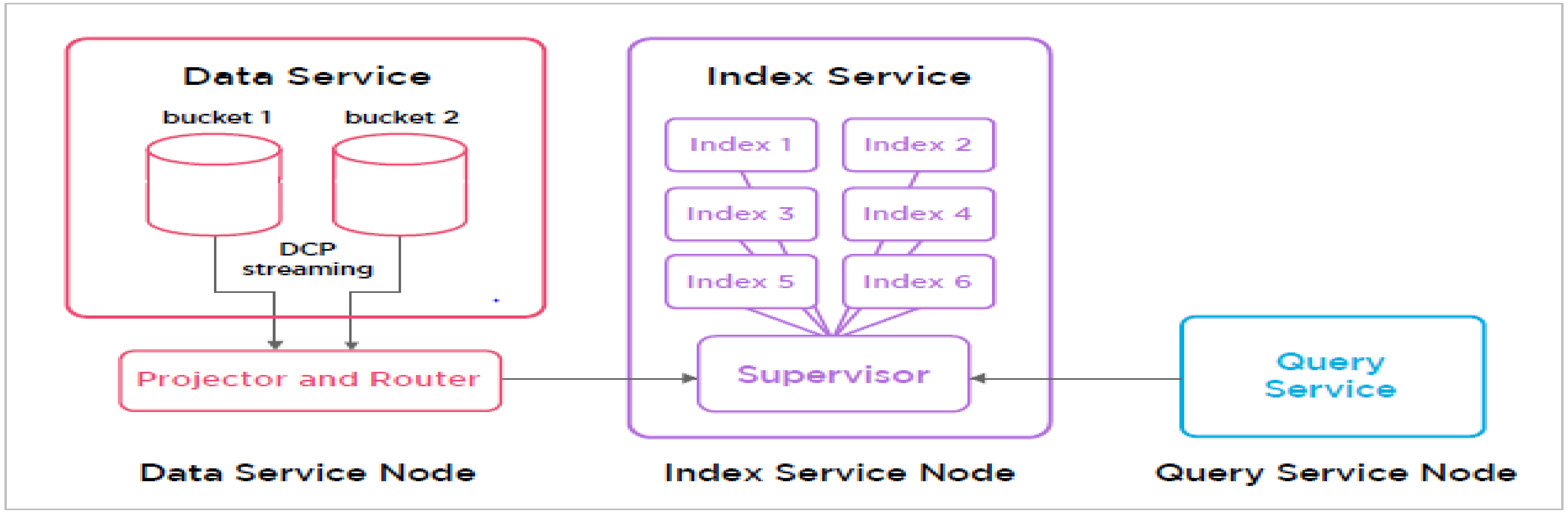
Indexes in Couchbase

ANJU MUNOTH

Index Service Architecture

- Components essential for the Index Service reside not only on each node to which the Index Service is assigned, but also on each node to which the Data Service is assigned



Index Service Architecture

- Data Service: Uses the DCP protocol to stream data-mutations to the Projector and Router process, which runs as part of the Index Service, on each Data Service node.
- Projector and Router: Provides data to the Index Service, according to the index-definitions provided by the Index Service Supervisor.
- When the Projector and Router starts running on the Data Service-node, the Data Service streams to the Projector and Router copies of all mutations that occur to bucket-items.
- Prior to the creation of any indexes, the Projector and Router takes no action.
- When an index is first created, the Index Service Supervisor contacts the Projector and Router; and passes to it the corresponding index-definitions.
- The Projector and Router duly contacts the Data Service, and extracts data from the fields specified by the index-definitions.
- It then sends the data to the Supervisor, so that the index can be populated.
- Subsequently, the Projector and Router continuously examines the stream of mutations provided by the Data Service.
- When this includes a mutation to an indexed field, the mutated data is passed by the Projector and Router to the Supervisor, and the index thereby updated.

Index Service Architecture

- Supervisor: The main program of the Index Service; which passes index-definitions to the Projector and Router, creates and stores indexes, and handles mutations sent from the Projector and Router.

- Query Service: Passes to the Supervisor clients' create-requests and queries, and handles the responses

Saving Indexes

- By default, an index is saved on the node on which it is created.
- Each index is created on one bucket only; but multiple indexes may be created on a single bucket.

- The Index Service can be configured to use either standard or memory-optimized storage

Standard

- The supervisor process persists to disk all changes made to individual indexes. Each index gets a dedicated file.
- In Couchbase Server Community Edition, writes can either be append-only or write with circular reuse, depending on the write mode selected for global secondary indexes.
- In Couchbase Server Enterprise Edition, compaction is handled automatically.

Memory-Optimized

- Only available in Couchbase Server Enterprise Edition.
- Indexes are saved in-memory.
- Provides increased efficiency for maintenance, scanning, and mutation.
- A snapshot of the index is maintained on disk, to permit rapid recovery if node-failures are experienced.
- A skiplist (rather than a conventional B-tree) structure is used; optimizing memory consumption.
- Lock-free index-processing enhances concurrency.

Global Secondary Indexes

- Secondary Indexes, often referred to as Global Secondary Indexes or GSIs
- Constitute the principal means of indexing documents to be accessed by the Query Service.
- Secondary Indexes can be created on specified fields; placed on specific cluster-nodes; and replicated.

Features of GSI

- Advanced Scaling: GSIs can be assigned independently to selected nodes, without existing workloads being affected.
- Predictable Performance: Key-based operations maintain predictable low-latency, even in the presence of a large number of indexes.
  - Index-maintenance is non-competitive with key-based operations, even when data-mutation workloads are heavy.
- Low Latency Querying: GSIs independently partition into the Index Service nodes: they do not have to follow hash partitioning of data into vBuckets.
  - Queries using GSIs can achieve low latency response times even when the cluster scales out; since GSIs do not require a wide fan-out to all Data Service nodes.
- Independent Partitioning: The Index Service provides partition independence: data and its indexes can have different partition keys.
  - Each index can have its own partition key, so each can be partitioned independently to match the specific query.
  - As new requirements arise, the application will also be able to create a new index with a new partition key, without affecting performance of existing queries.

Index Replication

Secondary indexes can be replicated across cluster-nodes. This ensures:

- Availability: 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 across them.

Index Replication

- Index-replicas can be created with the N1QL CREATE INDEX statement.
- Whenever a given number of index-replicas is specified for creation, the number must be less than the number of cluster-nodes currently running the Index Service.
- If it is not, the index creation fails.
- If, following creation of the maximum number of copies, the number of nodes running the Index Service decreases, Couchbase Server progressively assigns replacement index-replicas to any and all Index-Service nodes subsequently be added to the cluster, until the required number of index-replicas again exists for each replicated index.

Creating Index Replicas

- Specifying, by means of the WITH clause, the destination nodes. In the following example, an index with two replicas is created. The active index is on node1, and the replicas are on node2 and node3:

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

Creating Index Replicas

- Specifying no destination nodes; but specifying instead, by means of the WITH clause and the num_replica attribute, only the number of replicas required.
- The replicas are automatically distributed across those nodes of the cluster that are running the Index Service: the distribution-pattern is based on a projection of optimal index-availability, given the number and disposition of Index-Service nodes across defined server-groups.
- In the following example, an index is created with two replicas, with no destination-nodes specified:

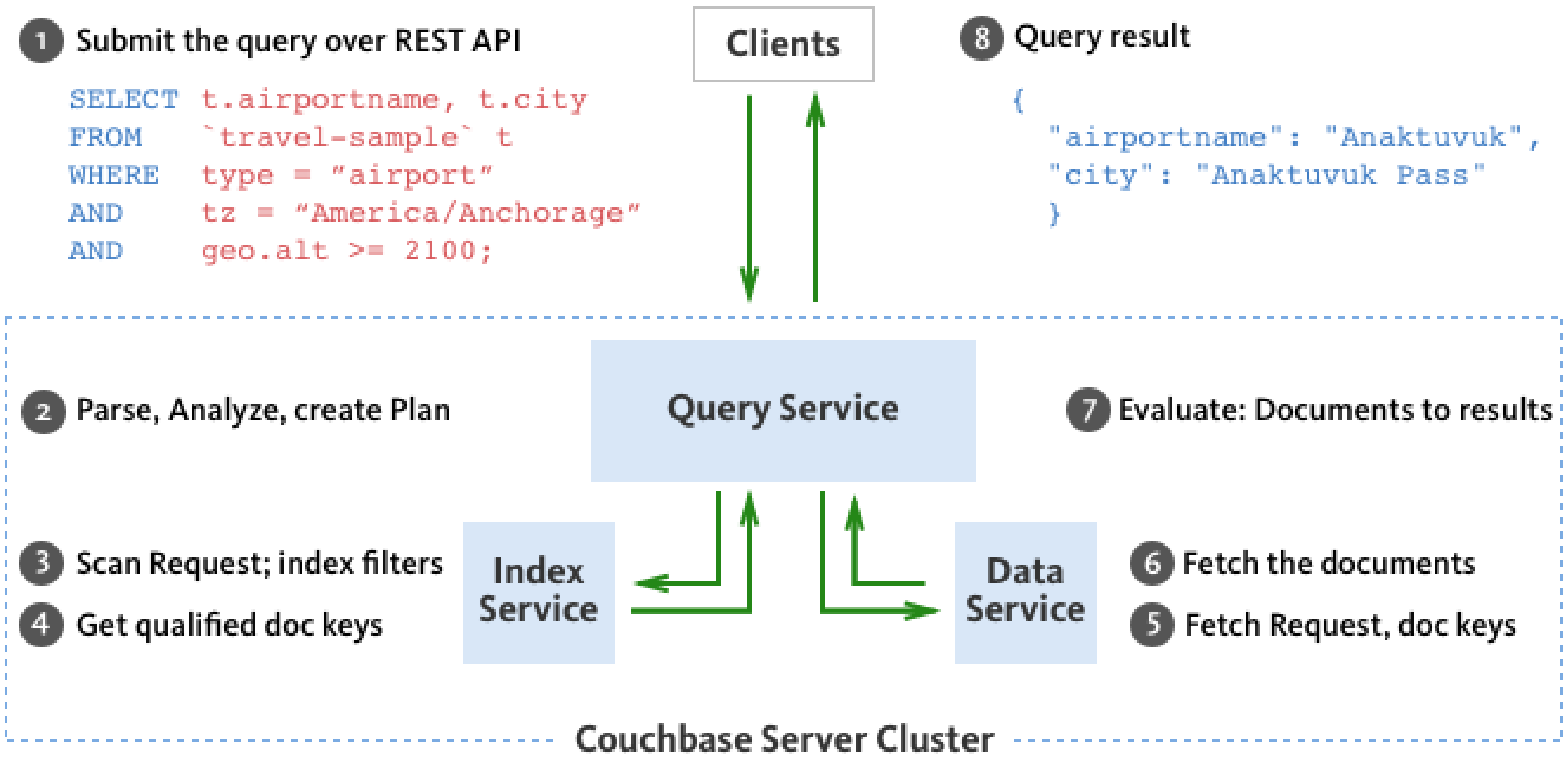
```
CREATE INDEX productName_index1 ON bucket_name(productName, ProductID)
    WHERE type="product" USING GSI
    WITH {"num_replica": 2};
```

Index Partitioning

- Index Partitioning increases query performance, by dividing and spreading a large index of documents across multiple nodes.
- This feature is available only in Couchbase Server Enterprise Edition.

The benefits include:

- The ability to scale out horizontally, as index size increases.
- Transparency to queries, requiring no change to existing queries.
- Reduction of query latency for large, aggregated queries; since partitions can be scanned in parallel.
- Provision of a low-latency range query, while allowing indexes to be scaled out as needed.


1 Submit the query over REST API
Clients
8 Query result
2 Parse, Analyze, create Plan
Query Service
7 Evaluate: Documents to results
3 Scan Request; index filters
4 Get qualified doc keys
Index Service
Data Service
6 Fetch the documents
5 Fetch Request, doc keys
Couchbase Server Cluster

```
SELECT t.airportname, t.city
FROM   `travel-sample` t
WHERE  type = "airport"
AND    tz = "America/Anchorage"
AND    geo.alt >= 2100;
```

```
{
  "airportname": "Anaktuvuk",
  "city": "Anaktuvuk Pass"
  }
```

Indexes

- Primary: Provided by the Index Service, Based on the unique key of every item in a specified bucket.
  - Every primary index is maintained asynchronously.
  - A primary index is intended to be used for simple queries, which have no filters or predicates.
- Secondary: Provided by the Index Service, this is based on an attribute within a document.
  - The value associated with the attribute can be of any type: scalar, object, or array.
  - Frequently referred to as a Global Secondary Index, or GSI.
  - Used most frequently in Couchbase Server, for queries performed with the N1Ql query-language.
- Full Text: Provided by the Search Service, this is a specially purposed index, which contains targets derived from the textual contents of documents within one or more specified buckets.
  - Text-matches of different degrees of exactitude can be searched for.
  - Both input and target text-values can be purged of irrelevant characters (such as punctuation marks or html tags).
- View: Supports Couchbase Views, with fields and information extracted from documents.

Couchbase 5.0- Index categories

Each Couchbase cluster can only have one category of index, either a standard global secondary index or a memory optimized global secondary index.

- Standard Secondary: Release 4.0 and above
  - *Based on ForestDB
  - *Released with Couchbase 4.0
- Memory Optimized Index: 4.5 and above
  - *100% of the index is in memory
  - *Index is written to disk for recovery only
  - *Predictable Performance
  - *Better mutation rate
- Standard Secondary: Release 5.0
  - * Uses the lockless skiplist based Plasma storage engine for enterprise edition
  - * Uses ForestDB storage engine for community edition
  - * Released with Couchbase 5.0.
  - * Designed to handle very large datasets

Couchbase 5.0- Index categories

- standard secondary index (from 4.0 to 4.6.x) stores uses the ForestDB storage engine to store the B-Tree index and keeps the optimal working set of data in the buffer.
  - Total size of the index can be much bigger than the amount of memory available in each index node.
- A memory-optimized index uses a novel lock-free skiplist to maintain the index and keeps 100% of the index data in memory.
  - A memory-optimized index (MOI) has better latency for index scans and can also process the mutations of the data much faster.
- The standard secondary index in 5.0 uses the plasma storage engine in the enterprise edition, which uses the lock-free skip list like MOI, but supports large indexes that don't fit in memory.
- All three types of indexes implement multi-version concurrency control (MVCC) to provide consistent index scan results and high throughput. During cluster installation, choose the type of index.

Types of Indexes

- Primary Index
- Named primary index
- Secondary index
- Composite Secondary Index
- Functional index
- Array Index
- ALL array
- ALL DISTINCT array
- Partial Index
- Adaptive Index
- Duplicate Indices
- Covering Index

Primary Index

- Primary index is simply the index on the document key on the whole bucket.
- Couchbase data layer enforces the uniqueness constraint on the document key.
- The primary index, like every other index in Couchbase, is maintained asynchronously.
- used for full bucket scans (primary scans) when the query does not have any filters (predicates) or no other index or access path can be used.
- In Couchbase, you store multiple keyspaces (documents of a different type, customer, orders, inventory, etc.) in a single bucket.
- So, when you do the primary scan, the query will use the index to get the document-keys and fetch all the documents in the bucket and then apply the filter. So, this is VERY EXPENSIVE.

metadata Primary index

```
select * from system:indexes where name = '#primary';
"indexes": {
  "datastore_id": "http://127.0.0.1:8091",
  "id": "f6e3c75d6f396e7d",
  "index_key": [],
  "is_primary": true,
  "keyspace_id": "travel-sample",
  "name": "#primary",
  "namespace_id": "default",
  "state": "online",
  "using": "gsi"
 }
```

Named Primary Index

- Can create multiple replicas of any index with a simple parameter to CREATE INDEX.
- The following will create 3 copies of the index and there has to be a minimum of 3 index nodes in the cluster.

```
CREATE PRIMARY INDEX ON 'travel-sample' WITH {"num_replica":2};
CREATE PRIMARY INDEX `def_primary` ON `travel-sample` ;
```

- Can also name the primary index.
- For the rest of the features of the primary index are the same, except the index is named.
- A good side effect of this is that you can have multiple primary indices in Couchbase versions before 5.0 using different names.
- Duplicate indices help with high availability as well as query load distribution throughout them. This is true for both primary indices and secondary indices.

Secondary Index

- Secondary index is an index on any key-value or document-key.
- Index can be any key within the document.
- Key can be of any time: scalar, object, or array.
- Query has to use the same type of object for the query engine to exploit the index.

Secondary Index

```
CREATE INDEX travel_name ON `travel-sample`(name);
```

name is a simple scalar value.

```
{   "name": "Air France"  }
```

```
CREATE INDEX travel_geo on `travel-sample`(geo);
```

geo is an object embedded within the document. Example:

```
  "geo": {
      "alt": 12,
      "lat": 50.962097,
      "lon": 1.954764
      }
```

Creating indexes on keys from nested objects is straightforward.

```
CREATE INDEX travel_geo on `travel-sample`(geo.alt);
CREATE INDEX travel_geo on `travel-sample`(geo.lat);
```

Composite Secondary Index

- Common to have queries with multiple filters (predicates).
- Want the indices with multiple keys so the indices can return only the qualified document keys.
- Additionally, if a query is referencing only the keys in the index, the query engine will simply answer the query from the index scan result without going to the data nodes.
- This is a commonly exploited performance optimization.

```
CREATE INDEX idx_stctln ON `travel-sample` (state, city, name.lastname)
```

- Each of the keys can be a simple scalar field, object, or an array.
- For the index filtering to be exploited, the filters have to use respective object type in the query filter.
- The keys to the secondary indices can include document keys (meta().id) explicitly if you need to filter on it in the index.

1.SELECT * FROM `travel-sample` WHERE state = 'CA';

The predicate matches the leading key of the index. So, this query uses the index to fully evaluate the predicate (state = 'CA').

2.SELECT * FROM `travel-sample` WHERE state = 'CA' AND city = 'Windsor';

The predicates match the leading two keys. So this is good fit as well.

3.SELECT * FROM `travel-sample` WHERE state = 'CA' AND city = 'Windsor' AND name.lastname = 'smith';

The three predicates in this query matches the three index keys perfectly. So, this is a good match.

4.SELECT * FROM `travel-sample` WHERE city = 'Windsor' AND name.lastname = 'smith';

In this query, although predicates match two of the index keys, the leading key isn't matched. So, the index cannot and is not used for this query plans.

5.SELECT * FROM `travel-sample` WHERE name.lastname = 'smith';
Similar to previous query, this query has the predicate on the third key of the index. So, this index cannot be used.

6.SELECT * FROM `travel-sample` WHERE state = 'CA' AND name.lastname = 'smith';

This query has predicate on first and the third key. While this index is and can be chosen, we cannot push down the predicate after skipping an index key (second key in this case).   So, only the first predicate (state = "CA") will be pushed down to index scan.

```
        "#operator": "IndexScan2",
        "index": "idx_stctln",
        "index_id": "dadbb12da565ed28",
        "index_projection": {          "primary_key": true         },
        "keyspace": "travel-sample",
        "namespace": "default",
        "spans": [
          {
            "exact": true,
            "range": [
              {                   "high": "\"CA\"",                    "inclusion": 3,                    "low": "\"CA\""              }
            ]
          }
```

7.SELECT * FROM `travel-sample` WHERE state IS NOT MISSING AND city = 'Windsor' AND name.lastname = 'smith';

This is a modified version of query 4 above. To use this index, query needs to have additional predicate (state IS NOT

Functional (Expression) Index

- It's common to have names in the database with a mix of upper and lower cases. When you need to search, "John," you want it to search for any combination of "John," "john," etc.
- Here's how you do it.

```
CREATE INDEX travel_cxname ON `travel-sample`(LOWER(name));
```

- Provide the search string in lowercase and the index will efficiently search for already lowercased values in the index.

- can use complex expressions in this functional index.

```
CREATE INDEX travel_cx1 ON `travel-sample`(LOWER(name), length*width, round(salary));
```

Array Index

- JSON is hierarchical. At the top level, it can have scalar fields, objects, or arrays. Each object can nest other objects and arrays. Each array can have other objects and arrays. And so on. The nesting continues.

CREATE INDEX travel_schedule ON `travel-sample`(schedule);

CREATE INDEX travel_sched ON `travel-sample`

(ALL DISTINCT ARRAY v.day FOR v IN schedule END)

- This index key is an expression on the array to clearly reference only the elements needed to be indexed. schedule is the array we're dereferencing into. v is the variable we've implicitly declared to reference each element/object within the array: schedule v.day refers to the element within each object of the array schedule.
- SELECT * FROM `travel-sample`WHERE ANY v IN SCHEDULE SATISFIES v.day = 2 END;

Partial Index

- Couchbase data model is JSON and JSON schema are flexible, an index may not contain entries to documents with absent index keys.
- Couchbase buckets can have documents of various types.
- Typically, customers include a type field to differentiate distinct types.
- to create an index of airline documents, you can simply add the type field for the WHERE clause of the index.

```
CREATE INDEX travel_info ON `travel-sample`(name, id, icoo, iata) WHERE type = 'airline’;
```

- create an index only on the documents that have (type = 'airline'). In your queries, you'd need to include the filter (type = 'airline') in addition to other filters so this index qualifies.

Partial Index

Various use cases to exploit partial indexes are:

- Partitioning a large index into multiple indices using the mod function.
- Partitioning a large index into multiple indices and placing each index into distinct indexer nodes.
- Partitioning the index based on a list of values. For example, you can have an index for each state.
- Simulating index range partitioning via a range filter in the WHERE clause. One thing to remember is Couchbase N1QL queries will use one partitioned index per query block. Use UNION ALL to have a query exploit multiple partitioned indices in a single query.

Adaptive Index

- Adaptive index creates a single index on the whole document or set of fields in a document.
- This is a form or array index using {"key":value} pair as the single index key.
- Purpose is to avoid the bane of the query having to match the leading keys of the index in traditional indexes.
- There are two advantages with Adaptive index:
  - Multiple predicates on the keyspace can be evaluated using different sections of the same index.
  - Avoid creating multiple indexes just to reorder the index keys.
  - Avoid the index key-order.

Adaptive Index

```
CREATE INDEX `ai_self`
   ON `travel-sample`(DISTINCT PAIRS(ai_self))
   WHERE type = "airport";


SELECT * FROM `travel-sample`
   WHERE faa = "SFO" AND `type` = "airport";
```

- Same index can be used for queries with other predicates as well.
- Reduces the number of indexes you'd need to create as the document grows.

```
SELECT * FROM `travel-sample`  WHERE city = "Seattle" AND `type` = "airport";
```

Duplicate Index

- This isn't really a special type of index, but a feature of Couchbase indexing.

Can create duplicate indexes with distinct names.

```
CREATE INDEX i1 ON `travel-sample`(LOWER(name),id, icoo) WHERE type = ‘airline’;
CREATE INDEX i2 ON `travel-sample`(LOWER(name),id, icoo) WHERE type = ‘airline’;
CREATE INDEX i3 ON `travel-sample`(LOWER(name),id, icoo) WHERE type = ‘airline’;
```

Or

```
CREATE INDEX i1 ON `travel-sample`(LOWER(name),id, icoo) WHERE type = ‘airline’ WITH {"num_replica" : 2 };
```

- All three indices have identical keys, an identical WHERE clause; the only difference is the name of the indices.
- Can choose their physical location using the WITH clause of the CREATE INDEX.
- During query optimization, the query will choose one of the names.
- During query runtime, these indices are used in a round-robin fashion to distribute the load.
- This gives you scale-out, multi-dimensional scaling, performance, and high availability.

Covering Index

- Index selection for a query solely depends on the filters in the WHERE clause of your query.
- After the index selection is made, the engine analyzes the query to see if it can be answered using only the data in the index.
- If it does, the query engine skips retrieving the whole document.
- This is a performance optimization to consider while designing the indices.

Create a secondary index that contains airports with an alt value greater than 1000 on the node 192.0.2.1.

- CREATE INDEX over1000 ON `travel-sample`(geo.alt) WHERE geo.alt > 1000 USING GSI WITH {"nodes": ["192.0.2.1:8091"]};

Create a deferred index

- Create a secondary index with the defer_build option.

```
CREATE INDEX `travel-sample-type-index` ON `travel-sample`(type) USING GSI
   WITH {"defer_build":true};
```

- Query system:indexes for the status of the index.

```
SELECT * FROM system:indexes WHERE name="travel-sample-type-index";
```

Build a deferred index

- Kick off a deferred build using the index name.

BUILD INDEX ON `travel-sample`(`travel-sample-type-index`) USING GSI;

- Query system:indexes for the status of the index.

SELECT * FROM system:indexes WHERE name="travel-sample-type-index";

Indexing all DISTINCT elements in an array

- Create an index on all schedules

```
CREATE INDEX idx_sched
ON `travel-sample` ( DISTINCT ARRAY v.flight FOR v IN schedule END );
```

- Q1: Find the list of scheduled 'UA' flights

```
SELECT * FROM `travel-sample`
WHERE ANY v IN schedule SATISFIES v.flight LIKE 'UA%' END;
```

Partial Index

- Partial index (with WHERE clause) of individual attributes from selected elements (using WHEN clause) of an array
- Create an index on flight IDs scheduled in the first 4 days of the week

```
CREATE INDEX idx_flight_day
ON `travel-sample` ( ALL ARRAY v.flight FOR v IN schedule WHEN v.day < 4 END )
WHERE type = "route" ;
```

- Find the list of scheduled 'UA' flights on day 1

```
SELECT * FROM `travel-sample`
WHERE type = "route"
AND ANY v IN schedule SATISFIES (v.flight LIKE 'UA%')
AND (v.day=1) END;
```

Adaptive Index

```
CREATE INDEX `ai_airport_day_faa` ON `travel-sample`(DISTINCT
PAIRS({airportname, city, faa, type})) WHERE type = "airport";
```