Contents

# Couchbase Architecture

References: <https://dzone.com/articles/couchbase-architecture-deep>

Couchbase is the merge of two popular NOSQL technologies:

* Membase, which provides persistence, replication, sharding to the high performance memcached technology
* CouchDB, which pioneers the document oriented model based on JSON

Transaction Model

Atomicity is guaranteed at a single document and transactions that span update of multiple documents are unsupported.  To provide necessary isolation for concurrent access, Couchbase provides a CAS (compare and swap) mechanism which works as follows …

* When the client retrieves a document, a CAS ID (equivalent to a revision number) is attached to it.
* While the client is manipulating the retrieved document locally, another client may modify this document.  When this happens, the CAS ID of the document at the server will be incremented.
* Now, when the original client submits its modification to the server, it can attach the original CAS ID in its request.  The server will verify this ID with the actual ID in the server.  If they differ, the document has been updated in between and the server will not apply the update.
* The original client will re-read the document (which now has a newer ID) and re-submit its modification.

Couchbase also provides a locking mechanism for clients to coordinate their access to documents.  Clients can request a LOCK on the document it intends to modify, update the documents and then releases the LOCK.  To prevent a deadlock situation, each LOCK grant has a timeout so it will automatically be released after a period of time.

Install

Quickstart installation (not for production use)

docker run -t --name db -p 8091-8094:8091-8094 -p 11210:11210 couchbase/server-sandbox:6.5.0

Couchbase Admin UI: http://localhost:8091

Login credentials: Administrator / password

N1QL

[https://query-tutorial.couchbase.com/tutorial/#index](https://query-tutorial.couchbase.com/tutorial/" \l "index)

|  |  |  |
| --- | --- | --- |
|  | SELECT 'Hello World' AS Greeting | {  "results": [  {  "Greeting": "Hello World"  }  ]  } |
|  | SELECT \*  FROM tutorial  WHERE fname = 'Ian' | {  "results": [  {  "tutorial": {  "type": "contact",  "title": "Mr.",  "fname": "Ian",  "lname": "Taylor",  "age": 56,  "email": "ian@gmail.com",  "children": [  {  "fname": "Abama",  "age": 17,  "gender": "m"  },  {  "fname": "Bebama",  "age": 21,  "gender": "m"  }  ],  "hobbies": [  "golf",  "surfing"  ],  "relation": "cousin"  }  }  ]  } |
| ‘.’ operator is used to refer to children, and the ‘[]’ is used to refer to an element in an array. | SELECT children[0].fname AS child\_name  FROM tutorial  WHERE fname='Dave' | {  "results": [  {  "child\_name": "Aiden"  }  ]  } |
|  | SELECT children[0].fname  FROM tutorial  WHERE fname='Dave' | {  "results": [  {  "fname": "Aiden"  }  ]  } |
| LIKE operator in the WHERE clause. | SELECT fname, email  FROM tutorial  WHERE email LIKE '%@yahoo.com' | {  "results": [  {  "email": "harry@yahoo.com",  "fname": "Harry"  }  ]  }  % is a wildcard that matches zero or more characters, and \_ is a wildcard that matches exactly one character. |
|  |  |  |
|  |  |  |
| Groupby  orderby | SELECT count(\*), state FROM customer WHERE customer.ccInfo.cardType="discover" GROUP BY customer.state ORDER BY customer.state LIMIT 5 OFFSET 5 | This query counts the number of customers per state who have a Discover credit card. The result set is grouped and ordered by state. Output is limited to 5 documents, after skipping the rst 5. |
| Normalized, rounded, and truncated | SELECT AVG(reviews.rating) / 5 AS normalizedRating, ROUND((AVG(reviews.rating) / 5), 2) AS roundedRating, TRUNC((AVG(reviews.rating) / 5), 3) AS truncRating FROM reviews AS reviews WHERE reviews.customerId = "customer62" | "results": [ { "normalizedRating": 0.65, "roundedRating": 0.65, "truncRating": 0.65 } ] |
| aggregation functions like SUM, MIN, and MAX |  |  |
| STRING CONCATENATION AND MATCHING | SELECT firstName || " " || lastName AS fullName FROM customer WHERE emailAddress LIKE "%.biz" | "results": [ { "fullName": "Joyce Murazik" }, ... ] |
| N1QL also provides string functions such as LOWER, UPPER, SUBSTR, and LENGTH |  |  |
| DISTINCT  COUNT | SELECT COUNT( DISTINCT customerId ) FROM purchases |  |
| NULL VALUES | SELECT fname, children FROM tutorial WHERE children IS NULL |  |
| IS MISSING | SELECT fname, children FROM tutorial WHERE children IS MISSING |  |
| INDEXES | CREATE INDEX idx ON `customer`(`emailAddress`) | N1QL uses indexes to perform queries. You can create primary indexes and global secondary indexes. |
| EXPLAIN |  | EXPLAIN shows how a statement will operate |
| ARRAYS AND OBJECTS | { "address" : { "city": "Toronto"}, "revision": [2014] } | dot “.” operator to access elds nested inside other objects  bracket [index] to access elements inside an array. |
| ARRAY functions |  | ARRAY\_LENGTH()  ARRAY\_PREPEND(,)  ARRAY\_APPEND(,)  ARRAY\_CONCAT(,) |
| COLLECTION EXPRESSIONS | SELECT \* FROM purchases WHERE ANY item IN purchases.lineItems SATISFIES item.count >= 5 END | A collection in N1QL is an array-valued subpath or expression. Collection predicates allow you to test a boolean condition over the elements of a collection. The ANY operator allows you to search through an array, returning TRUE when at least one match is found. With the EVERY operator, every single element needs to match. To search for purchase orders with a particular item purchased 5 times or more: |
| EVERY |  |  |
| ARRAY and FIRST | SELECT ARRAY item.product FOR item IN purchases.lineItems END AS product\_ids FROM purchases | To map and lter elements of a collection, you can use the ARRAY and FIRST operators.  Changing ARRAY to FIRST will produce the rst product in each purchase order. |
| JOIN, NEST, and UNNEST | SELECT c, pr FROM purchases pu JOIN customer c ON KEYS pu.customerId NEST product pr ON KEYS ARRAY li.product FOR li IN pu.lineItems END WHERE pu.customerId = "customer1"  (To assemble a complete list of products purchased by a customer) | A JOIN in N1QL is similar to SQL; a single result is produced for each matching left and right-hand input. NEST produces a single result for each left-hand input, while the right-hand input is collected and nested into a single array-valued eld in the result. |
| UNREST |  | The UNNEST clause allows you to take contents of a nested array and join them with the parent object. |
|  | SELECT p FROM product p UNNEST p.categories AS category WHERE category= "Appliances" | To list products belonging to a particular category: |