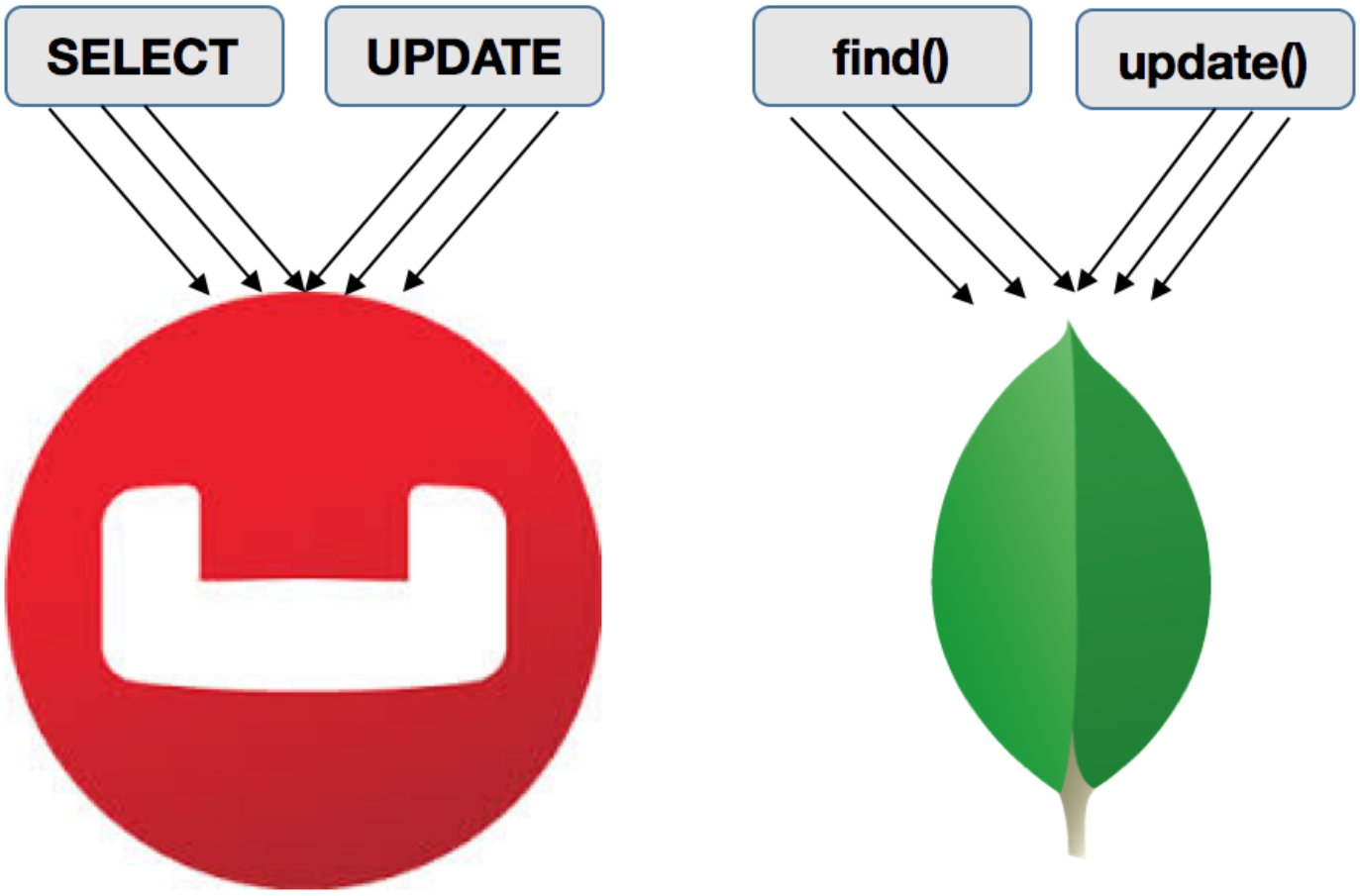
The Couchbase Blog
Couchbase, the NoSQL Database

Concurrency Behavior: MongoDB vs. Couchbase

Keshav Murthy on November 30, 2016



Next →
Austin User Group Visits – O…

David Glasser of Meteor wrote a blog on an MongoDB query missing matching documents issue he ran into. It is straightforward to reproduce the issue on both MongoDB MMAPv1 and MongoDB WiredTiger engine. Here are his conclusions from his article (emphasis is mine)

Long story short…

- This issue doesn't affect queries that don't use an index, such as queries that just look up a document by ID.
- It doesn't affect queries which explicitly do a single value equality match on all fields used in the index key.
- It doesn't affect queries that use indexes whose fields are never modified after the document is originally inserted.
- But any other kind of MongoDB query can fail to include all the matching documents!

Here's another way to look at it. In MongoDB, if the query can retrieve two documents using a secondary index (index on something other than _id) when concurrent operations are going on, the results could be wrong. This is a common scenario in many database applications.

Here is the test:

1. Create a Container: Bucket, table, or collection.
2. Load the data with a small dataset, say 300K documents.
3. Create an index on the field you want to filter on (predicates).
4. In one session, update the indexed field in one session and query on the other.

MongoDB Testing

Next →
Austin User Group Visits – O…

1. Install MongoDB 3.2.

2. Bring up mongod with either MMAPv1 or WiredTiger.

3. Load data using tpcc.py

4. python tpcc.py –warehouses 1 –no-execute mongodb

5. Get the count

```
> use tpcc
> db.ORDER_LINE.find().count();
299890
```

6. db.ORDER_LINE.ensureIndex({state:1});

MongoDB Experiment 1: Update to higher value

Setup the state field with the value aaaaaa and then concurrently update this value to zzzzzz and query for total number of documents with the two values ['aaaaaa','zzzzzz'] matching the field. When the value of the indexed field moves from lower (aaaaaa) to higher (zzzzzz) value, these entries are moving from one side of the B-tree to the other. Now, we're trying to see if the scan returns duplicate value, translated to higher count() value.

```
> db.ORDER_LINE.update({OL_DIST_INFO:{$gt:””}}, {$set: {state:”aaaaaa”}},
{multi:true});

WriteResult({ “nMatched” : 299890, “nUpserted” : 0, “nModified” : 299890 })
> db.ORDER_LINE.find({state:{$in:['aaaaaa','zzzzzz']}}).count();
299890


> db.ORDER_LINE.find({state:{$in:['aaaaaa','zzzzzz']}}).explain();
{
    “queryPlanner” : {
        “plannerVersion” : 1,
        “namespace” : “tpcc.ORDER_LINE”,
        “indexFilterSet” : false
                “$in” : [
                    “aaaaaa”,
```

Next →
Austin User Group Visits – O…

```
                    "zzzzzz"
                ]
            }
        },
        "winningPlan" : {
            "stage" : "FETCH",
            "inputStage" : {
                "stage" : "IXSCAN",
                "keyPattern" : {
                    "state" : 1
                },
                "indexName" : "state_1",
                "isMultiKey" : false,
                "direction" : "forward",
                "indexBounds" : {
                    "state" : [
                        "["aaaaaa", "aaaaaa"]",
                        "["zzzzzz", "zzzzzz"]"
                    ]
                }
            }
        },
        "rejectedPlans" : [ ]
    },
    "serverInfo" : {
        "host" : "Keshavs-MacBook-Pro.local",
        "port" : 27017,
        "version" : "3.0.2",
        "gitVersion" : "6201872043ecbbc0a4cc169b5482dcf385fc464f"
    },
    "ok" : 1
}
```

1. Update statement 1: Update all documents to set state = "zzzzzz"

```
db.ORDER_LINE.update({OL_DIST_INFO:{$gt:""}},
                {$set: {state: "zzzzzz"}}, {multi:true});
```

2. Update statement 2: Update all documents to set state = "aaaaaa"

Next →
Austin User Group Visits – O…

3. Count statement: Count documents:(state in [“aaaaaa”, “zzzzzz”])

```
db.ORDER_LINE.find({state:{$in:['aaaaaa','zzzzzz']}}).count();
```

Time	Session 1: Issue Update Statement1 (update state = “zzzzzz”)	Session 2: Issue Count Statement continuously.
T0	Update Statement starts	Count = 299890
T1	Update Statement Continues	Count = 312736
T2	Update Statement Continues	Count = 312730
T3	Update Statement Continues	Count = 312778
T4	Update Statement Continues	Count = 312656
T4	Update Statement Continues	Count = 313514
T4	Update Statement Continues	Count = 303116
T4	Update Statement Done	Count = 299890

Result: In this scenario, the index does double counting of many documents and reports more than it actually has.

Cause: Data in the leaf level of B-Tree is sorted. As the update B-Tree gets updated from aaaaaa to zzzzzz, the keys in the lower end are moved to the upper end. The concurrent scans are unaware of this move. MongoDB does not implement

Next →
Austin User Group Visits – O…

more. It just depends on the concurrent operations.

MongoDB Experiment 2: Update to lower value

Let's do the reverse operation to update the data from 'zzzzzz' to 'aaaaaa'. In this case, the index entries are moving from a higher value to a lower value, thus causing the scan to miss some of the qualified documents, shown to be undercounting.

Time	Session 1: Issue Update Statement2 (update state = "aaaaaa")	Session 2: Issue Count Statement continuously.
T0	Update Statement starts	Count = 299890
T1	Update Statement Continues	Count = 299728
T2	Update Statement Continues	Count = 299750
T3	Update Statement Continues	Count = 299780
T4	Update Statement Continues	Count = 299761
T4	Update Statement Continues	Count = 299777
T4	Update Statement Continues	Count = 299815
T4	Update Statement Done	Count = 299890

Result: In this scenario, the index misses many documents and reports fewer documents than it actually has.

Next →
Austin User Group Visits – O…

modified to aaaaaa, items go from the higher to the lower end of the B-Tree. Again, since there is no stability in scans, it would miss the keys that moved

from the higher end to the lower end.

MongoDB Experiment 3: Concurrent UPDATES

Two sessions update the indexed field concurrently and continuously. In this case, based on the the prior observations, each of the sessions run into both overcount and undercount issue. The nModified result varies because MongoDB reports only updates that changed the value.

But the total number of modified documents is never more than 299980. So, MongoDB does avoid updating the same document twice, thus handling the classic halloween problem. Because they don't have a stable scan, I presume they handle this by maintaining lists of objectIDs updated during this multi-update statement and avoiding the update if the same object comes up as a qualified document.

SESSION 1

```
>  db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"aaaaaa"}}, {multi:true});
WriteResult({ "nMatched" : 299890, "nUpserted" : 0, "nModified" : 299890 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"zzzzzz"}}, {multi:true});
WriteResult({ "nMatched" : 303648, "nUpserted" : 0, "nModified" : 12026 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"aaaaaa"}}, {multi:true});
WriteResult({ "nMatched" : 194732, "nUpserted" : 0, "nModified" : 138784 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"zzzzzz"}}, {multi:true});
WriteResult({ "nMatched" : 334134, "nUpserted" : 0, "nModified" : 153625 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"aaaaaa"}}, {multi:true});
WriteResult({ "nMatched" : 184379, "nUpserted" : 0, "nModified" : 146318 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"zzzzzz"}}, {multi:true});
WriteResult({ "nMatched" : 335613, "nUpserted" : 0, "nModified" : 153403 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"aaaaaa"}}, {multi:true});
WriteResult({ "nMatched" : 183559, "nUpserted" : 0, "nModified" : 146026 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"zzzzzz"}}, {multi:true});
WriteResult({ "nMatched" : 335238, "nUpserted" : 0, "nModified" : 149337 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"aaaaaa"}}, {multi:true});
WriteResult({ "nMatched" : 187815, "nUpserted" : 0, "nModified" : 150696 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"zzzzzz"}}, {multi:true});
```

Next →
Austin User Group Visits – O…

```
WriteResult({ "nMatched" : 188774, "nUpserted" : 0, "nModified" : 153279 })
> db.ORDER_LINE.update({state:{$gt:""}}, {$set: {state:"zzzzzz"}}, {multi:true});
```

```
WriteResult({ “nMatched” : 334408, “nUpserted” : 0, “nModified” : 155970 })
> db.ORDER_LINE.update({state:{$gt:””}}, {$set: {state:”zzzzzz”}}, {multi:true});
WriteResult({ “nMatched” : 299890, “nUpserted” : 0, “nModified” : 0 })
>
```

SESSION 2:

```
> db.ORDER_LINE.update({state:{$gt:””}}, {$set: {state:”zzzzzz”}}, {multi:true});
WriteResult({ “nMatched” : 302715, “nUpserted” : 0, “nModified” : 287864 })
> db.ORDER_LINE.update({state:{$gt:””}}, {$set: {state:”aaaaaa”}}, {multi:true});
WriteResult({ “nMatched” : 195248, “nUpserted” : 0, “nModified” : 161106 })
> db.ORDER_LINE.update({state:{$gt:””}}, {$set: {state:”zzzzzz”}}, {multi:true});
WriteResult({ “nMatched” : 335526, “nUpserted” : 0, “nModified” : 146265 })
> db.ORDER_LINE.update({state:{$gt:””}}, {$set: {state:”aaaaaa”}}, {multi:true});
WriteResult({ “nMatched” : 190448, “nUpserted” : 0, “nModified” : 153572 })
> db.ORDER_LINE.update({state:{$gt:””}}, {$set: {state:”zzzzzz”}}, {multi:true});
WriteResult({ “nMatched” : 336734, “nUpserted” : 0, “nModified” : 146487 })
> db.ORDER_LINE.update({state:{$gt:””}}, {$set: {state:”aaaaaa”}}, {multi:true});
WriteResult({ “nMatched” : 189321, “nUpserted” : 0, “nModified” : 153864 })
> db.ORDER_LINE.update({state:{$gt:””}}, {$set: {state:”zzzzzz”}}, {multi:true});
WriteResult({ “nMatched” : 334793, “nUpserted” : 0, “nModified” : 150553 })
> db.ORDER_LINE.update({state:{$gt:””}}, {$set: {state:”aaaaaa”}}, {multi:true});
WriteResult({ “nMatched” : 186274, “nUpserted” : 0, “nModified” : 149194 })
> db.ORDER_LINE.update({state:{$gt:””}}, {$set: {state:”zzzzzz”}}, {multi:true});
WriteResult({ “nMatched” : 336576, “nUpserted” : 0, “nModified” : 145833 })
> db.ORDER_LINE.update({state:{$gt:””}}, {$set: {state:”aaaaaa”}}, {multi:true});
WriteResult({ “nMatched” : 183635, “nUpserted” : 0, “nModified” : 146611 })
> db.ORDER_LINE.update({state:{$gt:””}}, {$set: {state:”zzzzzz”}}, {multi:true});
WriteResult({ “nMatched” : 336904, “nUpserted” : 0, “nModified” : 143920 })
>
```

Couchbase Testing

1. Install Couchbase 4.5.

2. Load data using tpcc.py

3. python tpcc.py –warehouses 1 –no-execute n1ql

Next →
Austin User Group Visits – O…

```
> SELECT COUNT(*) FROM ORDER_LINE;
    300023
```

5. CREATE INDEX i1 ON ORDER_LINE(state);

6. UPDATE ORDER_LINE SET state = ‘aaaaaa’;

Couchbase Experiment 1: Update to higher value

Do the initial setup.
> UPDATE ORDER_LINE SET state = ‘aaaaaa’ WHERE OL_DIST_INFO > “”;

Verify the Count. state field(attribute) with values “aaaaaa” is 300023.

```
> select count(1) a_cnt FROM ORDER_LINE where state =  'aaaaaa'
 UNION ALL
 select count(1) z_cnt FROM ORDER_LINE where state =  'zzzzzz';
   “results”: [
       {
           “a_cnt”: 300023
       },
       {
           “z_cnt”: 0
       }
   ],
```

Let’s ensure the index scan happens on the query.

```
EXPLAIN SELECT COUNT(1) AS totalcnt
FROM ORDER_LINE
WHERE state = 'aaaaaa' or state = 'zzzzzz';
            “~children”: [
                  {
                      “#operator”: “DistinctScan”,
                      “scan”: {
                          “#operator”: “IndexScan”,
                          “covers”: [
                              “cover ((ORDER_LINE.state))”,
                          “index”: “i2”,
                          “index_id”: “665b11a6c36d4136”,
```

Next →
Austin User Group Visits – O…

```
                    "keyspace": "ORDER_LINE",
                    "namespace": "default",
                    "spans": [
                        {
                            "Range": {
                                "High": [
                                    ""aaaaaa""
                                ],
                                "Inclusion": 3,
                                "Low": [
                                    ""aaaaaa""
                                ]
                            }
                        },
                        {
                            "Range": {
                                "High": [
                                    ""zzzzzz""
                                ],
                                "Inclusion": 3,
                                "Low": [
                                    ""zzzzzz""
                                ]
                            }
                        }
                    ],
                    "using": "gsi"
                }
            },
```

We can also use UNION ALL of two separate predicates (state = 'aaaaaa') and (state = 'zzzzzz') to have the index count efficiently.

```
cbq> explain select count(1) a_cnt
FROM ORDER_LINE
where state =  'aaaaaa'
UNION ALL
select count(1) z_cnt
FROM ORDER_LINE
```

Next →
Austin User Group Visits – O…

```
{
   "requestID": "ef99e374-48f5-435c-8d54-63d1acb9ad22",
   "signature": "json",
```

```
"results": [
    {
        "plan": {
            "#operator": "UnionAll",
            "children": [
                {
                    "#operator": "Sequence",
                    "~children": [
                        {
                            "#operator": "IndexCountScan",
                            "covers": [
                                "cover ((ORDER_LINE.state))",
                                "cover ((meta(ORDER_LINE).id))"
                            ],
                            "index": "i2",
                            "index_id": "665b11a6c36d4136",
                            "keyspace": "ORDER_LINE",
                            "namespace": "default",
                            "spans": [
                                {
                                    "Range": {
                                        "High": [
                                            ""aaaaaa""
                                        ],
                                        "Inclusion": 3,
                                        "Low": [
                                            ""aaaaaa""
                                        ]
                                    }
                                }
                            ],
                            "using": "gsi"
                        },
                        {
                            "#operator": "IndexCountProject",
                            "result_terms": [
                                {
                                    "as": "a_cnt",
                                    "expr": "count(1)"
                                }
                            ]
                        },
                    ]
                },
                {
```


Next →
Austin User Group Visits – O…

```
                {
                    "#operator": "Sequence",
                    "~children": [
                        {
                            "#operator": "IndexCountScan",
                            "covers": [
                                "cover ((ORDER_LINE.state))",
                                "cover ((meta(ORDER_LINE).id))"
                            ],
                            "index": "i2",
                            "index_id": "665b11a6c36d4136",
                            "keyspace": "ORDER_LINE",
                            "namespace": "default",
                            "spans": [
                                {
                                    "Range": {
                                        "High": [
                                            ""zzzzzz""
                                        ],
                                        "Inclusion": 3,
                                        "Low": [
                                            ""zzzzzz""
                                        ]
                                    }
                                }
                            ],
                            "using": "gsi"
                        },
                        {
                            "#operator": "IndexCountProject",
                            "result_terms": [
                                {
                                    "as": "z_cnt",
                                    "expr": "count(1)"
                                }
                            ]
                        }
                    ]
                }
            ]
        },
```


Next →
Austin User Group Visits – O…


```
 'zzzzzz'"
      }
```

```
    ],
    "status": "success",
    "metrics": {
        "elapsedTime": "2.62144ms",
        "executionTime": "2.597189ms",
        "resultCount": 1,
        "resultSize": 3902
    }
}
```

Setup the state field with the value aaaaaa. Then update this value to zzzzzz and concurrently query for total number of documents with the either of the two values.

Session 1: Update state field to value zzzzzz

```
UPDATE ORDER_LINE SET state = 'zzzzzz' WHERE OL_DIST_INFO > "";
{    "mutationCount": 300023 }
```

Session 2: query the count of 'aaaaaa' and 'zzzzzz' from ORDER_LINE.

Time	Session 1: Issue Update Statement1 (update state = "zzzzzz")	a_cnt	z_cnt	Total
T0	Update Statement starts	300023	0	300023
T1	Update Statement Continues	288480	11543	300023
T2	Update	259157	40866	300023
T3	Update	197167	102856	300023

Next →
Austin User Group Visits – O…

	Statement Continues			
T4	Update Statement Continues	165449	134574	300023
T5	Update Statement Continues	135765	164258	300023
T6	Update Statement Continues	86584	213439	300023
T7	Update Statement Done	0	300023	300023

Result: Index updates happen as the data gets updated. As the values migrate from ‘aaaaaa’ to ‘zzzzzz’, they’re not double counted.

Couchbase indexes provide stable scans by snapshotting the index at regular frequency. While this is a considerable engineering effort, as we’ve seen from this issue, it provides stability of answers even under extreme concurrency situations.

The data that index scan retrieves will be from the point scan starts. Subsequent updates coming in concurrently, will not be returned. This provides another level of stability as well.

It’s important to note that stability of scans is provided for each index scan. Indices take snapshots every 200 milliseconds. When you have a long running query with multiple index scans on the same or multiple indices, the query could use multiple snapshots. So, different index scans in a long running query will each return different results. This is an use case we’ll be improving in a

Next →
Austin User Group Visits – O…
Couchbase Experiment 2: Update to lower value

Let’s do the reverse operation to update the data from ‘zzzzzz’ back to ‘aaaaaa’.

Session 1: Update state field to value aaaaaa

```
UPDATE ORDER_LINE SET state = ‘aaaaaa’ WHERE OL_DIST_INFO > “”;
{ “mutationCount”: 300023 }
```

Session 2: query the count of ‘aaaaaa’ and ‘zzzzzz’ from ORDER_LINE.

Time	Session 1: Issue Update Statement1 (update state = “aaaaaa”)	a_cnt	z_cnt	Total
T0	Update Statement starts	0	300023	300023
T1	Update Statement Continues	28486	271537	300023
T2	Update Statement Continues	87919	212104	300023
T3	Update Statement Continues	150630	149393	300023
T4	Update Statement Continues	272358	27665	300023
T5	Update	299737	286	300023
T6	Update Statement Done	0	300023	300023

Next →
Austin User Group Visits – O…

	Statement Done			

Couchbase Experiment 3:Concurrent UPDATES

Two sessions update the indexed field concurrently and continuously. The stable scans of the index alway returns the full list of qualified documents: 30023 and Couchbase updates all of the documents and reports the mutation count as 30023. An update is an update regardless of whether the old value is the same as the new value or not.

Session 1:

```
update ORDER_LINE set state = 'aaaaaa' where state > “”;
{     “mutationCount”: 300023 }
update ORDER_LINE set state =  ‘zzzzzz'where state > “”;
{     “mutationCount”: 300023 }
update ORDER_LINE set state = 'aaaaaa' where state > “”;
{     “mutationCount”: 300023 }
update ORDER_LINE set state =  ‘zzzzzz'where state > “”;
{     “mutationCount”: 300023 }
update ORDER_LINE set state = 'aaaaaa' where state > “”;
{     “mutationCount”: 300023 }
update ORDER_LINE set state =  ‘zzzzzz'where state > “”;
{     “mutationCount”: 300023 }
update ORDER_LINE set state = 'aaaaaa' where state > “”;
{     “mutationCount”: 300023 }
update ORDER_LINE set state =  ‘zzzzzz'where state > “”;
{     “mutationCount”: 300023 }
```

Session 2:

```
update ORDER_LINE set state =  ‘zzzzzz'where state > “”;
{     “mutationCount”: 300023 }
```

Next →
Austin User Group Visits – O...

```
update ORDER_LINE set state =  ‘zzzzzz'where state > “”;
{     “mutationCount”: 300023 }
update ORDER_LINE set state = 'aaaaaa' where state > “”;
```

```
{     “mutationCount”: 300023 }
update ORDER_LINE set state =  ‘zzzzzz'where state > “”;
{     “mutationCount”: 300023 }
update ORDER_LINE set state = 'aaaaaa' where state > “”;
{     “mutationCount”: 300023 }
update ORDER_LINE set state =  ‘zzzzzz'where state > “”;
{     “mutationCount”: 300023 }
```

Conclusions

MongoDB:

1. MongoDB queries will miss documents if there are concurrent updates that move the data from one portion of the index to another portion of the index (higher to lower).
2. MongoDB queries will return the same document multiple times if there are concurrent updates that move the data from one portion of the index to another portion of the index (lower to higher).
3. Concurrent multi updates on MongoDB will run into the same issue and will miss updating all the required documents, but they do avoid updating the same document twice in a single statement.
4. When developing applications that use MongoDB, you must design a data model so you select and update only one document for each query. Avoid multi-document queries in MongoDB since it will return incorrect results when there are concurrent updates.

Couchbase:

1. Couchbase returns the expected number of qualifying documents even when there are concurrent updates. Stable index scans in Couchbase provide the protection to the application that MongoDB does not.
2. Concurrent updates benefit from stable index scans and process all of the qualified documents when the application statement is issued.

Acknowledgements

Thanks to Sriram Melkote and Deepkaran Salooja from Couchbase Indexing team for their

Next →
Austin User Group Visits – O…

References

1. MongoDB Strong Consistency: https://www.mongodb.com/nosql-explained
2. MongoDB Concurrency: https://docs.mongodb.com/manual/faq/concurrency/
3. MongoDB queries don't always return all matching documents!: https://engineering.meteor.com/mongodb-queries-dont-always-return-all-matching-documents-654b6594a827#.s9y0yheuv
4. Couchbase: http://www.couchbase.com
5. N1QL: http://query.couchbase.com

Posted in: N1QL / Query

Tagged in: Concurrency, Consistency, couchbase, mongodb, performance

Posted by Keshav Murthy

Keshav Murthy is a Vice President at Couchbase R&D. Previously, he was at MapR, IBM, Informix, Sybase, with more than 20 years of experience in database design & development. He lead the SQL and NoSQL R&D team at IBM Informix. He has received two President's Club awards at Couchbase, two Outstanding Technical Achievement Awards at IBM. Keshav has a bachelors degree in Computer Science and Engineering from the University of Mysore, India, holds eight US patents.

All Posts Website

Leave a reply

Next →
Austin User Group Visits – O…

Comment