Course code : **CSE3009**
Course title : **No SQL Data Bases**
Module : **4**
Topic : **5**

Consistency Implementation and Capped Collection

Objectives

This session will give the knowledge about

- Consistency Implementation
- Distributed consistency
- Eventual Consistency
- Capped Collection

Consistency Implementation

It's perfectly acceptable in some applications to have a slight lag in the time.

- Facebook posts don't appear instantly to all users.
- You can also see on Twitter that someone new is following you before the total number of followers is updated.

However, the same isn't true in situations such as in the following:

- Primary trading systems for billion dollar transactions
- Emergency medical information in an emergency room
- Target tracking information in a battle group headquarters

Consistency Implementation

Not all NoSQL databases support full ACID guarantees, unlike their relational database management systems counterparts.

It’s not that you’re either consistent or inconsistent. Instead, there’s a range of consistency levels.

Some products support just one level; others allow you to select from a range of levels for each database operation.

Consistency in MongoDB

By default - in a single-server deployment - a MongoDB database provides strict single-document consistency.

When a MongoDB document is being modified, it is locked against both reads and writes by other sessions.

However, when MongoDB replica sets are implemented, it is possible to configure something closer to eventual consistency by allowing reads to complete against secondary servers that may contain out-of-date data.

MongoDB Locking

Consistency for individual documents is achieved in MongoDB by the use of locks.

Locks are used to ensure that two writes do not attempt to modify a document simultaneously, and also that a reader will not see an inconsistent view of the data.

Multi-version concurrency control (MVCC) algorithm can be used to allow readers to continue to read consistent versions of data concurrently with update activity. MVCC is widely used in relational databases because it avoids blocking.

MongoDB does not implement an MVCC system, and therefore readers are prevented from reading a document that is being updated.

MongoDB Locking

The granularity of MongoDB locks has changed during its history.

In versions prior to MongoDB 2.0, a single global lock serialized all write activity, blocking all concurrent readers and writers of any document across the server for the duration of any write.

Lock scope was increased to the database level in 2.2, and to the collection level in 2.8.

In the MongoDB 3.0 release, locking is applied at the document level, providing the collection is stored using the WiredTiger storage engine.

MongoDB Locking

When document-level locking is in effect, an update to a document will only block readers or writers who wish to access the same document.

Locking is a controversial topic in MongoDB; the original “global” lock and lack of MVCC compared poorly with the mechanisms familiar in relational databases.

Now that lock scope is limited to the document level, these concerns have been significantly reduced.

Replica Sets and Eventual Consistency

In the multi-server scenario - MongoDB does not provides strict consistency.

All reads are directed to the primary server, which will always have the latest version of a document.

However, we saw in the previous that we can configure the MongoDB read preference to allow reads from secondary servers, which might return stale data.

Eventually all secondary servers should receive all updates, so this behavior can loosely be described as “eventually consistent.”

Using eventual consistency

With eventual consistency, a write operation is successful on the server that receives it but all replicas of that data aren't updated at the same time. They are updated later based on system replication settings.

Some databases provide only eventual consistency (Couchbase), whereas others allow tuning of consistency on a per operation basis, depending on the settings of the originating client request (MongoDB, Microsoft DocumentDB).

Most social networks use tuning consistency model for new posts. This model gives you very fast write operations, because you don't have to wait for all replicas to be updated in order for the write operation to be complete.

Inconsistency tends to last only a few seconds while the replicas catch up.

Using ACID consistency

ACID consistency is the gold standard of consistency guarantees. An ACID-compliant database ensures that

- All data is safe in event of failure.
- Database updates always keep the database in a valid state (no conflicts).
- One set of operations doesn’t interfere with another set of operations.
- Once the save completes, reading data from any replica will always give the new “current” result.

Using ACID consistency

MarkLogic Server provides ACID transactions both on the server-side (when applying a set of changes in a single operation) and across several client requests in an application (when applying each change individually, then having a user select ‘apply’).

Microsoft’s DocumentDB provides ACID transactions only on the server-side, when executing a JavaScript stored procedure.

The MarkLogic Data Hub Platform integrates and curates your enterprise data to provide immediate business value. Running on a NoSQL foundation for speed and scale, it’s multi-model, elastic, transactional, secure, and built for the cloud.

Capped Collections

Capped collections are fixed-size collections that support high-throughput operations that insert and retrieve documents based on insertion order.

Capped collections work in a way similar to circular buffers: once a collection fills its allocated space, it makes room for new documents by overwriting the oldest documents in the collection.

As an alternative to capped collections, consider MongoDB's TTL (Time To Live) indexes.

TTL indexes are not compatible with capped collections.

Create a Capped Collection

When creating a capped collection you must specify the maximum size of the collection in bytes, which MongoDB will pre-allocate for the collection.

The size of the capped collection includes a small amount of space for internal overhead.

```
db.createCollection( "log", { capped: true, size: 100000 } )
```

If the size field is less than or equal to 4096, then the collection will have a cap of 4096 bytes. Otherwise, MongoDB will raise the provided size to make it an integer multiple of 256.

Create a Capped Collection

Additionally, you may also specify a maximum number of documents for the collection using the max field as in the following document:

```
db.createCollection("log", { capped : true, size : 5242880, max : 5000 } )
```

IMPORTANT

The size argument is always required, even when you specify max number of documents. MongoDB will remove older documents if a collection reaches the maximum size limit before it reaches the maximum document count.

Query a Capped Collection

If you perform a find() on a capped collection with no ordering specified, MongoDB guarantees that the ordering of results is the same as the insertion order.

To retrieve documents in reverse insertion order, issue find() along with the sort() method with the $natural parameter set to -1, as shown in the following example:

```
db.cappedCollection.find().sort( { $natural: -1 } )
```

Query a Capped Collection

Check if a Collection is Capped

Use the isCapped() method to determine if a collection is capped, as follows:

```
db.collection.isCapped()
```

Convert a Collection to Capped

You can convert a non-capped collection to a capped collection with the convertToCapped command:

```
db.runCommand({"convertToCapped": "mycoll", size: 100000});
```

The size parameter specifies the size of the capped collection in bytes.

Behavior

Insertion Order

Capped collections guarantee preservation of the insertion order. As a result, queries do not need an index to return documents in insertion order. Without this indexing overhead, capped collections can support higher insertion throughput.

Automatic Removal of Oldest Documents

To make room for new documents, capped collections automatically remove the oldest documents in the collection without requiring scripts or explicit remove operations.

Restrictions and Recommendations

Updates

- If you plan to update documents in a capped collection, create an index so that these update operations do not require a collection scan.

Document Size

- If an update or a replacement operation changes the document size, the operation will fail.

Document Deletion

- You cannot delete documents from a capped collection. To remove all documents from a collection, use the drop() method to drop the collection and recreate the capped collection.

Restrictions and Recommendations

Sharding

- You cannot shard a capped collection.

Query Efficiency

- Use natural ordering to retrieve the most recently inserted elements from the collection efficiently. This is (somewhat) analogous to tail on a log file.

Aggregation $out

- The aggregation pipeline stage $out cannot write results to a capped collection.

Transactions

- Starting in MongoDB 4.2, you cannot write to capped collections in transactions. Reads from capped collections are still supported in transactions.

Summary

This session will give the knowledge about

- Consistency Implementation
- Distributed consistency
- Eventual Consistency
- Capped Collection
- References:
  - https://docs.mongodb.com/manual/tutorial/
  - “Seven NoSQL Databases in a Week” Author: Aaron Ploetz Aaron