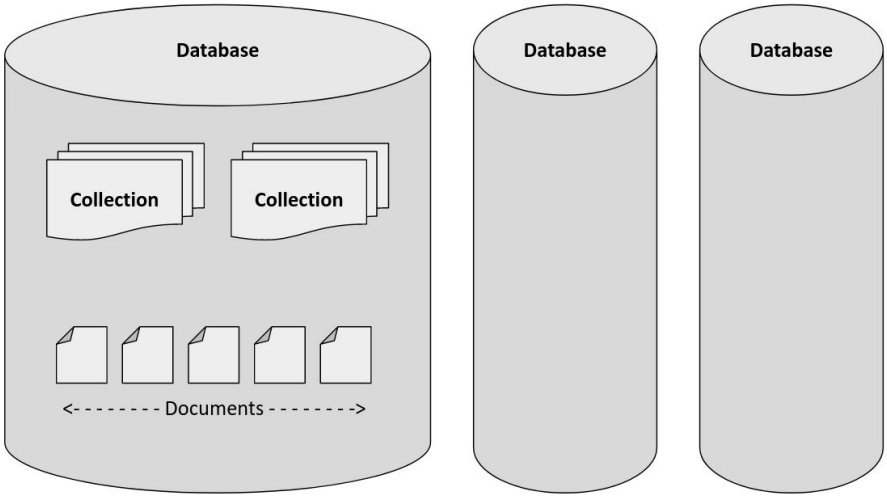
Concurrency control in MongoDB

Vishakh.S (B16CS038)

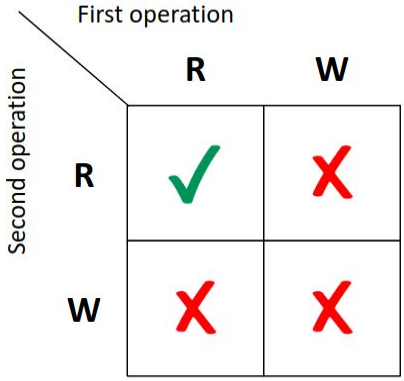
MongoDB has the following data hierarchy. Databases store collections of documents.



MongoDB permits multiple users to access and update data stored simultaneously. In order to guarantee consistency of data, it uses locks and other concurrency control measures. Besides providing a guarantee that the same piece of data is not modified by 2 clients simultaneously, MongoDB also ensures atomicity of such updates (operations should either proceed to completion or not affect the system at all).

In **MongoDB 2.0** and before, a single lock at the top level was used to guarantee consistency. This lock is called the **top lock** or global lock. This lock had 2 modes - namely **Read** (**R**) mode and **Write** (**W**) mode. Readers and writers take the lock into the R and W modes respectively.

This design allowed multiple readers in the system. Due to this, the read mode is also called **Shared** mode. Only one writer is allowed to update the system at any time. Hence, the write mode is also called **Exclusive** mode. If multiple readers are working on the system, every writer should queue up behind them. Similarly, if one writer is working on the system, every other operation should queue up behind it. The adjacent figure shows the compatibility of 2 operations.



The same lock was used for operations at every level of the data hierarchy. This severely limited concurrency since 2 operations updating 2 different databases or collections or documents could not be allowed to proceed simultaneously.

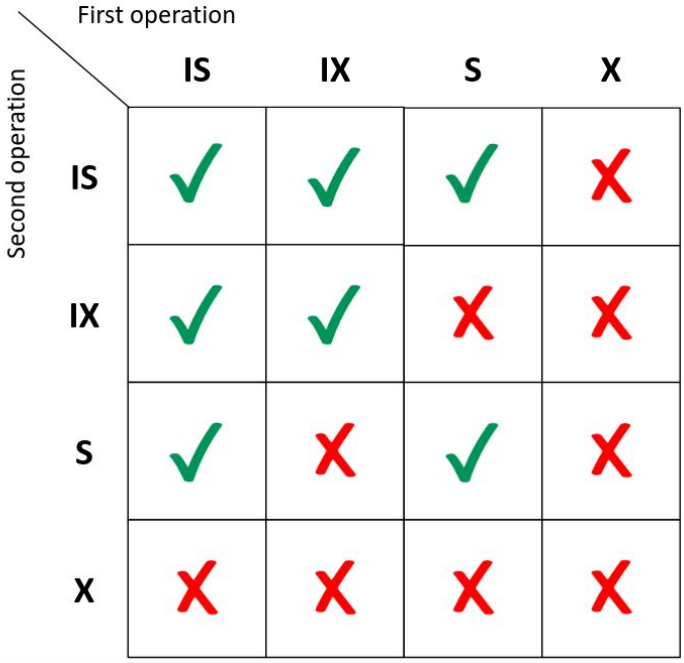
MongoDB 2.2 introduced **per database level** locks. This allowed 2 operations on 2 different databases to proceed in parallel. Within a single database, the operations still had to serialize with each other. This was implemented by preserving the top lock and adding **intents**. Intents specify an "intention" to access one or more children of an item. A key feature of intents is that they **don't conflict with each other.** When an intent is acquired at a particular level, you need to acquire an actual lock further down (at the database level). The lock applies to the item that it is acquired on and recursively on any item below it in the data hierarchy. Acquiring an intent does not carry any associated overhead.

There are four types of locking modes :

1. **S (shared)** – This mode allows concurrent reads to access the same resource. Used for read operations. It is also called read mode.
2. **X (exclusive)** – It is used for write operations. This mode allows only 1 writer to update the item on which the lock is acquired.
3. **IS (intent shared)** – This lock indicates an intent to read a resource.
4. **IX (intent exclusive)** – This lock indicates an intent to write a resource.

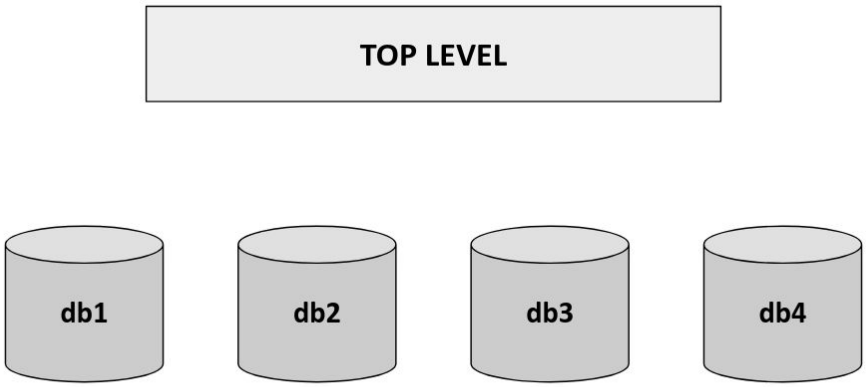
The table shown alongside shows the compatibility of these modes :

- A single database can be simultaneously locked in both intent shared and intent exclusive modes as intents are not actual locks and don't conflict.
- A shared mode is only compatible with other shared and intent shared modes. In case of X and IX, the second operation has to wait for the first one to proceed to completion.
- An exclusive mode cannot coexist with any other mode.



Examples :

The two examples that follow are for the scenario where there are 4 databases. (figure below)



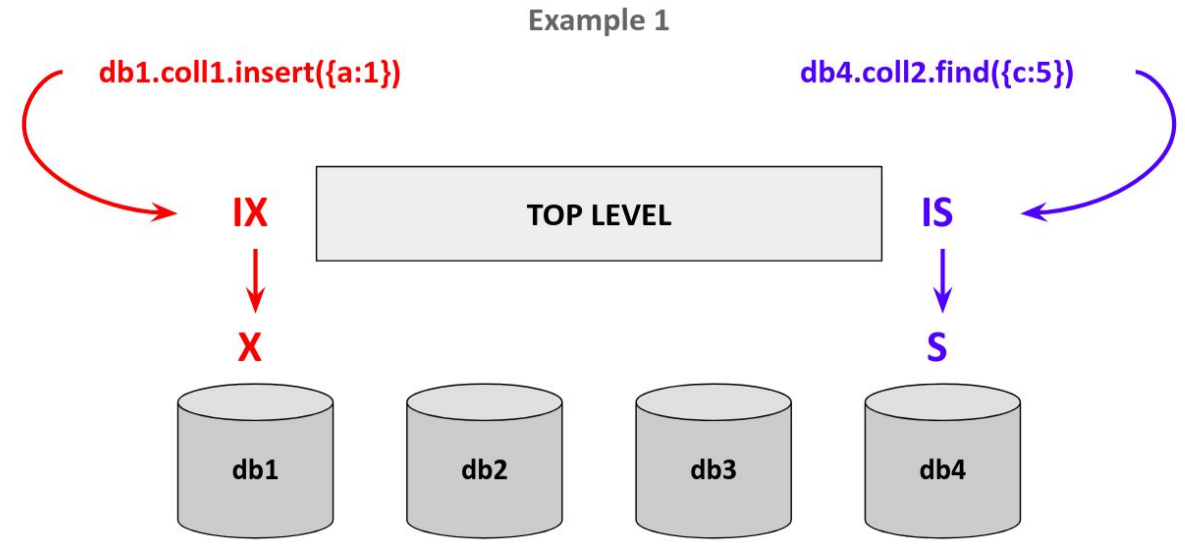
The status of the system and the lock modes can be checked with db.serverStatus() command. The following table shows the lock modes and their descriptions:

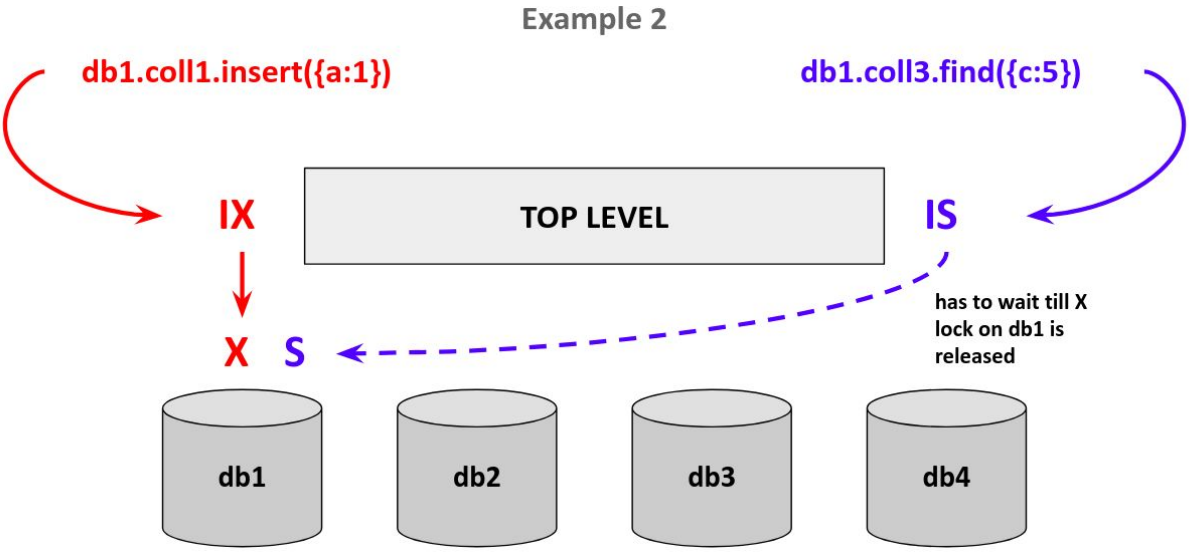
Lock mode	Lock	Description
R	S	Shared lock
W	X	Exclusive lock
r	IS	Intent shared lock
w	IX	Intent exclusive lock

1. **db1.coll1.insert({a:1})** and **db4.coll2.find({c:5})** come into the system.
   - **db1.coll1.insert({a:1})** takes an intent on the top level lock (**IX** mode).
   - **db1.coll1.insert({a:1})** proceeds to take an actual lock on db1 (**X** mode).
   - **db4.coll2.find({c:5})** tries to take an intent on the top level lock (**IS** mode).
   - It is granted as both **IS** and **IX** are compatible.
   - **db4.coll2.find({c:5})** proceeds to take an actual lock on db4 (**S** mode).

2. **db1.coll1.insert({a:1})** and **db1.coll3.find({c:5})** come into the system.
   - **db1.coll1.insert({a:1})** takes an intent on the top level lock (**IX** mode).
   - **db1.coll1.insert({a:1})** proceeds to take an actual lock on db1 (**X** mode).
   - **db1.coll3.find({c:5})** tries to take an intent on the top level lock (**IS** mode).
   - It is granted as both **IS** and **IX** are compatible.
   - **db1.coll3.find({c:5})** attempts to take an actual lock on db1 (**S** mode).
   - It is not granted as **S** and **X** are incompatible.
   - **db1.coll3.find({c:5})** has to wait till **db1.coll1.insert({a:1})** has completed.

The following 2 figures give an illustration of the above mentioned experiments.



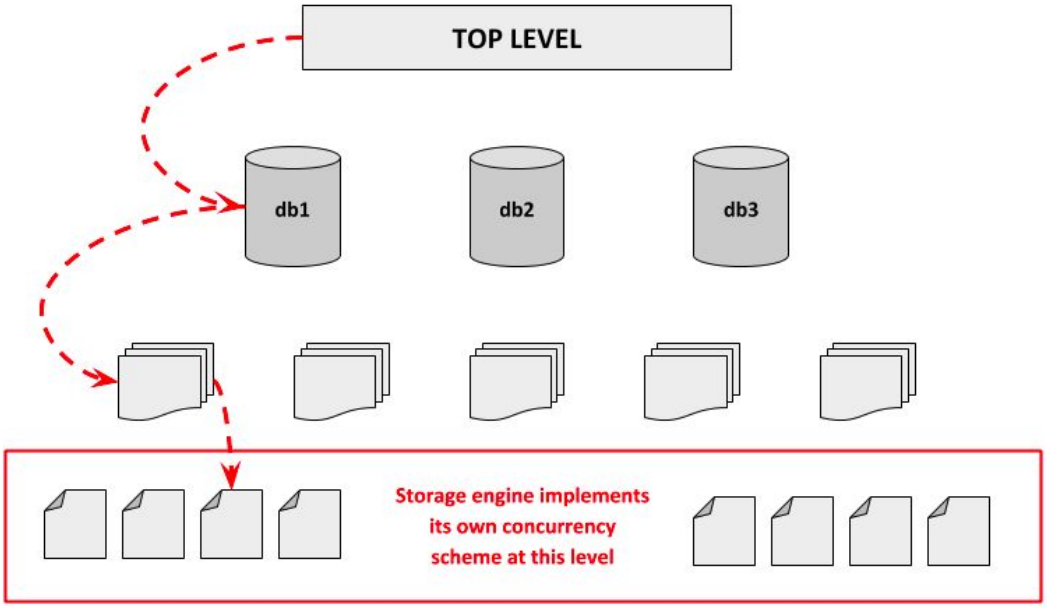


Example 2 makes it evident that further improvements were needed for proper concurrency. Operations (at least 1 of them being a write) to 2 different collections of the same database had to serialize with each other.

MongoDB 3.0 introduced collection and document level locking below the top and database level locks. The **multi-granularity locking mechanism** has now 4 levels of locking as described below:

1. **global or top level** – lock affects all the databases.
2. **database level** – lock affects only the database on which it is applied.
3. **collection level** – lock affects only the collection on which it is applied.
4. **document level** – lock affects only the document on which it is applied (allows each storage engine to implement its own concurrency control scheme).

Locks are acquired in a **hierarchical** fashion. When an application comes in, it cannot directly acquire a lock on the document it needs. Instead, it has to start from the top and acquire an intent (usually) on the global level. It has to acquire a lock on the database and then on the collection, before it finally acquires one on the document accessed. This hierarchical acquisition of locks is depicted below.



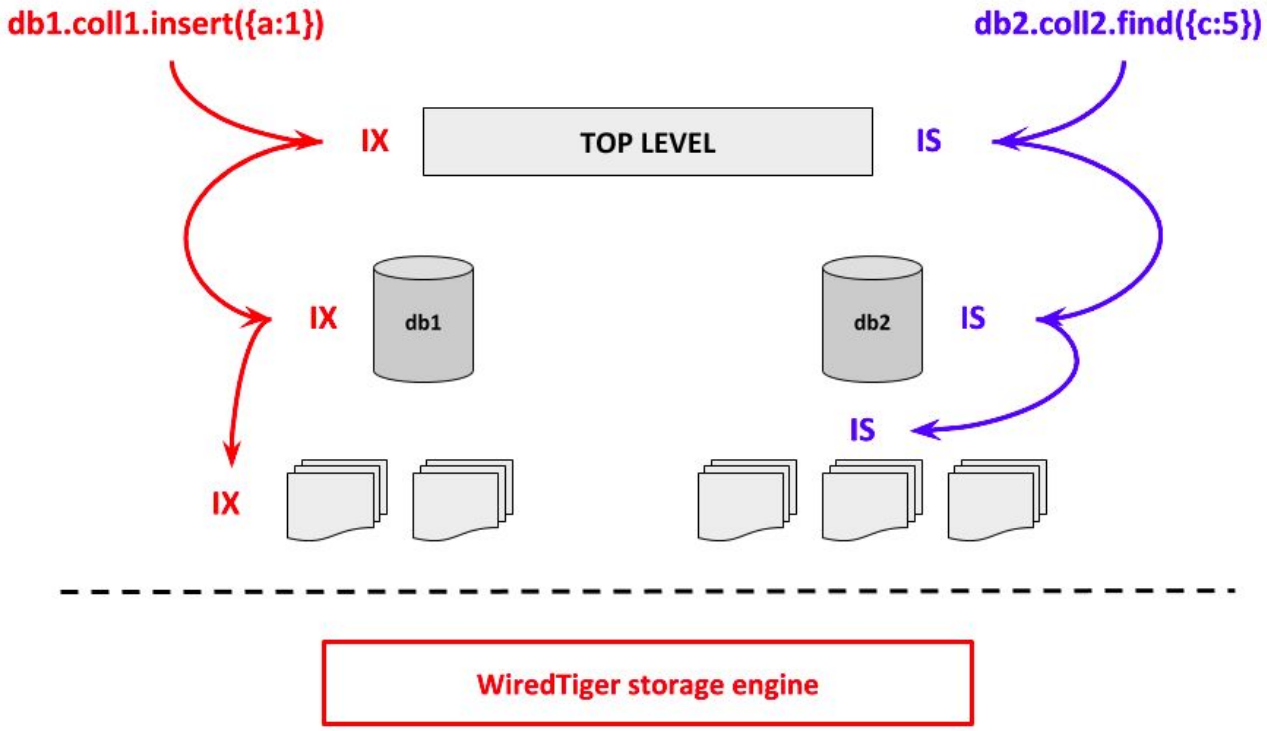
There are 2 main storage engines used by MongoDB - WiredTiger and MMAP V1. With **WiredTiger** storage engine, **intents are acquired at the global, database and collection level** and the document level concurrency is on WiredTiger. On the other hand, with MMAP V1, **intents are acquired at the global and database level**. An **actual lock is acquired at the collection level**. The

	WiredTiger	MMAP V1
Top level	IS or IX	IS or IX
db	IS or IX	IS or IX
Collection	IS or IX	S or X

document level concurrency is on MMAP V1.
The following examples are obtained with the default **WiredTiger** storage engine.

1. 2 databases are created. **Operations access 2 different databases**. **db1.coll1.insert({a:1})** and **db2.coll2.find({c:5})** come into the system.
   - **db1.coll1.insert({a:1})** takes an intent on the top level lock (**IX** mode).
   - **db1.coll1.insert({a:1})** proceeds to take an intent on db1 (**IX** mode).
   - **db1.coll1.insert({a:1})** proceeds to take an intent on coll1 of db1 (**IX** mode).
   - **db2.coll2.find({c:5})** tries to take an intent on the top level lock (**IS** mode).
   - It is granted as both **IS** and **IX** are compatible.
   - **db2.coll2.find({c:5})** proceeds to take an intent on db2 (**IS** mode).
   - **db2.coll2.find({c:5})** proceeds to take an intent on coll2 of db2 (**IS** mode).

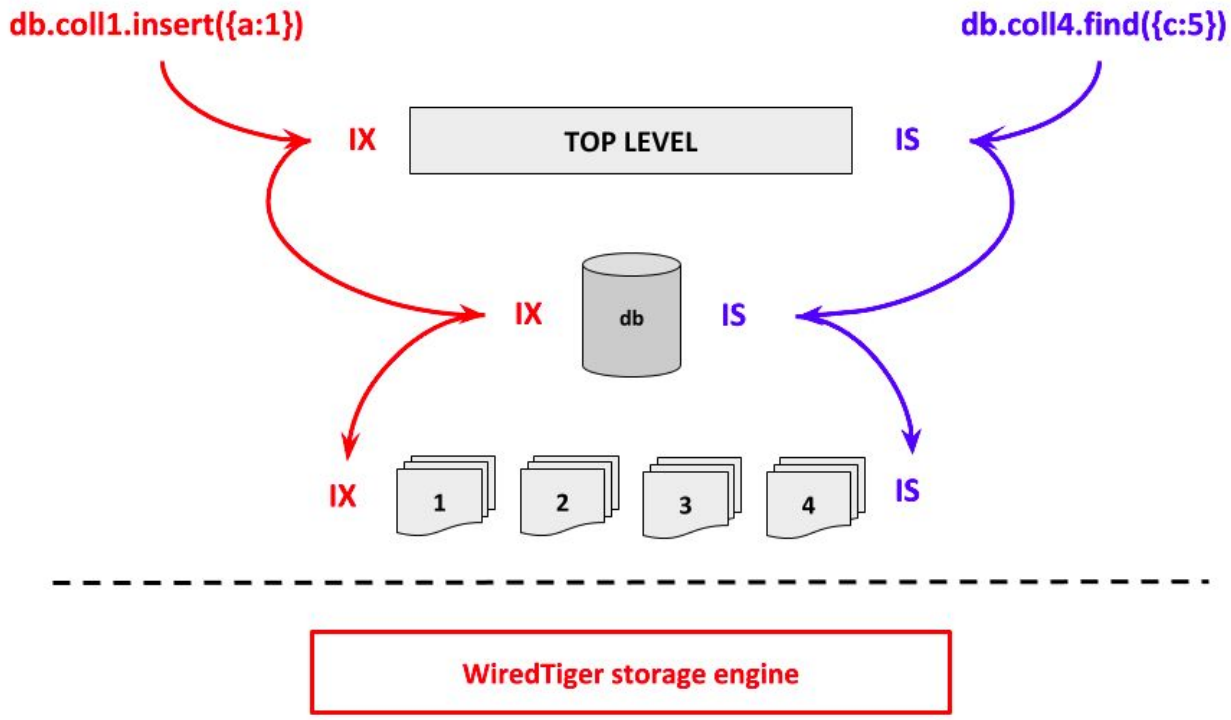
   Figure given below shows the acquisition of intents for this example.



2. 1 single database is created. **Operations access 2 different collections of the same database**. **db.coll1.insert({a:1})** and **db.coll4.find({c:5})** come into the system.
   - **db.coll1.insert({a:1})** takes an intent on the top level lock (**IX** mode).
   - **db.coll1.insert({a:1})** proceeds to take an intent on db (**IX** mode).
   - **db.coll1.insert({a:1})** proceeds to take an intent on coll1 of db (**IX** mode).
   - **db.coll4.find({c:5})** tries to take an intent on the top level lock (**IS** mode).

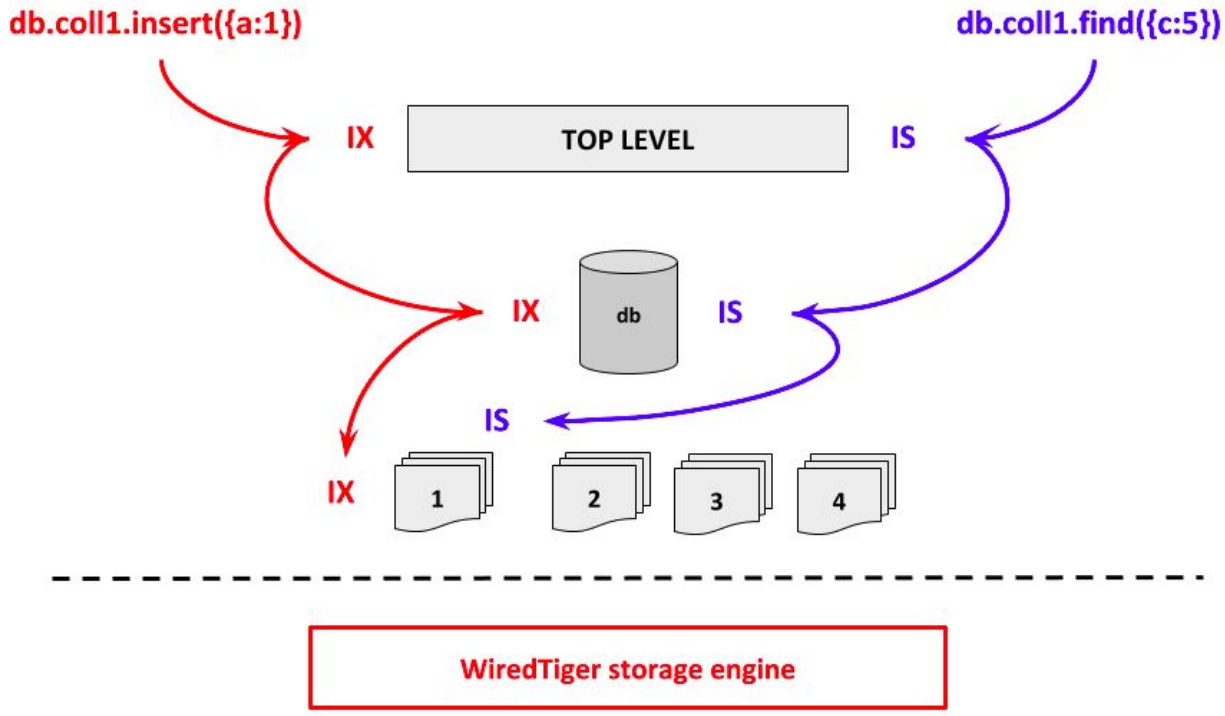
- It is granted as both **IS** and **IX** are compatible.
- **db.coll4.find({c:5})** attempts to take an intent on db (**IS** mode).
- It is granted as both **IS** and **IX** are compatible.
- **db.coll4.find({c:5})** proceeds to take an intent on coll4 of db (**IS** mode).

In this case, the problem of concurrent accesses to 2 different collections of the same database has been resolved. Figure given below shows the acquisition of intents for this example.



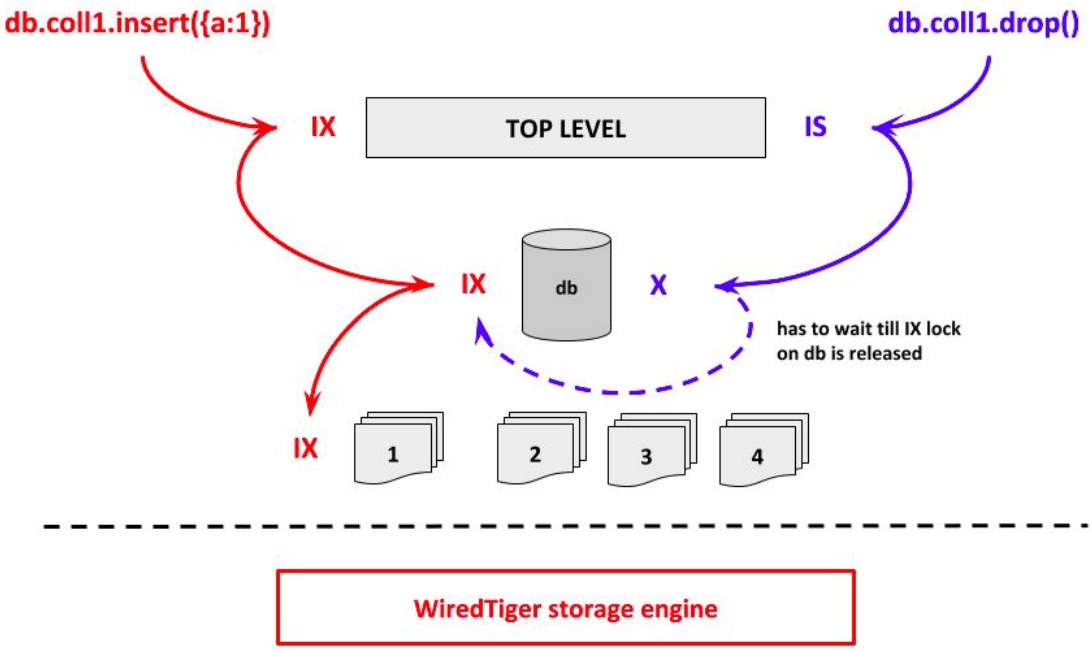
3. 1 single database is created. **Operations access the same collection**. **db.coll1.insert({a:1})** and **db.coll1.find({c:5})** come into the system.
   - **db.coll1.insert({a:1})** takes an intent on the top level lock (**IX** mode).
   - **db.coll1.insert({a:1})** proceeds to take an intent on db (**IX** mode).
   - **db.coll1.insert({a:1})** proceeds to take an intent on coll1 of db (**IX** mode).
   - **db.coll1.find({c:5})** tries to take an intent on the top level lock (**IS** mode).
   - It is granted as both **IS** and **IX** are compatible.
   - **db.coll1.find({c:5})** attempts to take an intent on db (**IS** mode).
   - It is granted as both **IS** and **IX** are compatible.
   - **db.coll1.find({c:5})** attempts to take an intent on coll1 of db (**IS** mode).
   - It is granted as both **IS** and **IX** are compatible.

In this case, the problem of concurrent accesses to 2 different documents of the same collection has been resolved. Figure given below shows the acquisition of intents for this example.



4. 1 single database is created. **db.coll1.insert({a:1})** and **db.coll1.drop()** come into the system.
   - **db.coll1.insert({a:1})** takes an intent on the top level lock (**IX** mode).
   - **db.coll1.insert({a:1})** proceeds to take an intent on db (**IX** mode).
   - **db.coll1.insert({a:1})** proceeds to take an intent on coll1 of db (**IX** mode).
   - **db.coll1.drop()** tries to take an intent on the top level lock (**IX** mode).
   - It is granted as both **IS** and **IX** are compatible.
   - **db.coll1.drop()** attempts to take an exclusive lock on db (**X** mode). This operation involves deleting a child (collection) of the database itself. So, it proceeds like in MongoDB 2.2, by acquiring an actual lock on the database whose collection is to be deleted.
   - It is not compatible with **IX** and has to wait till **db.coll1.insert({a:1})** is complete. Once the update to collection 1 of the database is complete, we drop the collection. This ensures consistency at all times during the process.

Figure given below shows the acquisition of intents for this example. Dotted line shows that the operation is waiting for any previous one to complete execution.



Observations and conclusion :

- MongoDB guarantees consistency of data and atomicity of operations during concurrent accesses by using locks and intents.
- Locks are used for reserving accesses to an item. Intents are not locks in the true sense of the term and hence don't conflict with each other.
- Concurrency and locking mechanisms at the top/ global level, database level and collection level are handled in the lock manager code. The storage engine is responsible for concurrency at the document level.
- A key observation is that if the conflict queue contains 6 concurrent operations requesting for IS, IS, X, IX, S and IS modes respectively; all the S and IS modes are granted. Once all the S and IS modes drain out, all writes are granted access.
- Thus, locks ensure that 2 accesses do not see an inconsistent piece of data or leave the system in an inconsistent state after the operation.

References:

1. Concurrency - https://docs.mongodb.com/manual/faq/concurrency/
2. https://www.mongodb.com/presentations/concurrency-control-in-mongodb-3-0
3. https://docs.mongodb.com/manual/reference/method/db.serverStatus/