Asymmetrical Locking for eXist-db

Adam Retter, Evolved Binary

February 5, 2018

After providing some background information on the problem domain, we examine various locking patterns, before justifying an asymmetrical approach. Finally we examine code patterns for managing asymmetrical locking and propose a design pattern for asymmetrical locking between Collection and persistent Document objects in eXist-db.

Background

In eXist-db there are two main types of container objects of concern, Collection and Document. Collections in eXist-db are hierarchical just like folders on a filesystem, a Collection links to child Documents or further sub-Collections.

1. A Collection object holds a map of Document (objects) by filename, and a set of sub-Collection names. It also has the properties:
    - ID
    - URI
    - Created date
    - Permissions
2. A Document object contains a number of functions for reading nodes of the document from storage. It holds an array of addresses for its child nodes. It also has the properties:
    - ID
    - Filename
    - Permissions
    - Number of Child nodes
    - Created date
    - Last modified date
    - Media Type
    - Page Count
    - User persistent lock id
    - DocType (optional)
    - WebDav lock token

eXist-db attempts to provide concurrent access to Collections and Documents. To try and ensure integrity of Collections and Documents during concurrent read and writes, eXist-db employs a shared-exclusive locking paradigm, where each Collection and Document object has an associated lock object with both read and write modes. In practice, a request for a write lock is only granted, if there are no read lock leases, and read locks cannot be granted whilst a write lock is held. In simpler terms, we can say that these Shared/Exclusive locks permit a single writer thread or multiple reader threads.

Concurrent Use Cases

There are various classes of concurrent CRUD (Create, Read, Update and Delete) operations that may occur, which involve different combinations of objects. If we assume that each CRUD operation is an independent transaction, and we wish to ensure the consistency guarantees of an ACID (Atomicity, Consistency, Isolation, Durability) isolation level of "serializable", the CRUD use cases and their associated locks modes are:

1. Adding a Collection: requires only a write mode lock on the parent Collection.
2. Reading or writing any property of a Collection (excluding its URI or Permissions): requires only a read or write mode lock on the Collection.
3. Renaming a Collection (modifying its URI): requires both a write mode lock on its parent Collection (to update the sub-Collection names set) and a write mode lock on the Collection.
4. Modifying the permissions of a Collection: requires a write mode lock on the Collection. However, during this update, neither further read/write access should be granted to child Documents or sub-Collections of the Collection recursively; as permissions in eXist-db are hierarchical, just like in Linux, and so execute access is required on the parent collection to read/write a document or sub-collection.
5. Deleting a Collection: requires write mode locks on itself, all child Documents and all sub-Collections (and their Documents recursively), and a write mode lock on its parent Collection (to update the sub-Collection names set). Deleting a Collection sub-tree might appear to be an expensive operation!
6. Adding a Document: requires both a write mode lock on the parent Collection, and a write mode lock for the Document which is to be created (so as to stop a concurrent user also creating the same document simultaneously).
7. Reading or writing any property of a Document (excluding its Filename): requires only a read or write mode lock on the Document.
8. Replacing the content of a Document: this only involves modifying its array of addresses for Child Nodes and Number of Child nodes property, as such it is the same as (7) in write mode.

9. Renaming a Document (modifying its Filename): requires both a write mode lock on its parent Collection (to update the Documents map) and a write mode lock on the Document.
10. Deleting a Document: requires both a write mode lock on the Document (to prevent concurrent access), and a write mode lock on the parent Collection (to update the Documents map).
11. Copying a Document: requires a read mode lock on the source Document, and the same locking modes as (6) or (8), for adding the copied document to the destination Collection, depending on whether it is a new document or replacing just the content of an existing document.
12. Moving a document: requires the same locking modes as (10) on the source Document and its parent Collection, and the same locking modes as (6) or (8) for adding the moved document to the destination Collection, depending on whether it is a new document or replacing just the content of an existing document.
13. Copying a Collection: requires read mode locks on the source Collection, its Documents and sub-Collections (recursively), and either:
    1. if the destination Collection does not exist, a write mode lock on the parent of the destination Collection, write mode locks on the destination Collection and any Documents and sub-Collections (recursively) which are to be created.
    2. if the destination Collection does exists, a write mode lock on the destination Collection and its Documents and sub-Collections (recursively) which are to be replaced or removed, and write mode locks on any new sub-Collections and their Documents (recursively) that will be created in the destination Collection.
14. Moving a Collection: requires write mode locks on the parent of the source Collection, the source Collection and its Documents and sub-Collections (recursively), and either:
    1. if the destination Collection does not exist, a write mode lock on the parent of the destination Collection, write mode locks on the destination Collection and any Documents and sub-Collections (recursively) which are to be created.
    2. if the destination Collection does exists, a write mode lock on the destination Collection and its Documents and sub-Collections (recursively) which are to be replaced or removed, and write mode locks on any new sub-Collections and their Documents (recursively) that will be created in the destination Collection.

Locking Patterns

To simplify this part of our discussion, we can for the time being ignore the hierarchical nature of Collections in eXist-db, and concern ourselves with access to a Document within a single Collection.

Our goal is *simply* to ensure safe concurrent access to a Document within a Collection. We want to achieve this without introducing deadlocks.

A deadlock can occur where there are multiple locks and concurrent access. For example, a scenario which would exhibit this would be, if `User A` holds the lock for `Resource A`, and wants to access `Resource B`, whilst `User B` holds the lock for `Resource B` and wants to access `Resource A`, neither user can make progress, they are "*deadlocked*" waiting for the lock that each other has.

It is well known in Computer Science, that one mechanism for deadlock avoidance is to ensure that locks are always acquired and released in the same order. So, for example to rewrite the above scenario with lock ordering, `User A` would take the lock on `Resource A`, `User B` would have to wait for the lock on `Resource A` (which held by `User A`), `User A` would take the lock on `Resource B` and perform some work, `User A` would then release the lock on `Resource B`, and then on `Resource A`, finally `User B` can acquire a lock on `Resource A`, then `Resource B` and perform their operations before releasing the locks.

A goal of any locking pattern that we create then, should be to ensure the consistent ordering of lock acquisition and release.

Hierarchical Symmetrical Locking

A locking scheme which immediately presents itself, is that of a hierarchy of locks, with one level for the Collection and one for the Document. The general form of such a locking scheme is:

1. Get and Lock Collection
2. Get the Document from the Collection
3. Lock the Document
4. Perform operation(s) on the Document
5. Unlock the Document
6. Unlock the Collection

For example, to write to the content of a Document within a Collection, the progress of this symmetrical scheme would look like:

1. Get and Lock Collection for READ
2. Get the Document from the Collection

3. Lock the Document for WRITE
4. Modify the Document content
5. Unlock the Document (from WRITE)
6. Unlock the Collection (from READ)

The symmetrical scheme expressed simply in Java might look like:

```
Collection collection = null;
try {
    col = getCollection("col1_name", READ_MODE);

    Document doc = null;
    try {
        doc = col.getDocument("doc1_name", WRITE_MODE);

        // Here we perform operations on the Document content

    } finally {
        if (doc != null) {
            doc.release(WRITE_MODE);
        }
    }
} finally {
    if (col != null) {
        col.release(READ_MODE);
    }
}
```

This scheme ensures that both the Collection lock is always taken before the Document lock and the Document lock is always released before the Collection lock, so that the total lock ordering is consistent.

One of the nice features of this approach is that we can also easily improve the naive Java code in various ways to enforce the locking order contract, and we can remove the onus on the developer to explicitly use our pattern. One such improved approach could be:

```
Collection collection = null;
try (final Collection col = getCollection("col1_name", READ_MODE);
     final Document doc = col.getDocument("doc1_name", WRITE_MODE)) {

        // Here we perform operations on the Document content

    }
}
```

The above code assumes that `Collection` and `Document` are refactored to implement `java.lang.AutoCloseable`, and that `AutoCloseable#close()` releases the locks for the modes that we acquired in the `try-with-resources` expression.

Unfortunately, whilst the above scheme fits our needs and we can express a design pattern for it nicely in code, it has a major drawback that the Collection remains locked whilst the operation on the Document is performed. There is no reason the Collection need remain locked whilst the Document is modified, and as operations on Documents may be resource intensive, this limits the concurrent operations that may be performed on Collections.

Hierarchical Asymmetrical Locking

We combine our previous Hierarchical Locking approach with Lock Interleaving on lock release between the levels of the hierarchy to try and release the Collection lock as early as possible. This subsequently reduces contention on the Collection lock and improves concurrency. Keeping in mind our previously described Use Cases, we need to be able to perform operations on locked Documents both before and after releasing the Collection lock. We must also preserve the goal of always acquiring and releasing locks in the same order. We believe that the following locking scheme provides such flexibility:

1. Get and Lock Collection
2. Perform any operation(s) that only require a Collection lock
3. Get the Document from the Collection
4. Lock the Document
5. Perform any operation(s) that require both a Collection and Document lock
6. Unlock the Collection
7. Perform any operation(s) that require only a Document lock
8. Unlock the Document

For example, to write to the content of a Document within a Collection, the progress of this asymmetrical scheme would look like:

1. Get and Lock Collection for READ
2. Get the Document from the Collection
3. Lock the Document for WRITE
4. Unlock the Collection (from READ)
5. Modify the Document content
6. Unlock the Document (from WRITE)

The asymmetrical scheme expressed simply in Java might look like:

```
Collection col = null;
Document doc = null;
try {
    col = getCollection("col1_name", READ_MODE);

    // Here we perform any operation(s) that only require the Collection
     ↪  lock

    try {
        doc = col.getDocument("doc1_name", WRITE_MODE);

        // Here we perform any operation(s) that require both the
         ↪  Collection lock and Document lock

    } finally {
        if (col != null) {
            col.release(READ_MODE);
        }
    }

    // Here we perform any operation(s) that only require the Document
     ↪  lock ...like modifying the Document content

} finally {
    if (doc != null) {
            doc.release(WRITE_MODE);
        }
    }
}
```

This scheme, like the symmetrical scheme, still ensures that the total locking order is consistent, but uses a modified ordering to interleave the release of the Document and Collection locks. It gives us greater flexibility and the ability to release the Collection lock sooner and therefore reduce lock contention.

Unfortunately, whilst this is a better scheme and our choice, it is much harder to express the pattern in Java in a more compact and safe manner when compared with that of the symmetrical scheme. In the next section, we examine variations on Java code patterns of the asymmetrical scheme.

Patterns for implementing Asymmetrical Locking in Java

Improving on the above Java expression of the asymmetrical scheme described above poses several challenges:

1. as its expression in the Java syntax itself is asymmetrical, it does not lend itself well to `RAII` like schemes such as Java 7's `try-with-resources`.
2. eXist-db's code base was started in Java in 2001, long before the current resurgence of Functional Programming to the popular mainstream, and as such uses Exceptions heavily for error conditions. This makes utilizing Java 8 lamdas for such resource (lock) management a difficult integration task.

With that in mind we present several approaches and discuss their merits and shortcomings. Many of these patterns were either directly influenced by or suggested by Stack Overflow users in response to the question: Best design pattern for managing asymmetrical resource use.

try-with-resources Lock Swapping

An initial approach we came up with for using `try-with-resources` when you want to asymmetrically manage object lifetimes, was to swap the Collection lock with the Document lock when the Document is retrieved from the Collection, this allows the locks to be released in the reverse order.

```
try (final ManagedRelease<Collection> mcol =
        new ManagedRelease<>(getCollection("col1_name", WRITE_MODE))) {

    // Here we perform any operation(s) that only require the Collection
     ↪ lock

    try (final ManagedRelease<Document> mdoc =
            mcol.withAsymetrical(mcol.resource.getDocument("doc1_name",
 ↪  WRITE_MODE))) {

        // Here we perform any operation(s) that require both the
         ↪ Collection lock and Document lock

    }  // NOTE: Collection lock is released here

    // Here we perform any operation(s) that only require the Document
     ↪ lock

}  // NOTE: Document lock is released here
```

This approach requires the introduction of a `ManagedRelease` class which has a mutable reference to the lock it needs to release. When `#withAsymetrical(AutoCloseable)` is called, the reference to the Collection lock object in `mcol` is replaced with a reference to the Document lock, and `mdoc` has its lock reference set to the Collection lock. The complete implementation for this pattern can be found at adamretter/asymmetrical-locking.

Advantages	Disadvantages
Manages the release of the locks for the developer. Introduces a small amount of changes.	Without the code *comments* inline it is not obvious which locks are released when. Likely to lead to developer confusion and mistakes. It is possible to accidentally use the Collection after it has been unlocked, because the reference is still present.

try-with-resources Idempotent Close

An approach suggested by `SergGr` on Stack Overflow would be to make the `#close()` (`#release()`) method of Collection Idempotent. Unfortunately, since our question was slightly underspecified, we can't directly do as `SergGr` suggested because we have the situation of nested locking, and unlocking more times than we should might cause unintended consequences. However, we can modify the suggested approach to better suit our needs:

```
try (final CollectionWithLock col = getCollection("col1_name",
 ↪  WRITE_MODE)) {

    // Here we perform any operation(s) that only require the Collection
     ↪  lock

    try (final Document doc = col.getDocument("doc1_name", WRITE_MODE)) {

        // Here we perform any operation(s) that require both the
         ↪  Collection lock and Document lock

        col.close();  // NOTE: Collection lock is released here

        // Here we perform any operation(s) that only require the Document
         ↪  lock

    }  // NOTE: Document lock is released here

}  // NOTE: nothing is released here (as already done above)
```

The class `CollectionWithLock` simply guards against us releasing the lock more than once:

```
class CollectionWithLock extends Collection {
    private boolean closed = false;

    @Override
```

```
    public void close() {
        if(!closed) {
            closed = true;
            super.close();
        }
    }
}
```

Advantages	Disadvantages
Partially manages the release of the locks for the developer. Introduces a small amount of changes, making integration with existing code simple.	By default does not release locks in the asymmetrical scheme order. If the developer forgets to call `col.close()` at the appropriate place the desired locking scheme is inconsistent, possibly leading to deadlocks. It is possible to accidentally use the Collection after it has been unlocked, because the reference is still present.

try-with-resources, Lambdas, and Abstraction

An approach suggested by `Douglas` on Stack Overflow would be to take our initial *try-with-resources Lock Swapping* approach and add an additional level of abstraction to make the lifecycle of the locks clear within enclosing `try-with-resources` expressions. We modified the suggested approach, so that method naming might better our domain, and expanded on the resource management properties:

```
try(final ManagedCollectionAndDocument mcoldoc1 =
 ↪  ManagedFactory.manage(brokerPool,
        "col1_name", READ_MODE,
        (broker, col) -> "doc1_name", WRITE_MODE)) {


    try(final ManagedWrapper<Collection> wcol1 =
     ↪  mcoldoc1.withCollection()) {

        // Here we perform any operation(s) that only require the
         ↪  Collection lock

    }

    // NOTE: Collection is not actually closed yet, but wcol1 is now out
     ↪  of scope
```

```java
    try(final ManagedWrapper<Tuple2<Collection, DocumentImpl>> wcoldoc1 =
     ↪  mcoldoc1.withCollectionAndDocument()) {

        // Here we perform any operation(s) that require both the
         ↪  Collection lock and Document lock

    }   // NOTE: Collection is closed here


    // NOTE: Document is not actually closed yet, but wcoldoc1 is now out
     ↪  of scope


    try(final ManagedWrapper<DocumentImpl> wdoc1 =
     ↪  mcoldoc1.withDocument()) {

        // Here we perform any operation(s) that only require the Document
         ↪  lock

    }   // NOTE: Document is closed here
}
```

Internally the acquisition of the Collection and Document are done lazily, so you need not call all methods in sequence, if you have no operations to perform at the various lock stages. For example, you need not even call `withCollection` or `withDocumentAndCollection`, and could just call `withDocument` if you just needed to perform an operation on the Document (after it has been safely retrieved from the Collection). The complete implementation for this pattern can be found at adamretter/asymmetrical-locking.

This was the most complicated to implement of our patterns, but offers quite some safety which can be encoded into the implementation by managing state.

Advantages	Disadvantages
Manages the release of the locks for the developer.	The use of lambdas and lazy initialisation leads to a complex implementation.
Introduces a small amount of changes (to existing code blocks).	The use of lambdas makes it hard to integrate with existing Exception based error handling.
It is impossible to use the Collection or Document after it has been unlocked, because the references are out of scope.	Developer must always call the functions in the correct order, i.e. cannot call `withDocument` and then `withCollection`.

Basic Lambda API

Another approach we investigated was using function passing as opposed to `try-with-resources`. In this approach we encapsulate our messy lock handling inside a function which we call `BasicLamdaApi.execute`, and we instead pass in (up to) three functions, one for each of the locking stages at which we wish to perform operations on the objects.

```
final Function<Collection, Long> colFun = collection -> {

    // Here we perform any operation(s) that only require the Collection
    ↪ lock

    return collection.getCreationTime();
};

final BiFunction<Collection, DocumentImpl, String> colDocFun =
↪ (collection, document) -> {

    // Here we perform any operation(s) that require both the Collection
    ↪ lock and Document lock

    return collection.getURI().append(document.getFileURI()).toString();
};

final Function<DocumentImpl, Long> docFun = document -> {

    // Here we perform any operation(s) that only require the Document
    ↪ lock

    final long now = System.currentTimeMillis();
    document.getMetadata().setLastModified(now);
    return now;
};


final Tuple3<Optional<Long>, Optional<String>, Optional<Long>> results =
↪ BasicLambdaApi.execute(
        brokerPool,
        "col1_name", READ_MODE,
        Optional.of(colFun),
        collection -> "doc1_name", WRITE_MODE,
        Optional.of(colDocFun),
        Optional.of(docFun)
);
```

As a user of such an API, it has a very nice lightweight and clean design to it. However, our ability to integrate easily with legacy code, which uses Exceptions for error notification, is limited by the use of functional programming constructs. For example, `Function` does not allow you to throw any Checked Exception within its lambda body. The complete implementation for this pattern can be found at adamretter/asymmetrical-locking.

Advantages	Disadvantages
Manages the release of the locks for the developer. It is impossible to use the Collection or Document after it has been unlocked, because the references are out of scope. Ordering is implicit, so the developer does not need to worry. Unlike the previous example, there is no lazy evaluation or need to maintain state across method calls, so the implementation is quite simple.	Would require a lot of changes to existing code. Integration with existing code that makes use of Checked Exceptions is tricky.

Fluent Lambda API

A final approach we considered was suggested by `Dean Xu` on Stack Overflow, we have modified it so that acquisition of resources is lazy, and that the method names better relate to our domain. It builds on the functional lambdas approach but uses a Fluent approach to construct an operation to execute. We recognize that we could also employ a builder pattern within an implementation of this Fluent API to enforce that only appropriate methods can be called within each state, so for example we could prevent `withCollection` being called after `withDocument`.

```
final Function<Collection, Long> colFun = collection -> {

    // Here we perform any operation(s) that only require the Collection
     ↪  lock

    return collection.getCreationTime();
};

final BiFunction<Collection, DocumentImpl, String> colDocFun =
 ↪  (collection, document) -> {
```

```java
    // Here we perform any operation(s) that require both the Collection
     ↪  lock and Document lock

    return collection.getURI().append(document.getFileURI()).toString();
};

final Function<DocumentImpl, Long> docFun = document -> {

    // Here we perform any operation(s) that only require the Document
     ↪  lock

    final long now = System.currentTimeMillis();
    document.getMetadata().setLastModified(now);
    return now;
};


final Tuple3<Long, String, Long> results = BasicFluentApi
    .withCollection("col1_name", READ_MODE)
    .execute(colFun)                                      // Lock
 ↪  Collection, and perform on Collection
    .withDocument(collection -> "doc1_name", WRITE_MODE)
    .execute(colDocFun)                                   // Lock
 ↪  Document, perform on Collection and Document
    .withoutCollection()                                  // Unlock
 ↪  Collection
    .execute(docFun)                                      // Perform on
 ↪  Document
    .doAll();                                             // Unlock any
 ↪  remaining (e.g. Document)
```

Advantages	Disadvantages
Manages the release of the locks for the developer.	Would require a lot of changes to existing code.
It is impossible to use the Collection or Document after it has been unlocked, because the references are out of scope.	Integration with existing code that makes use of Checked Exceptions is tricky.
Ordering is implicit, so the developer does not need to worry.	Complexity of implementation would be in state changes using a Builder pattern before `doAll` is called.

Advantages	Disadvantages
Builder API is powerful. 1. we can be precise about our return types (don't need `Optional`). 2. Could be extended and used to access more than one Collection or Document at a time. Possibly provides the cleanest approach.	

The complete implementation for this pattern can be found at adamretter/asymmetrical-locking, although due to the verbosity of the builder state machine implementation, it might be easier to study the unit tests.

Conclusion

Each of the proposed Patterns for implementing Asymmetrical Locking in Java has varying advantages and disadvantages which we have summarised.

Initially we favoured the Fluent Lambda API pattern due to the clean, clear, and concise code pattern that it enforces through its use. Its design explicitly prevents developers from being able to misuse the desired asymmetrical locking pattern. We implemented a version of this within eXist-db here. Unfortunately, we found that in practice, the main expected disadvantage of this pattern (namely integrating with the existing non-functional legacy code), proved too costly, and would have involved a huge amount of re-factoring of eXist-db's existing code base to achieve the integration; whilst not impossible, it would have greatly exceeded the resources available to this project.

Learning from our experiences of integration, we settled on the try-with-resources Idempotent Close pattern, because it required the least amount of changes to the existing code base for integration. To try and reduce confusion of the locking pattern in future we made careful use to annotate our explicit use of eagerly closing the Collection lock by calling `#close()`, e.g.:

```
// NOTE: early release of Collection lock inline with Asymmetrical Locking
 ↪ scheme
collection.close();
```

An example of this can be seen inline in the updated AbstractCompressFunction in eXist-db.

We hope that this Asymmetrical Locking Pattern is just one step towards further work in the future to improve and simplify the locking sub-systems in eXist-db. Certainly in the future, we would hope to see eXist-db adopt a more modern and functional style of coding, which would allow an implementation such as the Fluent Lambda API to be easily used in practice.