Part 2: Multigranularity

Dataphase

A working implementation of Part 1 is required for Part 2. If you have not yet finished Part 1, you should do so before continuing.

In this part, you will implement the middle layer (`LockContext`) and the declarative layer (in `LockUtil`). The `concurrency` directory contains a partial implementation of a lock context (`LockContext`), which you must complete in this part of the project.

Your Tasks

Task 3: LockContext

The `LockContext` class represents a single resource in the hierarchy; this is where all multigranularity operations (such as enforcing that you have the appropriate intent locks before acquiring or performing lock escalation) are implemented.

You will need to implement the following methods of `LockContext` :

- `acquire` : this method performs an acquire via the underlying `LockManager` after ensuring that all multigranularity constraints are met. For example, if the transaction has IS(database) and requests X(table), the appropriate exception must be thrown (see comments above method). If a transaction has a SIX lock, then it is redundant for the transaction to have an IS/S lock on any descendant resource. Therefore, in our implementation, we prohibit acquiring an IS/S lock if an ancestor has SIX, and consider this to be an invalid request.

- `release`: this method performs a release via the underlying `LockManager` after ensuring that all multigranularity constraints will still be met after release. For example, if the transaction has X(table) and attempts to release IX(database), the appropriate exception must be thrown (see comments above method).
- `promote`: this method performs a lock promotion via the underlying `LockManager` after ensuring that all multigranularity constraints are met. For example, if the transaction has IS(database) and requests a promotion from S(table) to X(table), the appropriate exception must be thrown (see comments above method). In the special case of promotion to SIX (from IS/IX/S), you should simultaneously release all descendant locks of type S/IS, since we disallow having IS/S locks on descendants when a SIX lock is held. You should also disallow promotion to a SIX lock if an ancestor has SIX, because this would be redundant.

Note: this does still allow for SIX locks to be held under a SIX lock, in the case of promoting an ancestor to SIX while a descendant holds SIX. This is redundant, but fixing it is both messy (have to swap all descendant SIX locks with IX locks) and pointless (you still hold a lock on the descendant anyways), so we just leave it as is.

- `escalate`: this method performs lock escalation up to the current level (see below for more details). Since interleaving of multiple `LockManager` calls by multiple transactions (running on different threads) is allowed, you must make sure to only use one mutating call to the `LockManager` and only request information about the current transaction from the `LockManager` (since information pertaining to any other transaction may change between the querying and the acquiring).
- `getExplicitLockType`: this method returns the type of the lock explicitly held at the current level. For example, if a transaction has X(db), `dbContext.getExplicitLockType(transaction)` should return X, but `tableContext.getExplicitLockType(transaction)` should return NL (no lock explicitly held).
- `getEffectiveLockType`: this method returns the type of the lock either implicitly or explicitly held at the current level. For example, if a transaction has X(db):
  - `dbContext.getEffectiveLockType(transaction)` should return X
  - `tableContext.getEffectiveLockType(transaction)` should *also* return X (since we implicitly have an X lock on every table due to explicitly having an X lock on the entire database).

Since an intent lock does *not* implicitly grant lock-acquiring privileges to lower levels, if a transaction only has SIX(database), `tableContext.getEffectiveLockType(transaction)` should return S (not SIX), since the transaction implicitly has S on table via the SIX lock, but not the IX part of the SIX lock (which is only available at the database level). It is possible for the explicit lock type to be one type, and the effective lock type to be a different lock type, specifically if an ancestor has a SIX lock.

The following helper methods may come in handy for this task: methods of `LockType` and `LockManager`, `ResourceName#parent` and `ResourceName#isDescendantOf`, `hasSIXAncestor` and `sisDescendants` which you will implement, `fromResourceName`.

Hierarchy

The `LockContext` objects all share a single underlying `LockManager` object. The `parentContext` method returns the parent of the current context (e.g. the lock context of the database is returned when `tableContext.parentContext()` is called), and the `childContext` method returns the child lock context with the name passed in (e.g. `tableContext.childContext(0L)` returns the context of page 0 of the table). There is exactly one `LockContext` for each resource: calling `childContext` with the same parameters multiple times returns the same object.

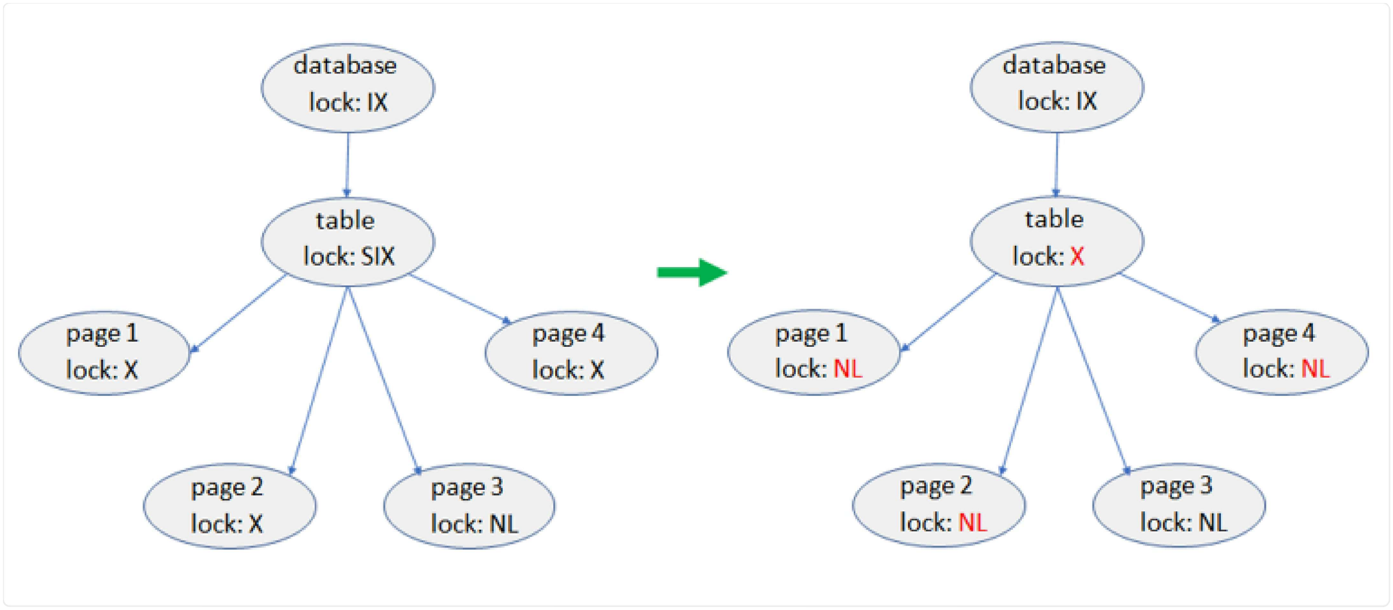
The provided code already initializes this tree of lock contexts for you. For performance reasons, however, we do not create lock contexts for every page of a table immediately. Instead, we create them as the corresponding `Page` objects are created.

Escalation

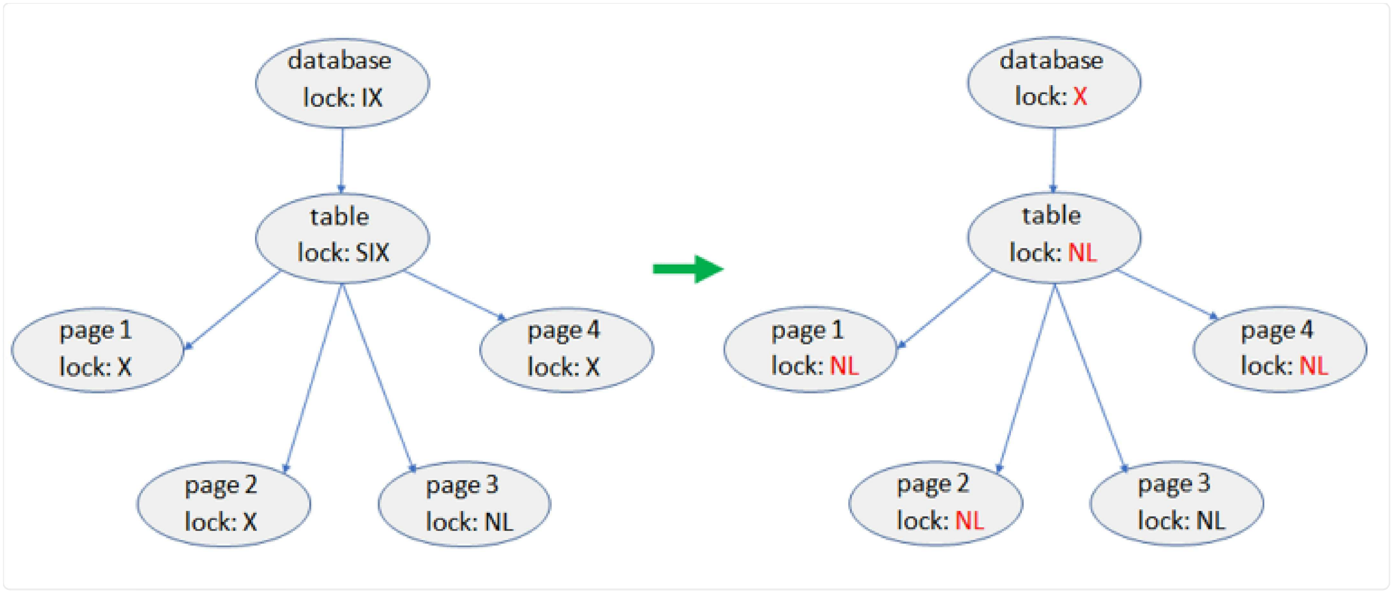
Lock escalation is the process of going from many fine locks (locks at lower levels in the hierarchy) to a single coarser lock (lock at a higher level). For example, we can escalate many page locks a transaction holds into a single lock at the table level.

We perform lock escalation through `LockContext#escalate`. A call to this method should be interpreted as a request to escalate all locks on descendants (these are the fine locks) into one lock on the context `escalate` was called with (the coarse lock). The fine locks may be any mix of intent and regular locks, but we limit the coarse lock to be either S or X.

For example, if we have the following locks: IX(database), SIX(table), X(page 1), X(page 2), X(page 4), and call `tableContext.escalate(transaction)`, we should replace the page-level locks with a single lock on the table that encompasses them:



Likewise, if we called `dbContext.escalate(transaction)`, we should replace the page-level locks and table-level locks with a single lock on the database that encompasses them:



Note that escalating to an X lock always "works" in this regard: having a coarse X lock definitely encompasses having a bunch of finer locks. However, this introduces other complications: if the transaction previously held only finer S locks, it would not have the IX locks required to hold an X lock, and escalating to an X reduces the amount of concurrency allowed unnecessarily. We therefore require that `escalate` only escalate to the least permissive lock type (between either S or X) that still encompasses the replaced finer locks (so if we only had IS/S locks, we should escalate to S, not X).

Also note that since we are only escalating to S or X, a transaction that only has IS(database) would escalate to S(database). Though a transaction that only has IS(database) technically has no locks at lower levels, the only point in keeping an intent lock at this level would be to acquire a normal lock at a lower level, and the point in escalating is to avoid having locks at a lower level. Therefore, we don't allow escalating to intent locks (IS/IX/SIX).

Task 4: LockUtil

The LockContext class enforces multigranularity constraints for us, but it's a bit cumbersome to use in our database: wherever we want to request some locks, we have to handle requesting the appropriate intent locks, etc.

To simplify integrating locking into our codebase (the second half of this part), we define the `ensureSufficientLockHeld` method. This method is used like a declarative statement. For example, let's say we have some code that reads an entire table. To add locking, we can do:

```
LockUtil.ensureSufficientLockHeld(tableContext, LockType.S);

// any code that reads the table here
```

After the `ensureSufficientLockHeld` line, we can assume that the current transaction (the transaction returned by `Transaction.getTransaction()`) has permission to read the resource represented by `tableContext`, as well as any children (all the pages).

We can call it several times in a row:

```
LockUtil.ensureSufficientLockHeld(tableContext, LockType.S);
LockUtil.ensureSufficientLockHeld(tableContext, LockType.S);

// any code that reads the table here
```

or write several statements in any order:

```
LockUtil.ensureSufficientLockHeld(pageContext, LockType.S);
LockUtil.ensureSufficientLockHeld(tableContext, LockType.S);
LockUtil.ensureSufficientLockHeld(pageContext, LockType.S);

// any code that reads the table here
```

and no errors should be thrown, and at the end of the calls, we should be able to read all of the table.

Note that the caller does not care exactly which locks the transaction actually has: if we gave the transaction an X lock on the database, the transaction would indeed have permission to read all of the table. But this doesn't allow for much concurrency (and actually enforces a serial schedule if used with 2PL), so we additionally stipulate that `ensureSufficientLockHeld` should grant as little additional permission as possible: if an S lock suffices, we should have the transaction acquire an S lock, not an X lock, but if the transaction already has an X lock, we should leave it alone (`ensureSufficientLockHeld` should never reduce the permissions a transaction has; it should always let the transaction do at least as much as it used to, before the call).

We suggest breaking up the logic of this method into two phases: ensuring that we have the appropriate locks on ancestors, and acquiring the lock on the resource. You will need to promote in some cases, and escalate in some cases (these cases are not mutually exclusive).

Task 5: Two-Phase Locking

At this point, you should have a working system to acquire and release locks on different resources in the database. In this task you'll add logic to acquire and release locks throughout the course of a transaction.

Tip: for all the files mentioned below, you can use IntelliJ's `Ctrl/Cmd + Shift + N` to search for the file quickly! If you still couldn't find the file, click into the link that will launch you into the Github repo.

Acquisition Phase

Reads and Writes: The simplest scheme for locking is to simply lock pages as we need them. As all reads and writes to pages are performed via the `Page.PageBuffer` class, it suffices to change only that. Modify the `get` and `put` methods of `Page.PageBuffer` to lock the page (and acquire locks up the hierarchy as needed) with the least permissive lock types possible.

Scans: If we know we'll be scanning multiple pages of a table, we're better off just getting a single lock on the table instance of many fine grained locks on the table's pages. Modify the ridIterator and recordIterator methods to acquire an appropriate lock on the table before doing a scan.

Write Optimization: When we modify a page, we'll almost always end up reading it first (acquiring IS/S locks) and then write back our updates to it afterwards (promoting to IX/X locks). If we know ahead of time that we're going to modify a page, we can skip the IS/S locks altogether by just acquiring IX/X locks to begin with. Modify the following methods to request the appropriate lock upfront:

- `PageDirectory#getPageWithSpace`
- `Table#updateRecord`
- `Table#deleteRecord`

Note: no more tests will pass after doing this, see the next section for why.

Release Phase

At this point, transactions should be acquiring lots of locks needed to do their queries, but no locks are ever released! We will be using Strict Two-Phase Locking in our database, which means that lock releases only happen when the transaction finishes, in the `cleanup` method.

Modify the `close` method of `Database.TransactionContextImpl` to release all locks the transaction acquired. You should only use `LockContext#release` and not `LockManager#release` - `LockManager` will not verify multigranularity constraints, but other transactions at the same time assume that these constraints are met, so you do want these constraints to be maintained. Note that you can't just release the locks in any order! Think about in what order you are allowed to release the locks.

You should pass the all the tests in `TestDatabaseDeadlockPrecheck` and `TestDatabase2PL` after implementing the acquisition and release phase.

Putting it all together

After implementing project 4, our database now supports locking, a fundamentally important functionality for database concurrency and isolation! Navigate to `CommandLineInterface.java` and uncomment line 41 and comment out line 38. Run the code to start our CLI. This should open a new panel in IntelliJ at the bottom. Click on this panel. We've provided 3 demo tables (Students, Courses, Enrollments). Recall from project 0 that we can run queries on this CLI. Let's try starting a transaction and querying a table by running:

```
BEGIN TRANSACTION;
```

and then

```
SELECT * FROM Students AS s INNER JOIN Enrollments AS e ON s.sid = e.sid;
```

We can display all the locks held by this transaction by running `\locks`.

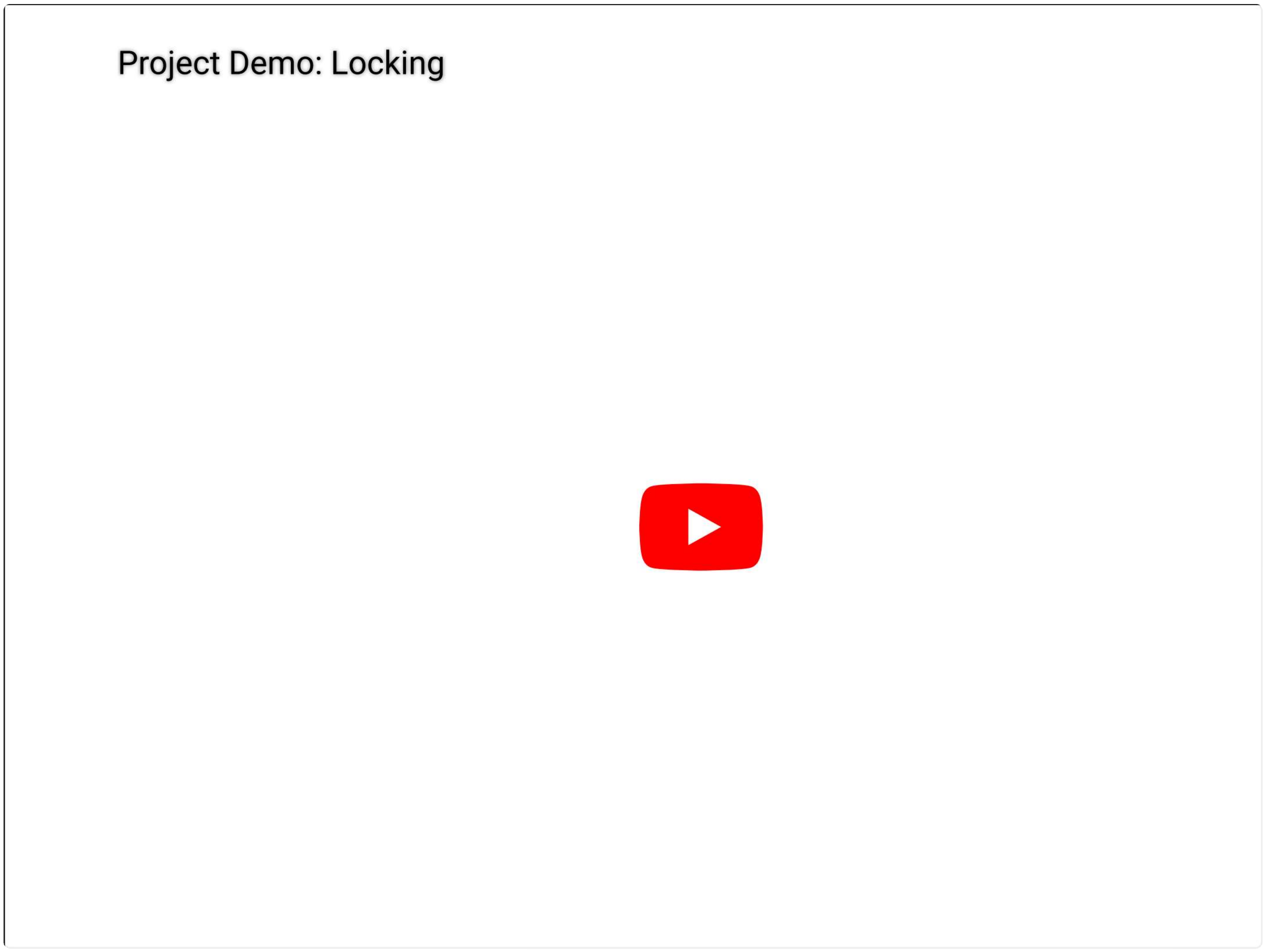
Now let's run:

```
INSERT INTO Students VALUES (3000, 'Name', 'Major', 5.0);
```

and display the locks held using `\locks`. Notice how we've upgraded some locks and now hold an X lock on a page of the Students table.

Let's commit and end our transaction by running `COMMIT;`.

Locking is important when we have multiple clients modifying and querying the database. The demo below shows how this works. We run RookieDB as a server and open connections to this server to run transactions and interact with the database.



You can try it yourself by running Server.java. You can open multiple connections via client.py. Check out these files for more information!

Additional Notes

After this, you should pass all the tests we have provided to you under `database.concurrency.*`, as well as the tests in `TestDatabaseDeadlockPrecheck` and `TestDatabase2PL`.

Note that you may **not** modify the signature of any methods or classes that we provide to you, but you're free to add helper methods. Also, you should only modify code in the `concurrency` directory for this section.

Previous
Part 1: Queuing

Next
Testing