OATSdb: V1 Milestone

Providing Atomicity & Isolation For a Main-Memory Database

Avraham Leff

Yeshiva University

avraham.leff@yu.edu

COM3563: 2020

1. Introduction
2. Atomicity
3. Isolation
4. Testing: A Non-Definitive Discussion
5. Measuring Performance

Introduction

In the V0 milestone you implemented a "transactionally aware" main-memory database

- A Map-based programming model
- A server-side container which required client to be participate in a transaction when using the programming model
- Clients can "begin", "commit", and "rollback" transactions
- The V1 milestone will build upon your previous work

In the V1 milestone you will provide atomicity and isolation properties for your main-memory database

- These are the A and I from transactional ACID
- Specifically: you will use locking semantics as an implementation technique
  - The details are up to you ...
  - But: you're required to use "pessimistic" locking approach

What About Persistence?

- There is a common perception that databases are about persistence
- I grant that persistence is a very important feature ☺…
- But: it's independent of "atomicity" and "isolation"!
  - Sanity check: did any of the textbook or lecture algorithms ever get involved with "main-memory" *versus* "disk" issues?
- And: "main-memory" transactions definitely provide value
  - Because "all-or-nothing" semantics are very attractive
- These semantics require that the DBMS provide "atomicity" to address client-side failures or deliberate rollbacks
- These semantics require that the DBMS provide "isolation" to address multi-threaded environments

- The V1 milestone requires you to have a good understanding of Java concurrency
- The good news: Java has fantastic support built into the JDK
- The bad news: successive waves of increased JDK capabilities mean that there are a lot of ways to accomplish what you want
- Online information less helpful than usual: tends to assume that you already understand the big picture ☹
  - Also: such information may not have been updated even when "better alternatives" became available
- Book recommendation: Java Concurrency in Practice, by Brian Goetz and Tim Peierls
  - Yes, it's "old", published in 2006
  - Still very worth reading, especially if you read the newer Javadocs as well
  - Amazon

Interface from V0 will not change despite DBMS and TxMgr functional changes

- You can & will implement as you choose
- So long as the only required public APIs are those provided by `DBMS`, *Tx*, and *TxMgr*
  - And you support the OATSDBType dbmsFactory and txMgrFactory methods for the V1 enum value
- See the requirements document for package and $DIR information

Atomicity

- Independent of concurrency, transactions offer the ability to "draw a box" around a chunk of code that states:
  - Do not commit this work until the entire chunk succeeds
  - If a failure occurs in the middle of the transaction, clients should not see any change from the pre-transaction state of the data
    - Semantics: as if the transaction had never executed ☺
- Clients can explicitly force the transaction to be rolled back through the `TxMgr.rollback()` API
  - Expectation: no code effect of that transaction will be visible to the next transaction or piece of code
  - Same semantics as unintentional failure
- Note: the DBMS may trigger a rollback for reasons of its own
  - "Stay tuned"

Any ideas?

- Remember: there are several possible techniques

> You can use any implementation technique that you want: here, the only requirement is that you provide the semantics specified above!

- What follows is only a suggestion to get you thinking
- Key idea: rollback requirement implies that the DBMS must be able to restore all client data database system to pre-transaction state
- At the same time, the DBMS must make client-induced data changes visible to the client as her code executes
  - Meaning: all CUD operations must behave "normally" from the client's perspective
  - (And as we provide isolation, those CUD operations should be "invisible" to other clients)
- How can we accomplish these seemingly contradictory goals?

- Not my terminology (see Wikipedia)
  - Not clear whether my usage is broadly adopted ...

- Key idea: apply client's ongoing tx work to a copy of the database
- I'll refer to this "copy" as a shadow db
  - If tx commits, propagate shadow db state to "real" db
  - If tx rolls back, "no harm, no foul", just silently dispose of the shadow db ☺

Issue you'll want to consider:

- Do we need to provide clients with a complete copy of the database?
- Do we need to provide clients with a copy of data that they've only read (not accessed via a CUD operation)?

- This approach really shines with respect to handling tx failure and rollback ☺
- But: how do we propagate the client's work to the "real" database when a transaction commits?
- Conceptually: want to (atomically) replay all of the tx's work against the "real" db
- What would such a log look like?
- At a minimum: need to log the method name and the method parameters
- But: that's not enough!
  - Remember: we're using a Java programming model
  - Method state involves more than the method parameters!
  - Method "state" also involves the entire state of the object instance on which the method was invoked
- Can **you suggest** how we can propagate the shadow database to the real database?

Use Java Serialization

- (Almost) all Java objects can be serialized
  - Meaning: converted into a stream of bytes
- Take a look at the Java Object Serialization Specification (or a textbook)
- Serialized objects can be subsequently deserialized
  - Meaning: the stream of bytes are converted (back) to a live object instance
- So an object can be “flash frozen” now and “reconstituted” later
- Example scenarios:
  - Save the bytes to a file
  - Transmit the bytes across the network
  - And: for the V1 milestone ...
    - Copy the bytes from the “real db” → “shadow db”
    - Copy the bytes back from the “shadow db” → “real db”
  - Bytes are bytes ☺

Java Serialization: Keeping It Simple

- You don’t need anything subtle from Java serialization in this project
  - You’re not allowing external clients to use serialization to pass you objects with possibly risky data
    - That can be a serious security problem
- And: “we’ll cheat” and not worry about ensuring that object evolution (versioning) doesn’t break your code
- Plenty of material on the ’net on how to do the basics ...
- (see my previous link to the serialization protocol ...)
- Make sure you understand Java’s approach to Object Streams

> Key point: we’re using serialization as an easy (for us) way to make object copies on demand

Values must be serializable because they are "copied via serialization" from real Map to shadow Map (and back again)

- Q: do Map Keys have to be serializable as well?
  - Why or why not?
  - A: for V1: Keys do not need to be serializable
  - Because they stay constant during the transaction!
  - Only Values associated with keys may change
- Note: this description (misleadingly) makes serialization sound like work

> *A Java object is serializable if its class or any of its superclasses implements either the `java.io.Serializable` interface or its subinterface, `java.io.Externalizable`.*
>
> *Java Tutorial*

- You'll find that most objects are serializable "out of the box"

- The **OATSdb** programming model requires that database resources only be accessed within a transaction
- Q: Can we prevent the client from breaking the rules with the following (trivial) scenario?

```
TxMgr.begin();
// get reference to Map, then get reference to Account instance
final Account account = AccountMap.put("key", 42);
TxMgr.commit();
// Client accesses Account instance (outside a tx)!
final int value = account.get("key");
```

- A: the “shadow db” approach also protects the DBMS from this scenario
- Serialization ensure that your clients only get access to “shadow Map” and “shadow Object” references
- The “Real map” and “real Object” reference are different references!
- These semantics follow from the fact that “serialization” creates brand new references
  - State of the new reference is identical to original object
  - But the references are different!

Important caveat:

- Unfortunately, we can't prevent the client from using that ill-gotten Account reference ☹
- Ideally, we'd like to provide semantics that permit clients to
  1. Acquire a reference in tx_1
  2. Use that reference in tx_2
- ...With the DBMS silently "swizzling" the client's reference to point to the "real" server's reference
- Similar semantics to "single-level store" semantics: client's shouldn't have to worry about transaction subtleties!
- Note: you must provide precisely these semantics for Map references!
  - Map references acquired in a committed (or rolled back) tx_1 are valid in tx_2

- Q: if we want this "single-level" store semantics . . .
  - And we are providing these semantics for Map references . . .
  - Why not insist on these semantics for Map.Entry references as well?
- A: because this would require a lot of work on your part ☺
- In Vo, you must have figured out a way to provide clients with a "Map" reference that's really a reference to an enhanced Map object
  - One that's transactionally aware in a way that a "vanilla" Map reference is unable to be
- That's fairly easy to do because the Map API is "known in advance"
- Requiring similar behavior for arbitrary Map.Entry references requires a "code generation" step that must precede deploying code to your DBMS

- After consultation with experts, I decided that a “code generation” phase requires too much work for too little pedagogical insight ☹
- Also: the straightforward implementation technique would still constrain the programming model to require that DBMS objects have interfaces
  - That constraint is unacceptable: e.g., would disallow String and Long values
- Bottom line: clients must acquire fresh references to Map.Entry instances on a per-transaction basis
  - The DBMS doesn’t have to detect that a client is violating this rule
  - But: any work performed by a client on a “stale” reference is wasted!
    - State modifications performed on the stale reference are not performed on either the “real” or a “shadow” copy

If you can think of a way to solve these issues neatly, please speak to me (after class) ☺

- You absolutely do not have to adopt the implementation approach we've been discussing!
- The isolation requirements are the following:
  - Work performed in tx_1 is only visible to tx_2 after tx_1 commits
  - If tx_1 fails or is rolled back, its work will not be visible to tx_2
- All of the V0 requirements apply "as is" to V1, modulo the reference semantics discussed above
  - Map references do not have to be (re)acquired in a new tx
  - Map.Entry references do have to be (re)acquired in a a new tx
    - Otherwise: tx_1 work performed on a stale reference will not be visible to tx_2
  - But: the DBMS is not responsible for throwing an exception if a client uses a stale Map.Entry reference
- Note: these atomicity requirements are independent of isolation!
  - Here tx_2 refers to a transaction that executes after tx_1 completes

Atomicity: Summarizing The Suggested Approach

- Clients never get a reference to a "real db object"
  - Only to a "shadow db object"
- On a per-tx basis, DBMS grows a set of "shadow db" objects that are associated with this transaction instance
  - One set of objects per Map instance
- All of the client's work is performed on the shadow database!
- The DBMS transparently creates the "shadow db" objects using Java Serialization on behalf of the client
- `Tx.commit()`: the DBMS copies the shadow database state back to "real db", using Serialization to protect the "real db" from the client's current set of references
- `Tx.rollback()`: the DBMS discards "shadow db" objects so that they are not visible to the "real db"

Isolation

- Atomicity is about changing database state in "all or nothing" fashion
- Isolation is about protecting one database user from another
- In our "single user" implementation, this may seem artificial
  - But the algorithms are the same as for the "multi-user" implementation
- The problem is all about concurrency: intuitively, "how does the system protect two users from trashing each other's work"

- Pessimistic: assume that users will concurrently access the same data
  - DBMS must protect one from the other in advance
  - As discussed in lecture, pessimistic approach is implemented with a family of lock-based protocols
- Optimistic: assume that users will not concurrently access the same data
  - DBMS verifies that assumption when transaction commits
  - Rolls transaction back if assumption turns out to be false
- In V1, you must implement a lock-based protocol to provide isolation
- I urge you to read a good book on the concurrency facilities provided by the JDK
  - Followed by multiple readings of the java.util.concurrent Javadocs and java.util.concurrent.locks Javadocs
- Otherwise: you likely will reinvent the wheel, badly ☹

- Assume that the DBMS "locks" a resource on behalf of tx_1 such that tx_2 cannot access that resource
- What is the result of tx_2 invoking "get", "put", etc on the locked resource?
  - Q: should the DBMS throw an Exception?
  - A: absolutely not!
  - Think of all the tx_2 work that will be wasted simply because tx_1 is (temporarily) holding that lock
  - Instead: tx_2 blocks, waiting for DBMS to unlock the resource
- Q: when should the DBMS unlock the resource?
- A: when tx_1 either commits or rolls back …

Lock Granularities

- Naive approach: lock the entire database on a per-tx basis
  - Advantages: easy to implement ☺
  - Disadvantages: resulting performance will be atrocious ☹
- Slightly less naive: lock on per-Map granularities
  - Similar advantages and disadvantages as above
- Can we do better?
- Sure: at a minimum, lock on per MapEntry granularities
  - Motivation: this will increase DBMS concurrency because tx_1 and tx_2 can execute concurrently
  - Note: the finer granularity will not provide benefit if the two transactions access the same resource concurrently

> You must implement locking at MapEntry granularities!

- As you know from lecture, we can further refine a lock-based protocol by allowing resources to be locked in one of two modes:
  - Shared mode: tx_1 can only read the resource
    - And tx_2 can also read that resource
    - Potentially increasing concurrency considerably
  - Exclusive mode: tx_1 can both read & write the resource
    - No other tx_2 can even read that resource
    - Reduces concurrency, but required to provide transactional isolation

OATSdb V1 simplification: all locks are exclusive locks

Motivation:

- Given `Map.get` semantics, would be very difficult for the DBMS to detect when a lock must be "upgraded"
  - There is no API through which the client declares that she's about to modify the state of an object
  - She can acquire a Map.Entry reference via `Map.get`, then modify the reference through that object's API
- Good news: less work for you ☺
- But be aware of the performance implications: **OATSdb** will be unable to take advantage of applications with high read/write ratios ☹

- V1 will implement the strict 2PL locking protocol (refer back to lecture)
- Advantages: guarantees transactional serializability without requiring that you generate and analyze transaction schedules
  - Note: “tx serializability” has totally different meaning from the Java “serializability” we’ve been referring to ☺
- Implication: if you use the 2PL protocol, you’re guaranteeing transactional isolation

OATSdb Approach

- DBMS acquires lock on transactional resource "on demand" for tx_i
  - Blocks if another tx_j has previously acquired a lock on that resource
  - Proceeds only after lock is released
- Transactional commit or rollback releases all locks acquired in tx_i
- Where should you implement locks?
- Suggestion: at the boundary between the "real db" and the "shadow db"!
  - Atomicity implementation: client operates only on "shadow db" resource
  - Isolation implementation: client allowed to copy from "real db" to "shadow db" only if client first acquires a lock on that resource

- Because V1 is using a pessimistic, lock-based, protocol, client transactions can potentially deadlock
- Scenario:
  - tx_1 access $MapEntry_i$ and $MapEntry_j$ in that order
  - tx_2 access $MapEntry_j$ and $MapEntry_i$ in that order
  - DBMS grants locks in the order that the transactions request them
  - Both transactions get “stuck”: no further progress possible ☹
- From lecture you know that basically, two approaches possible
  - Prevent deadlocks
  - Detect and recover from deadlocks
  - Each of these high-level strategies has many, many, lower-level implementations

OATSdb approach: Implement "deadlock prevention" by using a timeout-based scheme

1. tx_1 requests a lock at time t_1, but is blocked by the DBMS
2. DBMS starts a clock ticking with a system-specified timeout period
   - Timeout expires at $t_1 + \mathit{SystemTimeout}$
3. If timeout expires and tx_1 still hasn't acquired the lock it needs ...
   - The DBMS rolls back tx_1, and releases all locks currently held by tx_1
   - (This allows other transactions to make progress)
4. Else ...

   4.1 The transaction previously holding the lock releases that lock

   4.2 The DBMS "wakes up" tx_1, allowing it to acquire the lock and resume execution

```
public interface ConfigurableDBMS extends DBMS {
  /** Sets the duration of the "transaction timeout".
   * A client whose transaction's duration exceeds
   * the DBMS's timeout will be automatically rolled
   * back by the DBMS.
   *
   * @param ms the timeout duration in ms, must be
   * greater than 0
   */
  void setTxTimeoutInMillis(int ms);

  /** Returns the current DBMS transaction timeout
   * duration.
   *
   * @return duration in milliseconds
   */
  int getTxTimeoutInMillis();
}
```

Your V1 DBMS must implement this interface

Problems With Timeout-Based Scheme

- Key problem: what's the optimal timeout duration?
- Set it too high $\Rightarrow$ limits concurrency
- Set it too low? $\Rightarrow$ may result in very irritated (rolled back) clients
  - Their transactions may have actually been making progress, just taking a long time
- Keep these serious issues in mind, but ...
- Don't worry about them for your V1 implementation!
- I'll simply supply your DBMS with an arbitrary timeout value, you'll plug that value into your lock-based concurrency control implementation

- Clients must acquire a lock on each MapEntry that her transaction accesses
  - Whether via “get”, “put”, “remove”
- Transaction execution blocks if a lock cannot be acquired
  - Presumably because another transaction has acquired the lock on that resource and not yet completed
- Transactions “queue” for resource locks
  - DBMS must ensure fairness
- Happy path: tx acquires locks on all its resources and commits
- Otherwise: tx times out and DBMS rolls back the transaction
  - This scenario manifests as a `edu.yu.oatsdb.base.ClientTxRolledBackException` to the client

Starvation Still Possible

<u>Scenario</u>

1. Tx acquires locks on all resources
2. Tx enters a long running computation
    - Or an infinite loop ☺
3. Or throws an exception
4. What happens to transactions blocked on this transaction's resource set?

- They're doomed ☹
- DBMS has no way to rollback another thread that's executing "arbitrary code"
- Programmers must understand this and adopt a "release locks as fast as possible" approach
    - Burden is on the transaction code
- Alternatively: DBMS can run a periodic sweep at transaction granularity to detect whether a transaction is taking too long
    - Release locks as necessary & rollback the transaction
    - Not responsible for this feature in V1

The “block while acquiring lock” approach relies on the fact that Java threads:

- Can sleep (suspend their own execution)
- And receive notifications that they were interrupted (told by another thread to do something else)
- This Java thread properties are the basis of any solution for “block & resume” implementation!
- This is true whether or not you directly exploit these properties or whether you use JDK classes that exploit these properties ...
- But: please, please, don’t reinvent the wheel ☺

- Tx thread sleeps the system-specified duration
- When sleeper awakes, DBMS makes *n* attempts to acquire the lock on behalf of that transaction
- After *n* unsuccessful attempts, rolls back the transaction
- Advantage: simplicity
  - See `Thread.sleep` API
  - But: consider the serious problems with this approach ☹
- Disadvantage: performance (throughput)
  1. tx_1 acquires lock
  2. tx_2 blocks and sleeps for ten minutes
  3. tx_1 commits two seconds after tx_2 goes to sleep
- This naive strategy doesn't provide a way for the DBMS to proactively wake up the sleeping thread
  - Result: throughput takes a hit ☹

- DBMS aggressively wakes up sleeping transaction
- Whenever tx_1 either commits or rolls back ...
  - DBMS selects another tx_2 (if any) that is waiting for a tx_1 resource
  - Wakes up tx_2
  - tx_2 now acquires that lock and resumes execution
- Advantage: improves throughput because tx will block for the *min*(timeout, lock release)
  - Timeouts are an upper-bound on duration of tx blocking
  - Also could be used for “zombie transactions” (see above, not required for V1)
- Disadvantage: implementation can’t simply do a “sleep with a timeout”

Testing: A Non-Definitive Discussion

- Can create Maps
  - But not if previously created
  - But not if name is already associated with a different Map
- Can retrieve those Maps
  - But only if previously created
- Can insert a MapEntry
- Can retrieve that MapEntry
- Can delete that MapEntry

All of the above only if client thread is currently associated with a transaction!

- Can I trick your DBMS into returning a Map with different key class?
- Can I trick your DBMS into returning a Map with different value class?
- Can I trick your DBMS into returning a Map with different name
  - With the same key & values classes as previously created Map …
- Does your DBMS enforce the requirement that value class must be serializable
- Can I trick your DBMS into allowing client to supply an "empty" (null, or only whitespace) name?

- All access to *TxMgr* API must be done from inside a transactional scope
  - Only exception is: `begin` ☺
  - And: `getTx`, `getStatus`
- Once transaction commits, client is no longer in a transaction
  - Unless she starts a new transaction
- Ditto if the client does a transaction rollback
- Does your DBMS prevent nested transactions?
- Have you correctly implemented the required transaction state transitions?

- Is "put then get" activity visible after tx commits?
  - Meaning: visible to another transaction
  - (Important: all of these scenarios invoke the Map methods in different – but sequential – transactions)
- What about a "put then get then modify" scenario
  - Verifying that once committed, that object state becomes part of the "real db" . . .
- What about a "put then remove then get" scenario
  - That "get" should return `null`
- What about a "put then remove & put then get" scenario
  - The "put" (taking place in the same transaction as the "remove") should supersede the effect of the "remove"
- Is a reference to a MapEntry rendered "useless" after tx commits?
  - Meaning: can't "cheat" by modifying that object outside transactional scope

Rollback Requirements

- Is “put then get” activity not visible after tx is rolled back?
- Is an reference to a MapEntry rendered “useless” after tx is rolled back?
  - Meaning: can’t “cheat” by modifying that object outside transactional scope

The above suggestions represent a set of requirements that I may or may not choose to validate when I test your code.
Furthermore, the above suggestions in no way preclude the use of different tests of your code.

- These above suggestions are essentially “JUnit” tests
  - They are “single-threaded” tests
  - The single thread serially simulates the behavior of serial transactions
- They apply to the atomicity part of the V1 milestone

Definition

A multi-threaded test is a test that involves multiple, concurrently executing client threads.

Q: Do you need to perform multi-threaded tests against **OATSdb**?

A: **Absolutely**! The point of this exercise is to have a transactional system, one that provides atomicity and isolation to clients in the face of concurrent activity from database clients

Why Non-JUnit Tests?

- It is very hard to use JUnit (or similar test harnesses) to do multi-threaded tests
- There are various approaches out there
  - My reaction: much more trouble than they're worth ☹
  - You're more than welcome to use any system you choose

- Please let me know if you find something that "works"
- Meaning:
  - A library that lets me focus on my test semantics
  - A library that includes "tests setup" APIs that are easy to understand and are unobtrusive

- You want to set up a number of concurrent clients
- Have them stress-test your system
  - Issuing “get/remove/put” requests
  - Against the same data
- Your test code needs to detect whether the **OATSdb** system under test failed to provide its transactional guarantees
  - Example: did one thread modify a MapEntry concurrently with another thread’s actions?
  - Example: did a transaction involving multiple MapEntries violate atomicity?

I already have that test, and it’s not hard for you to create your own

Some classes that I found helpful . . .

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.FutureTask;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicLong;
```

1. Create some “example Map resources”
2. Create a sample transaction that uses these resources
3. Instrument them to report a concurrency violation
4. Have a thread-pool of clients that spin up concurrent transactions
5. Step back and let the sparks fly

Easy to Detect a Failure, Hard to Diagnose Reason

- Advantage of "stress test" approach: either your DBMS buckles under the pressure or it doesn't …
- Disadvantage: what do you do if (when?) your DBMS fails
  - It's very, very difficult to debug concurrent code
  - And: your test results are non-deterministic
    - May pass, then fail, then pass
    - One failure trace will differ from the next
- You'll be tempted to simply ignore the problem
  - Especially if it's intermittent
  - Don't: this will cost you points
  - And bad job appraisals in your career ☹
- Approach: augment the stress test with a set of precision, multi-threaded tests
  - Each of which focuses on a specific scenario

We want each test to:

- Be a reproducible unit test for that scenario
- That implies fairly strict control of a few threads
- Pare back the scenario to its essentials
- Then sprinkle lots of print statements throughout the code
  - But not so many that they interfere with the concurrency
- Key idea: specify the concepts that you can assert about the correct behavior of your DBMS
  - Assert them! Never rely on “println validation” ☺
- Let’s walk through one of my examples …
  - Again: you don’t have to do it this way
  - You may well come up with a better approach!

Let's call it: Tx2BlocksWhileTx1Locks

- Your DBMS must block tx_2 from accessing a resource that is currently locked by tx_1
  - Note: stress-test may not detect a failure because tx_1 may access, then release the lock, so quickly that difficult to detect that tx_2 was ever blocked in the first place
- There are different things that can go wrong with this scenario
- Here we verify that
  - tx_2 is in fact blocked by tx_1 lock
  - tx_2 will will not be rolled back if tx_1 releases the lock before the timeout period elapses
  - (Separate test that verifies that tx_2 is rolled back if tx_1 holds on to that lock for too long)

- You don't need a special transactional resource and a transactional "program" for this sort of test
  - Such as you do need for "stress test" code
- All you need is minimal code that invokes `begin` and `commit`
  - You're not testing the state of the resource, just the locking behavior (sequencing)
- You can specify the timeout for your DBMS under test
  - Example: normally might be a minute, you can set it to 500ms
- Code up an (inner) class that will implement tx_1 behavior and another that will implement tx_2 behavior
  - Have each implement `Runnable`
  - Now you can specify (with reasonable milli-second accuracy) how long the associated tx thread should sleep

Test Setup: I

- Create the Maps and transactional resource(s) that tx_1 and tx_2 will access
  - Will have to do this in a transaction ☺
- Create a data-structure into which the transactions can record the time that they perform various actions of interest
- Your test will involve assertions about the order in which the transactions create these records
- Your test will involve assertions about the “content” of these records
  - Example: what’s the timestamp of an individual record
- You definitely don’t want this data-structure to be a transactional resource ☺
- Advice: use static inner classes to keep your code in one Java class, and to allow sharing of the global instance variables

Tx_1 (Override `run`)

1. Begin tx
2. Access transactional resource
3. Sleep for half a timeout period
    - Key point: tx_1 is blocking tx_2 from accessing this resource
    - Even though tx_1 isn't "using" the resource, all resources accessed by this transaction are locked until tx_1 commits or aborts
4. When thread wakes up, commit the tx
    - Key point: this unlocks the resources that were previously locked by the transaction
    - We expect tx_2 to be able to access the resource

Have the thread create (and record) appropriate data that allows you to re-create the event timeline

1. Sleep a bit at the beginning to ensure no “race condition” with tx_1
2. Begin tx that involves accessing the same resource that tx_1 is using
3. Q: how long should tx_2 block for a v1 implementation?
4. When tx_2 is permitted to access the resource, record the time
   - We can assert that
     4.1 That tx_2 will access the resource
     4.2 And: assert (approximately) what time it did so
5. Commit the transaction

Have the thread create (and record) appropriate data that allows you to re-create the event time-line

- Have your `main` configure the timeout period you want
- Create tx_1 and tx_2 instances
- Create Thread instances that wrap these “transaction” instances
  - Tip: you can `setName` to assign useful names to the threads you’re creating
- Start the threads
- When the threads finish executing, perform your “assert” logic and determine whether the test passed or failed

Q: Can you see a problem here?

This Will Not Work

- `main` will start and finish before tx_1 and tx_2 have finished executing ☹
- Any suggestions?
- We could have the `main` thread sleep for our “best guess” as to how long the transaction threads will take
  - Plus a bit more for safety
- Better approach: use a `java.util.concurrent.CountDownLatch`
  1. Initialize to number of transaction threads
  2. `main` invokes `latch.await`
     - Will not proceed with program execution until latch value reaches 0
  3. Each transaction thread invokes `latch.countDown` (just before) it finishes execution
  4. `main` will “automagically” resume processing

Measuring Performance

- We've focused so far on transactional correctness
  - Specifically: does the DBMS provide atomicity and isolation guarantees
- Suggestion: (fairly easy) implementation
  - Allow exactly one client to use the DBMS at a time
    - Note: this will not provide "atomicity", just "isolation"
- Q: what's wrong with the above approach?
- A: we also want our implementation to have good performance ☺
  - Which we'll define here in terms of throughput
    - Meaning: "number of committed transactions per second"

- Not difficult to do the basics ...
  - Create transactional resources
  - Create transactional program
  - Wind them up, and measure throughput ...
- We need at least one "measurement knob": must specify the test's concurrency factor
  - Defined as the *"number of transactions executing at the same time"*
  - How do you expect performance to change as you turn this "knob" one way or the other?
- Also must think about:
  - How <u>many</u> transactional resources will you create?
  - Does it even matter how many you create?

Scenario:

- I create a 100,000 resources
- My transactional program accesses only a single resource
- I create 10 threads, each of which randomly selects a single resource to access in its transaction
- There is almost no thread contention for DBMS resources at all!
  - Resulting numbers are "useless"
  - I can increase the number of threads and get increasingly better numbers without paying a penalty for quite a while
- We need to devise a mechanism that can serve as a "tunable knob" on our infrastructure

Transactional Footprint Ratio: II

- What if we set the number of transactional resources to some constant?
  - Almost any value will do
- Intuitively: by varying the number of resources used by each transaction, we can increase or decrease the amount of competition between transactions
  - That number can be expressed as a percentage (of the constant number of resources available
- Tx Footprint Ratio: $accountsPerTx/nAccounts$
  - The larger the ratio, the more inter-transactional competition
  - And this ratio is independent of the actual number of transactional resources!

- We definitely want to measure throughput
- Should report the total number of transactions leaving the system
  - Be useful if we start to wonder whether we're running the measurement long enough
- Should report the number of "successful" transactions
  - Defined as "txs that committed"
- And the number of "failed" transactions
  - Defined as "txs that were rolled back"
  - Key point: our transactional logic will not willingly rollback a tx
  - So this number reflects the number of txs rolled back by the system because of "supposed" deadlocks

- Think about this issue: what is the role of timeout duration in your performance infrastructure?
  - This factor seems to be important as it interacts with the average duration of transaction execution
  - Transaction duration is itself (partly) a function of the number of transactional resources per transaction
  - Do we (and how would we) make these tunable knobs?
- Also: I urge you to test your intuition against observed phenomena
- Example: given observed performance for a concurrency factor of 1 . . .
  - How do you predict performance to change as we increase the concurrency factor?
  - How do you predict performance to change as we increase the transaction footprint ratio?
- If your intuition doesn't match your observations
  - Is this a "teachable moment"?
  - Or: a bug in your code?

I Plan To Measure Your Implementation's Performance

- Getting your system "correct" is most important
- Getting good performance is less important (for now) but still very important
  - For one thing: otherwise you can "cheat" by reducing the opportunities for concurrency so as to decrease the likelihood of isolation failures ☺

Note: I am seriously considering assigning bonus project points based on how your implementation performs relative to e.g., average of class performance

Introduction

Atomicity

Isolation

Testing: A Non-Definitive Discussion

Measuring Performance