Transaction strategies: The High Concurrency strategy

Learn how to implement a transaction strategy for applications with high user concurrency

Mark Richards June 16, 2009

Transaction strategies series author Mark Richards describes how to implement a transaction strategy in the Java™ platform for applications with high-throughput and high-user-concurrency requirements. An understanding of the trade-offs involved will help you ensure a high level of data integrity and consistency — and spare you painful refactoring work late in the development process.

View more content in this series

The API Layer and Client Orchestration transaction strategies that I've covered in previous articles in this series are core strategies that apply to most standard business applications. They are simple, relatively easy to implement, and robust and they offer the highest levels of data integrity and consistency. However, there may be times when you need to reduce a transaction's scope to gain throughput, improve performance, and increase concurrency in the database. How do you do this and still maintain a high level of data integrity and consistency? The answer is to use the High Concurrency transaction strategy.

The High Concurrency strategy derives from the API Layer strategy. The API Layer strategy, though solid and robust, does have a few drawbacks. Always starting transactions at the highest level in the call stack (the API layer) can be inefficient, particularly in cases of high-user-throughput and high-database-concurrency needs. Barring specific business requirements, holding on to a transaction longer than necessary consumes extra resources, holds locks longer, and holds on to resources longer than may be necessary.

About this series

Transactions improve the quality, integrity, and consistency of your data and make your applications more robust. Implementation of successful transaction processing in Java applications is not a trivial exercise, and it's about design as much as about coding. In this series, Mark Richards is your guide to designing an effective transaction strategy for use cases ranging from simple applications to high-performance transaction processing.

Like the API Layer strategy, the High Concurrency strategy frees the client layer from any transaction responsibility. However, this also means that you can have only one call from the client

layer for any given logical unit of work (LUW). The High Concurrency strategy aims to reduce the transaction's overall scope so that resources are locked for less time, thereby increasing the application's throughput, concurrency, and performance characteristics.

The gains you are likely to realize from using this strategy depend somewhat on which database you're using and how it's configured. Some databases (such as Oracle and MySQL using the InnoDB engine) do not hold read locks, whereas others (such as SQL Server without the Snapshot Isolation Level) do. The more locks you hold, whether they are shared or exclusive, the more you affect the concurrency, performance, and throughput of the database (and hence your application).

But obtaining and holding locks in the database is only part of the high-concurrency story. Concurrency and throughput also have to do with when you release the locks. Regardless of the database you use, holding on to a transaction longer than you need to will hold shared and exclusive locks longer than necessary. Under high concurrency, this can cause the database to promote the locking escalation level from a row-level lock to a page-level lock, and — under extreme circumstances — a page-level lock to a table-level lock. In most cases, you have no control over the heuristics the database engine uses for choosing when to escalate the locking level. Some databases (such as SQL Server) allow you to disable the page-level locking in hopes that it won't escalate from a row-level lock to a table-level lock. Sometimes this gamble works, but in most cases you will not notice the concurrency improvement you might have hoped for.

The bottom line is that in high-database-concurrency scenarios, the longer you hold database locks (shared or exclusive), the better chance that any of the following problems will occur:

- Running out of database connections, thereby causing wait states in your application
- Deadlocks that are due to shared and exclusive locks, resulting in poor performance and failed transactions
- Locking escalation to page-level or table-level locking

In other words, the longer your application is in the database, the less concurrency the application can handle. Any of the issues I've listed will cause your application to slow down and will directly reduce overall throughput and performance — and consequently the application's ability to handle a large concurrent user load.

Trade-offs

The High Concurrency strategy addresses high-concurrency requirements by reducing the transaction's scope to the lowest possible level in the architecture. This causes the transaction to complete (commit or roll back) faster than in the API Layer transaction strategy. However, as you may have learned from the story of the good ship *Vasa* (see Related topics), you cannot have it all. Life is all about trade-offs, and transaction processing is no exception. You simply cannot expect to provide the same type of robust transaction processing possible with the API Layer strategy and at the same time provide maximum user concurrency and throughput at peak load.

So, what do you give up when using the High Concurrency transaction strategy? Depending on how your application is designed, you may be forced to execute read operations outside

the transaction's scope, even when the read operations are used for update intent. "Wait!" you say, "You can't do that — you might end up updating data that has been changed since it was last read!" This is a valid concern, and also where the trade-off game starts. With this strategy, because you are not holding read locks on data, the chance of getting stale data exceptions when you execute the update operations increases. However, as with the good ship *Vasa*, it all comes down to which characteristic is more important: a solid, bullet-proof transaction strategy (like the API Layer strategy), or high user concurrency and throughput. In high-concurrency situations, it is extremely difficult to have both, and if you try, you might end up in a situation where the application does neither well.

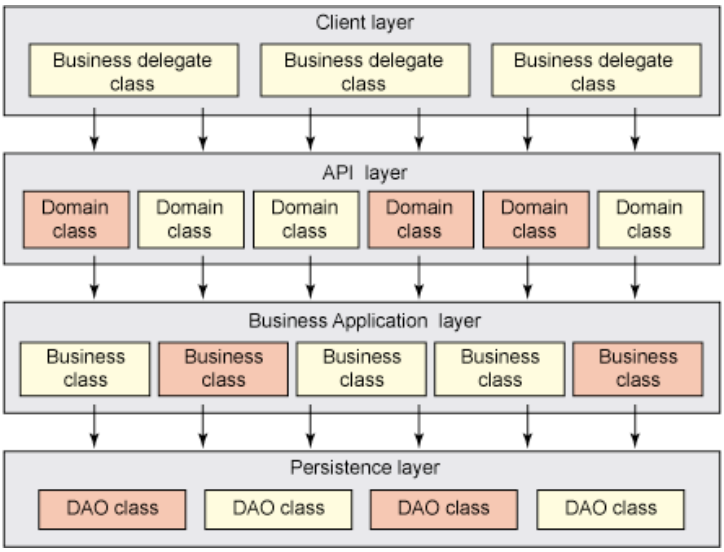
A second trade-off is an overall lack of transactional robustness. This strategy is difficult to implement, takes longer to develop and test, and is more error prone than the API Layer or Client Orchestration strategies. Given these trade-offs, you should first analyze your current situation to determine if using this strategy is the right approach. Since the High Concurrency strategy is a derivative of the API Layer strategy, one good approach is to start out with the API Layer strategy and load-test your application with a high user load (higher than you are expecting during your peak load). If you find you are experiencing poor throughput, poor performance, high wait times, or even deadlocks, then start moving to the High Concurrency strategy.

In the rest of the article, I'll show you some of the other characteristics of the High Concurrency transaction strategy and two approaches to implementing it.

Basic structure and characteristics

Figure 1 illustrates the High Concurrency transaction strategy via the logical application stack I've used throughout the *Transaction strategies* series. The classes containing transaction logic are shaded in red.

Figure 1. Architecture layers and transaction logic



Some of the API Layer strategy's characteristics and rules apply — but not all. Notice that the client layer in Figure 1 has no transaction logic, meaning that any type of client can be used for

this transaction strategy, including Web-based, desktop, Web services, and Java Message Service (JMS). And observe that the transaction strategy is spread throughout the layers below the client, but not completely. Some transactions might start in the API layer, some in the business layer, and some even in the DAO layer. This lack of consistency is one reason why this is a hard strategy to implement, maintain, and govern.

In most cases, you'll find that you need to use the Programmatic Transaction model to reduce the transaction scope, but sometimes you can still use the Declarative Transaction model. However, you generally cannot mix the Programmatic and Declarative Transaction models in the same application. It's a good idea to stick with the Programmatic Transaction model when using this transaction strategy so you don't run into issues down the road. However, if you do find yourself able to use the Declarative Transaction model with this strategy, you should mark all public write methods (insert, update, and delete) in whichever layer is starting the transaction with a transaction attribute of `REQUIRED`. This attribute indicates that a transaction is needed and will be started by the method if one is not already present.

As with the other transaction strategies, regardless of the component or layer you choose to start the transaction, the method that starts the transaction is considered the *transaction owner*. Whenever possible, the transaction owner should be the only method that performs commits and rollbacks on the transaction.

Transaction strategy implementation

You can use two main techniques to implement the High Concurrency transaction strategy. The *read-first* technique involves grouping read operations outside of the transaction scope at the highest application layer possible (usually the API layer). The *lower-level* technique involves starting the transaction at the lowest possible layer in the architecture while still being able to maintain atomicity and isolation of the update operations.

Read-first technique

The read-first technique involves refactoring (or writing) the application logic and workflow so that all processing and read operations occur first, outside of a transaction's scope. This approach eliminates unnecessary shared or read locks but can introduce stale data exceptions if the data is updated and committed before you have had a chance to commit your work. To deal with this possibility, make sure you use versioning if you are using an object-relational mapping (ORM) framework with this transaction strategy.

To illustrate the read-first technique, I'll start with some code that implements the API Layer transaction strategy. In Listing 1, the transaction starts at the API Layer and encompasses the entire unit of work, including all read operations, processing, and update operations:

Listing 1. Using the API Layer strategy

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void processTrade(TradeData trade) throws Exception {
   try {
```

```
      //first validate and insert the trade
      TraderData trader = service.getTrader(trade.getTraderID());
      validateTraderEntitlements(trade, trader);
      verifyTraderLimits(trade, trader);
      performPreTradeCompliance(trade, trader);
      service.insertTrade(trade);

      //now adjust the account
      AcctData acct = service.getAcct(trade.getAcctId());
      verifyFundsAvailability(acct, trade);
      adjustBalance(acct, trade);
      service.updateAcct(trade);

      //post processing
      performPostTradeCompliance(trade, trader);
   } catch (Exception up) {
      ctx.setRollbackOnly();
      throw up;
   }
}
```

Notice in Listing 1 that all of the processing is contained within the scope of the Java Transaction API (JTA) transaction, including all of the verification, validation, and compliance checking (both pre and post). If you were to run the `processTrade()` method through a profiler tool, you would see execution times similar to those in Table 1 for each method call:

Table 1. API Layer method profile — transaction scope

Method name	Execution time (ms)
`service.getTrader()`	100
`validateTraderEntitlements()`	300
`verifyTraderLimits()`	500
`performPreTradeCompliance()`	2300
`service.insertTrade()`	200
`service.getAcct()`	100
`verifyFundsAvailability()`	600
`adjustBalance()`	100
`service.updateAcct()`	100
`performPostTradeCompliance()`	1800

The `processTrade()` method's duration is a little over 6 seconds (6100 ms). Because the transaction starts at the method's beginning and terminates its end, the transaction duration is also 6100 ms. Depending on the type of database you are using and the particular configuration settings, you will hold both shared and exclusive locks for the duration of the transaction (starting from when the read operations are performed). Furthermore, any read operation performed from a method invoked by the `processTrade()` method might also hold a lock in the database. As you can guess, holding locks in the database for 6 seconds in this case does not scale to support a high user load.

The code in Listing 1 might work perfectly well in an environment without high-user-concurrency or high-throughput requirements. Unfortunately, this is the type of environment most people usually

test in. Once this code enters a production environment where hundreds of traders (perhaps globally) are placing trades, the system will most likely perform poorly, have poor throughput, and quite possibly experience database deadlocks (depending on the database you are using).

Now I'll fix the code in Listing 1 by applying the High Concurrency transaction strategy's read-first technique. The first thing to observe in the code in Listing 1 is that the update operations (insert and update) take only 300 ms combined. (I am assuming here that the other methods invoked by the `processTrade()` method do not perform update operations.) The basic technique is to perform the read operations and nonupdate processing outside of the transaction scope, and only wrap the updates within a transaction. The code in Listing 2 illustrates the refactoring necessary to reduce the transaction scope and still maintain atomicity:

Listing 2. Using the High Concurrency strategy (read-first technique)

```
public void processTrade(TradeData trade) throws Exception {
   UserTransaction txn = null;
   try {
      //first validate the trade
      TraderData trader = service.getTrader(trade.getTraderID());
      validateTraderEntitlements(trade, trader);
      verifyTraderLimits(trade, trader);
      performPreTradeCompliance(trade, trader);

      //now adjust the account
      AcctData acct = service.getAcct(trade.getAcctId());
      verifyFundsAvailability(acct, trade);
      adjustBalance(acct, trade);
      performPostTradeCompliance(trade, trader);

      //start the transaction and perform the updates
      txn = (UserTransaction)ctx.lookup("UserTransaction");
      txn.begin();
      service.insertTrade(trade);
      service.updateAcct(trade);
      txn.commit();
   } catch (Exception up) {
      if (txn != null) {
         try {
            txn.rollback();
         } catch (Exception t) {
            throw up;
         }
      }
      throw up;
   }
}
```

Notice that I moved the `insertTrade()` and `updateAcct()` methods to the end of the `processTrade()` method and wrapped them within a programmatic transaction. This way, all of the read operations and corresponding processing execute outside of a transaction's context, and consequently do not hold locks in the database for the duration of the transaction. The transaction duration in the new code is only 300 ms, dramatically lower than the 6100 ms from Listing 1. Again, the goal is to reduce the time spent in the database, thereby increasing the overall database concurrency and hence the application's ability to handle a larger concurrent user load. By spending only 300 ms in the database using the code in Listing 2, you allow for (theoretically) 20 times more throughput.

As you can see in Table 2, the code executing within the transaction scope is reduced to 300 ms:

Table 2. API layer method profile — transaction scope revised

Method name	Execution time (ms)
`service.insertTrade()`	200
`service.updateAcct()`	100

Although this is a significant improvement from a database-concurrency standpoint, the read-first technique introduces a risk: because no locks are held on the objects designated for update, anyone can update those unlocked entities during the course of this LUW. So you must ensure that the objects being inserted or updated are not normally updated by more than one user at a time. In the previous trading scenario, it's a safe assumption that only one trader will be working on the particular trade and account at any given time. However, this may not always be true, and stale data exceptions might occur.

One more thing: when using Enterprise JavaBeans (EJB) 3.0, you must tell the container that you plan to use programmatic transaction management. You can do this by using the `@TransactionManagement(TransactionManagementType.BEAN)` annotation. Notice that this annotation is at the class level (not the method level), indicating that you clearly cannot combine the Declarative and Programmatic transaction models in the same class. Pick one or the other and stick with it.

Lower-level technique

Suppose you want to stick with using the Declarative Transaction model to simplify the transaction processing but still increase throughput during a high-user-concurrency scenario. That's when you would use the lower-level technique within this transaction strategy. With this technique, you'll typically encounter the same trade-off as with the read-first technique: read operations are typically done outside of the context of the transaction scope. And implementing this technique will most likely require code refactoring.

I'll start again with the example in Listing 1. Rather than using programmatic transactions within the same method, you can move the update operations to another public method within the call stack. Then, when you are done with the read operations and processing, you can invoke the new update method; it will start a transaction, invoke the update methods, and return. Listing 3 illustrates this technique:

Listing 3. Using the High Concurrency strategy (lower-level technique)

```
@TransactionAttribute(TransactionAttributeType.SUPPORTS)
public void processTrade(TradeData trade) throws Exception {
   try {
      //first validate the trade
      TraderData trader = service.getTrader(trade.getTraderID());
      validateTraderEntitlements(trade, trader);
      verifyTraderLimits(trade, trader);
      performPreTradeCompliance(trade, trader);

      //now adjust the account
      AcctData acct = service.getAcct(trade.getAcctId());
```

```
      verifyFundsAvailability(acct, trade);
      adjustBalance(acct, trade);
      performPostTradeCompliance(trade, trader);

      //Now perform the updates
      processTradeUpdates(trade, acct);
   } catch (Exception up) {
      throw up;
   }
}

@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void processTradeUpdates(TradeData trade, AcctData acct) throws Exception {
   try {
      service.insertTrade(trade);
      service.updateAcct(trade);
   } catch (Exception up) {
      ctx.setRollbackOnly();
      throw up;
   }
}
```

With this technique, you are effectively starting the transaction at a lower level in the call stack, thereby reducing the amount of time you are in the database. Notice that the `processTradeUpdates()` method updates only the entities that are modified or created in the parent method (or above). Again, rather than holding the transaction for 6 seconds, you are holding it for only 300 ms.

Now for the hard part. Unlike the API Layer strategy or the Client Orchestration strategy, the High Concurrency strategy doesn't it lend itself to a consistent implementation approach. That's why Figure 1 looks like the face of an experienced hockey player (complete with missing teeth). For some API calls, the transaction might start and end at the API layer, whereas other times it might be scoped only at the DAO layer (particularly for single-table updates within the LUW). The trick is to identify methods that are shared between multiple client requests and ensure that if a transaction was started at a higher-level method, it will be consumed at the lower levels. Unfortunately, the effect of this characteristic is that a lower-level method that isn't the transaction owner might perform a rollback on exception. As a result, the parent method that started the transaction can't take corrective action on exception and will get an exception if it tries to roll back (or commit) a transaction that has already been marked for rollback.

Implementation guidelines

Some situations will require only a slightly reduced transaction scope to meet your throughput and concurrency requirements, whereas others will require a significant reduction to achieve the results you are looking for. Regardless of your particular situation, you can use the following implementation guidelines to help design and implement the High Concurrency strategy:

- Start with the read-first technique before embarking on the lower-level technique. This way transactions are at least contained within the application architecture's API layer and not spread throughout the other layers.
- When using declarative transactions, always use the `REQUIRED` transaction attribute rather than the `MANDATORY` transaction attribute to protect yourself against the chance that the method starting the transaction invokes another transactional method.

- Before embarking on this transaction strategy, make sure you are relatively safe in terms of performing read operations outside of the transaction scope. Look at your entity model and ask whether it is common, rare, or impossible for more than one user to act on the same entity at once. For example, can two users modify the same account at the same time? If your answer is that it is common, you run a greater risk of stale data exceptions, making this strategy a poor choice for your application profile.
- It is not a requirement that *all* read operations be outside the scope of the transaction. If you have a particular entity that is frequently changed by multiple users simultaneously, by all means add it to the transaction scope. However, be aware that the more read operations and processing you add to the transaction scope, the more you will reduce throughput and user-load capabilities.

Conclusion

It all boils down to trade-offs. To support high-throughput and high-user-concurrency requirements in an application or subsystem, you need high database concurrency. To support high database concurrency, you need to reduce database locks and hold resources for as little time as possible. Certain database types and configurations can handle some of this effort, but in most cases the solution boils down to how you design your code and transaction processing. Having some insight into these issues will alleviate painful and complicated refactoring work later. Choosing the right transaction strategy is critical for your application's success. For high-user-concurrency needs, look to the High Concurrency transaction strategy as a possible solution for ensuring a high level of data integrity and consistency while maintaining your high concurrency and throughput requirements.

Related topics

- *Java Transaction Design Strategies* (Mark Richards, C4Media Publishing, 2006): This book by this article's author offers an in-depth discussion of transactions on the Java platform.
- The *Vasa*: Read the true story of the *Vasa* (Wikipedia) and, in "Architectural Tradeoffs" (Mark Richards, 97 Things, July 2008), how it illustrates what happens when you try to have it all.
- *Enterprise JavaBeans 3.0 (5th Edition)* (Bill Burke and Richard Monson-Haefel): This is an excellent book for coming up to speed quickly with EJB 3.0.
- *Java Transaction Processing* (Mark Little, Prentice Hall, 2004): This book is another good reference on transactions.
- "*Java theory and practice*: Understanding JTS" (Brian Goetz, developerWorks, 2002): Get a handle on the basics of Java EE transaction processing in this three-article series.
- Chapter 9. Transaction management: More information about Spring transaction processing can be found in this section of the Spring Framework 2.5 documentation.
- Java EE APIs and Docs: You can find documentation for the EJB 3.0 specification here.