

Collection Concurrency White Paper

VERSION 2022

Jade Software Corporation Limited cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages, or loss of profits. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Jade Software Corporation Limited.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Copyright © 2025 Jade Software Corporation Limited.

All rights reserved.

JADE is a trademark of Jade Software Corporation Limited. All trade names referenced are the service mark, trademark, or registered trademark of the respective manufacturer.

For details about other licensing agreements for third-party products, you must read the Jade **Readme.txt** file.

Contents

Contents	iii
Collection Concurrency	4
Background to Collection Locking Behavior	4
Implicit Object Locking	4
Implicit Collection Locking	4
Releasing Locks	5
Transaction State	5
Load State	5
Transaction Phases	6
Processing Phase	6
Commit Phase	6
Collection Concurrency Feature Summary	6
Deferred Collection Methods	7
Deferred Inverse Maintenance	7
Collection Concurrency Feature Details	7
Conditional Collection Methods	8
Conditional Dictionary Methods	10
Common Exception Handling for Conditional Methods	11
Deferred Collection Update Methods	11
Deferred Collection Update Exception Handling	14
Deferred Update Visibility	15
Deferred Inverse Maintenance	15
Implementing Collection Concurrency	16
.NET API Collection Concurrency Methods	18
Benchmarking	22
Benchmark Scenarios	22
Benchmark Results	23
Appendix A Tips and Techniques	25
Locking Strategy	25
Deadlock Exceptions and How to Avoid Them	26

Collection Concurrency

Contention and concurrency are interconnected concepts. Achieving high concurrency has been a challenge in developing performant and scalable transaction processing systems with Jade due to contention for locks on persistent collections.

High lock contention results in low concurrency, leading to the following well-known impacts.

- Extends user response time for interactive workloads.
- Reduces throughput (transactions per second) for batch workloads.

This white paper describes deferred operation functionality introduced in the Jade 2020 release aimed at removing accidental complexity so that you can concentrate on delivering value to your business.

Background to Collection Locking Behavior

To help explain deferred operations, let's first review locking and introduce some terminology.

The locking behavior described in this section applies to persistent and shared transient instances.

Implicit Object Locking

When a Jade process updates an object, the Jade Object Manager (JOM) automatically locks the object to ensure that:

- Other processes cannot view the object in an incomplete state.
- Other processes cannot view uncommitted updates since those updates may be backed out if the transaction aborts.
- The updating process has the latest edition of the object before it is modified so that updates made by other processes are not lost.

Implicit Collection Locking

When a collection is accessed to be updated or read, JOM acquires a lock on the collection where the type of lock depends on several factors. One factor is whether the collection is being read or updated.

It is useful to distinguish between methods that update the collection and methods that read (or query) the collection.

- Updating methods

RootSchema collection methods like **add** and **remove** that update the receiver are annotated with the **lockReceiver** method option, as shown in the following example.

```
add(value: MemberType) lockReceiver, updating;
```

For all updating collection methods where **lockReceiver** is specified, JOM automatically acquires either an **exclusive** or an **update** lock on the collection. Normally, this will be an **exclusive** lock; it will be an **update** lock if the updating process has enabled **update** locks.

- Query methods

Collection query methods like **includes** or **getAtKey** that read the collection and do not update it do not have the **lockReceiver** method option specified, as shown in the following example.

```
includes(value: MemberType): Boolean;
```

For query methods where **lockReceiver** is not specified, JOM automatically acquires a **shared** lock on the collection to prevent it being altered by another process.

In addition, remember that when you iterate over a collection using an iterator or a **foreach** instruction, these call collection query methods which acquire a **shared** lock on the collection.

Releasing Locks

Shared locks are automatically released when the read function is finished unless the process remains in either transaction or load state. Let's consider transaction and load states.

Transaction State

As part of executing a **commitTransaction** or **abortTransaction** instruction, JOM automatically releases all transaction duration locks of the following types.

- Share
- Reserve
- Update
- Exclusive

It does not matter if the object was updated or not, or how the lock was obtained; that is:

- Implicitly by JOM
- Explicitly when your code included a lock instruction

It also does not matter when the lock was obtained; that is:

- Before the **beginTransaction** instruction
- Within transaction state

Session duration locks are not released at the end of transactions or at the end of load state.

Load State

All objects locked between **beginLoad** and **endLoad** instructions remain locked until the **endLoad** instruction is executed, at which time they are unlocked.

Objects locked with session duration and objects locked before the current **beginLoad** instruction are not unlocked by the **endLoad** instruction.

Unlock object requests are ignored between **beginLoad** and **endLoad** instructions.

For more details, including locking and unlocking in load state and lock state, see Chapter 6, "Jade Locking", in the *Developer's Reference*.

Transaction Phases

It is useful to consider the following two phases of a transaction. The:

- [Processing Phase](#)
- [Commit Phase](#)

The relevant actions executed in each phase of an updating transaction are shown in this section. For simplicity, exception paths that result from or cause a transaction abort are omitted.

Processing Phase

The processing phase of a transaction covers the application logic executed between a **beginTransaction** and a **commitTransaction** instruction.

In the processing phase, an application reads, creates, updates, and deletes objects and may explicitly lock objects including collections.

When not explicitly locked:

- Any object updated or deleted is locked with an **exclusive** or **update** lock.
- Collections that are read are **share** locked.
- A strict 2-phase locking protocol is enforced, which means any transaction duration locks of any type will *not* be released until the transaction commits (or aborts).

Commit Phase

The commit phase of a transaction covers the JOM actions processed when a **commitTransaction** instruction is executed.

The relevant actions are:

- **Update** locks are upgraded from **shared** to **exclusive**.
- Registered transaction callbacks are invoked.
- Object create, update, and delete operations are sent to the database server.
- Object updates are journaled by the database engine.
- A commit audit record is journaled, to ensure the transaction is durable.
- Transaction duration locks are released.
- Persistent notifications are sent.

With that background context in mind, let's look at conditional collection methods and deferred methods.

Collection Concurrency Feature Summary

The objective of the collection concurrency feature is to simplify the your task of writing code that efficiently maintains persistent collections, either directly or through automatic inverse maintenance, while minimizing contention and avoiding deadlocks.

Deferred Collection Methods

The Jade 2020 release introduced a set of collection methods that allow updates (and hence required locks) to persistent collections to be deferred until the enclosing transaction commits. For details, see "[Deferred Collection Update Methods](#)", later in this document.

Deferred Inverse Maintenance

The intention is to make it easy for you to start using deferred updates for automatically maintained collections. This is achieved with a deferred execution strategy for automatically maintained, multi-valued inverses, with two ways to control its scope.

- Schema defined per-property scope
- A runtime execution per-process scope, which overrides the property scope for the process

The deferred execution strategy applies to *all* state changes that currently trigger automatic collection maintenance, including:

- Assigning or changing the manual single value property
- Setting or changing key values for an auto-inverse member key dictionary
- Changing the values of properties used in a condition specified as a constraint on the multi-valued inverse

Collection Concurrency Feature Details

This section covers the following topics.

- [Conditional Collection Methods](#)
- [Conditional Dictionary Methods](#)
- [Common Exception Handling for Conditional Methods](#)
- [Deferred Collection Update Methods](#)
- [Deferred Collection Update Exception Handling](#)
- [Deferred Update Visibility](#)
- [Deferred Inverse Maintenance](#)

Conditional Collection Methods

Consider the following example `tryAddWithCheck` collection method, which, incidentally, was discovered in a production code base.

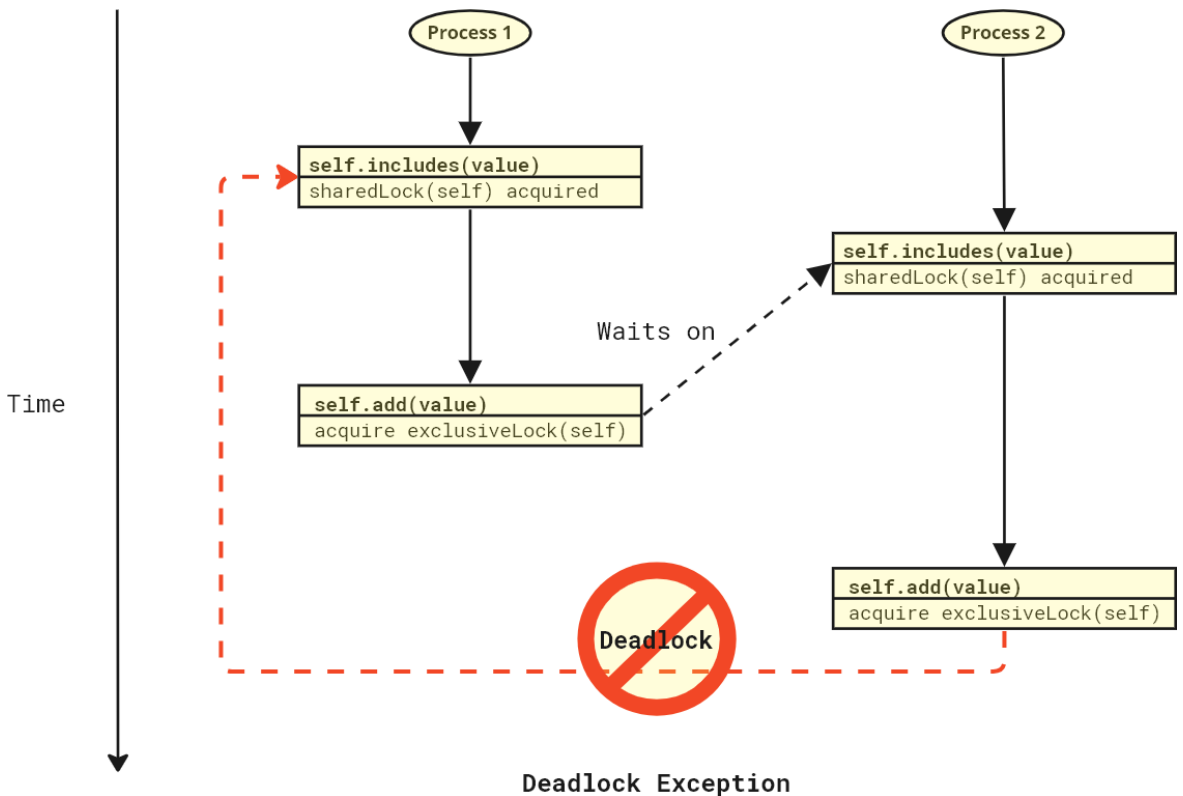
```
tryAddWithCheck(value : MemberType) : Boolean updating;
vars
begin
  if value <> null and not self.includes(value) then
    self.add(value);
    return true;
  endif;

  return false;
end;
```

The main issue with the method in the previous example is that it is prone to triggering deadlock exceptions when called on a persistent (or shared transient) collection.

A deadlock situation presents itself when two (or more) processes execute the `tryAddWithCheck` method in parallel.

The reason for the deadlock is depicted in the following diagram, which shows what happens when operations in a sequence overlap in a specific manner in transaction state.



As **Process 1** and **Process 2** call `self.includes(value)`, they each acquire a **shared** lock on `self` before continuing execution.

Process 1 is the first to execute `self.add(value)` and attempts to acquire the **exclusive** lock required for the update. The lock cannot be acquired because **Process 2** holds a conflicting **shared** lock, resulting in **Process 1** waiting.

When **Process 2** executes `self.add(value)` and attempts to acquire an **exclusive** lock on `self`, if we were to allow it to wait, **Process 1** and **Process 2** would end up waiting for each other to release their shared locks so both processes would come to a standstill until one of the lock waits timed out.

JOM doesn't allow the deadlock to manifest. As soon as JOM identifies a deadlock situation, the process that triggered the deadlock is given a deadlock exception and the action is aborted, which ultimately means the enclosing database transaction is aborted. In the previous example, **Process 2** is given the deadlock exception.

The previous deadlock scenario is often referred to as a **share to exclusive** upgrade deadlock. One way to avoid this form of upgrade deadlock is to acquire an **exclusive** lock on the receiver before it is read (and share locked). A simple way to accomplish that is to add the `lockReceiver` method option to the `tryAddWithCheck` method, as shown in the following example.

```
tryAddWithCheck(value: MemberType): Boolean lockReceiver, updating;
```

However, there's an easier and more efficient way to avoid upgrade deadlocks and that's to use the `tryAddIfNotNull` method.

The `tryAddIfNotNull` method is one of several conditional methods that are implemented by the `Collection` class and several of its subclasses. The following conditional collection methods provide a convenient and efficient way to add or remove a value from a collection without needing to check whether the collection contains the value.

- **tryAdd** method

```
tryAdd(value: MemberType): Boolean lockReceiver, updating, abstract;
```

This method adds the specified value if the value is not contained in the collection. It returns **true** if the value was successfully added or returns **false** if the collection already contains the value.

Exception Handling

Member key dictionaries with a no-duplicates constraint raise a duplicated key (1310) exception when the collection already contains the member key or keys with a different value.

This is not a case of adding the same object again; it is an attempt to add a different object that conflicts with an existing entry.

Applies to Version: 2020.0.01 and higher.

- **tryRemove** method

```
tryRemove(value: MemberType): Boolean lockReceiver, updating, abstract;
```

This method removes the specified value if the value is contained in the collection. It returns **true** if the value was successfully removed or returns **false** if the collection does not contain the value.

Applies to Version: 2020.0.01 and higher.

- **tryAddIfNotNull** method

```
tryAddIfNotNull(value: MemberType): Boolean receiverByReference,  
updating, final;
```

The `tryAddIfNotNull` method of the `Collection` class attempts to add the value specified by the `value` parameter to the collection if the value is not null and it is not already contained in the collection.

This method returns **true** if the value was successfully added; otherwise it returns **false**.

Applies to Version: 2020.0.02 and higher.

Conditional Dictionary Methods

The **tryCopy**, **tryCopyFrom**, **tryPutAtKey**, **tryRemoveKey**, and **tryRemoveKeyEntry** dictionary methods are methods defined on the **Dictionary** class and are implemented by the **DynaDictionary**, **ExtKeyDictionary**, and **MemberKeyDictionary** classes.

- **tryCopy** method

```
tryCopy(toCollection: Collection input): Collection;
```

This method copies the values from the receiver dictionary to the specified target **toCollection** collection that are not present in the target collection, and returns a reference to the target collection.

Notes Dictionary implementations (including the **MemberKeyDictionary** and **DynaDictionary** classes) support copying to an **ExtKeyDictionary** class with compatible keys.

Exception 1312 (*Class of value passed to a collection method incompatible with membership*) is raised if the member types are not compatible or exception 1000 (*Invalid parameter type*) for dictionary types if the keys are not compatible.

Applies to Version: 2022.0.01 and higher.

- **tryCopyfrom** method

```
tryCopyFrom(sourceCollection: Collection) lockReceiver, updating;
```

This method copies the values that are not present in the receiver from the collection specified in the **sourceCollection** parameter to the receiver.

Notes Dictionary implementations (including the **MemberKeyDictionary** and **DynaDictionary** classes) support copying to an **ExtKeyDictionary** class with compatible keys.

Exception 1312 (*Class of value passed to a collection method incompatible with membership*) is raised if the member types are not compatible or exception 1000 (*Invalid parameter type*) for dictionary types if the keys are not compatible.

Applies to Version: 2022.0.01 and higher.

- **tryPutAtKey** method

```
tryPutAtKey(keys: KeyType;  
            value: MemberType): Boolean abstract, lockReceiver, updating;
```

This method attempts to add the specified (key, value) pair to the dictionary. It returns **true** if the (key, value) pair was successfully added or returns **false** if the dictionary already contains the (key, value) pair.

Exception Handling

Dictionaries with a no-duplicates constraint raise a duplicated key (1310) exception when the collection already contains the member key or keys with a different value.

Applies to Version: 2020.0.01 and higher.

- **tryRemoveKey** method

```
tryRemoveKey(keys: KeyType): MemberType abstract, lockReceiver, updating;
```

This method attempts to remove a single (key, value) pair with the specified key or keys from the dictionary. It returns the member value if a single (key, value) pair was successfully removed or returns null if the dictionary does not contain the specified key.

Note No subclass of the RootSchema **Dictionary** class allows the insertion of a null object reference.

Applies to Version: 2020.0.01 and higher.

■ **tryRemoveKeyEntry** method

```
tryRemoveKeyEntry (keys: KeyType;
                   value: MemberType): Boolean abstract, lockReceiver,
                   updating;
```

This method attempts to remove the specified (key, value) pair from the dictionary. It returns **true** if the (key, value) pair was successfully removed or returns **false** if the dictionary does not contain the specified (key, value) pair.

Applies to Version: 2020.0.01 and higher.

Common Exception Handling for Conditional Methods

The conditional collection and dictionary methods do all of the same precondition checks on the parameters as their non-conditional counterparts do. When a precondition check fails, an exception is raised.

Deferred Collection Update Methods

In our *Performance Design Tips White Paper*, we discussed the importance of implementing an effective locking strategy. The locking strategy should support the greatest possible degree of multithreading.

A couple of key aspects of a recommended locking strategy are that objects should:

- Remain locked for the minimum time necessary
- Be locked in a consistent order, to reduce deadlocks

Deferred collection updates implement both aspects by:

- Capturing "conditional operations" during the processing phase of a transaction without fetching or locking the collection
- Executing deferred operations in the commit phase of the transaction
- Performing updates in collection OID order to avoid deadlocks

As an example, let's look at the **tryAddDeferred** method of the **Collection** class, which is defined as follows.

```
tryAddDeferred (value: MemberType): Boolean receiverByReference, updating,
               abstract;
```

The **receiverByReference** method option (introduced in the Jade 2020 release) instructs JOM to not lock or fetch the receiver.

When you execute **tryAddDeferred** in the processing phase of a transaction, Jade queues a request to try to add the specified value by executing a **tryAdd(value)** method.

The **tryAddDeferred** external method implementation is passed the collection OID (instead of the object buffer) and the **value** parameter. The implementation saves the collection OID, the value to be added, and the operation (in this case, the try add action). There is no need to access the collection, so it is not fetched and it is not locked.

The implementation registers a transaction callback so that it is called to process the deferred operations during the commit phase.

When your application logic executes a **commitTransaction** instruction, the commit phase of the transaction is entered. Registered transaction callbacks are invoked, one of which takes care of executing deferred collection operations. This is the stage where the required exclusive locks are acquired and (as mentioned) they are acquired in collection OID order.

Use of the **tryAddDeferred** method is ideal when your intent is to add the value if it's not present in the collection. You don't need to check whether it's present and you don't need to worry about multiple processes executing the same operation. If several processes queue a deferred try add action, the first process to execute the try add action achieves the intended result for all processes.

There are similar definitions for **tryRemoveDeferred**, **tryPutAtKeyDeferred**, and so on. In the processing phase, you can call **try<xxx>Deferred** methods multiple times for the same collection and even multiple times for the same collection and value. Multiple deferred operations to a **{collection, value}** combination are consolidated and the single net result is applied.

If you were to therefore execute the following sequence on the same collection, the **remove obj1** cancels the **add obj1** and so no change is applied.

```
tryAddDeferred(obj1) :  
tryRemoveDeferred(obj1) :
```

Deferred conditional collection methods are:

- Declared abstract at the collection or dictionary level with specific implementations for different collection types
- Supported for collection types that contain objects and for all collection instance lifetimes
- Not supported for primitive arrays

Deferred execution behavior is observed for persistent collections only (not shared transient collections).

Deferred execution methods and non-deferred execution updating methods cannot be called on the same collection within the same transaction. Attempts to do so raise exception 1471 (*Incompatible deferred update*); that is, collection locking is incompatible with prior updates.

The following summarizes the deferred collection update methods and their expected behavior.

- **tryAddDeferred** method

```
tryAddDeferred(value: MemberType): Boolean receiverByReference, updating,  
abstract;
```

This method attempts to add the value specified by the **value** parameter to the collection if it is not already present. For persistent dictionaries, a **tryAdd** operation is queued and executed when the database transaction commits.

Notes:

- For persistent dictionaries, the receiver is not fetched or locked.
- For transient dictionaries, a **tryAdd** operation is executed immediately.

- Returns **true** if a dictionary is persistent or the dictionary is transient, and the value was added; otherwise it returns **false**.

Applies to Version: 2020.0.01 and higher.

■ **tryPutAtKeyDeferred** method

```
tryPutAtKeyDeferred(keys: KeyType;
                    value: MemberType): Boolean receiverByReference, updating;
```

If the receiver has a persistent lifetime:

- The receiver is not fetched or locked.
- A request to execute a **tryPutAtKey** operation is queued and the method returns **true**.
- The queued **tryPutAtKey** operation is executed when the transaction successfully commits.

If the receiver has a non-persistent lifetime, the method is executed directly by calling **tryPutAtKey**.

Applies to Version: 2020.0.01 and higher.

■ **tryRemoveDeferred** method

```
tryRemoveDeferred(value: MemberType): Boolean receiverByReference, updating,
abstract;
```

If the receiver has a persistent lifetime:

- The receiver is not fetched or locked.
- A request to execute a **tryRemove** operation is queued and the method returns **true**.
- The queued **tryRemove** operation is executed when the transaction successfully commits.

If the receiver has a non-persistent lifetime, the method is executed directly by calling **tryRemove**.

Applies to Version: 2020.0.01 and higher.

■ **tryRemoveIfNotNull** method

```
tryRemoveIfNotNull(value: MemberType): Boolean abstract,
receiverByReference, updating;
```

The **tryRemoveIfNotNull** method of the **Collection** class attempts to remove the value specified in the **value** parameter from the collection if it is not null and is contained in the collection.

It returns **true** if the value was successfully removed; otherwise it returns **false**.

Applies to Version: 2020.0.02 and higher.

■ **tryRemoveKeyDeferred** method

```
tryRemoveKeyDeferred(keys: KeyType): MemberType, abstract,
receiverByReference, updating;
```

If the receiver has a persistent lifetime:

- The receiver is not fetched or locked.
- A request to execute a **tryRemoveKey** operation is queued and the method returns **true**.

- The queued **tryRemoveKey** operation is executed when the transaction successfully commits.

If the receiver has a non-persistent lifetime, the method is executed directly by calling **tryRemoveKey**.

Applies to Version: 2020.0.01 and higher.

- **tryRemoveKeyEntryDeferred** method

```
tryRemoveKeyEntryDeferred(keys: KeyType;  
                           value: MemberType): Boolean abstract,  
                           receiverByReference, updating;
```

If the receiver has a persistent lifetime:

- The receiver is not fetched or locked.
- A request to execute a **tryRemoveKeyEntry** operation is queued and the method returns **true**.
- The queued **tryRemoveKeyEntry** operation is executed when the transaction successfully commits.

If the receiver has a non-persistent lifetime, the method is executed directly by calling **tryRemoveKeyEntry**.

Applies to Version: 2020.0.01 and higher.

Deferred Collection Update Exception Handling

When invoked in the processing phase of a transaction, the deferred conditional collection and dictionary methods do all the same precondition checks on the parameters as their existing non-deferred counterparts do. When a precondition check fails, an exception is raised.

Precondition checks are repeated when deferred operations are applied in the commit phase. The likelihood of the precondition checks failing in the commit phase is low. For example, with member key dictionaries, it won't be possible to encounter any parameter precondition check failure because the object being added is locked and in the remove case, we do not require that the object being removed exists.

Unlike **update** locks, there is no window where an acquired **shared** lock is dropped and a new **exclusive** lock is acquired, which means deferred updates are not susceptible to intervening update (1146) exceptions.

Like **update** locks, deferred operations must acquire an **exclusive** lock on the target collection prior to committing, which means object locked (1027) and deadlock (1081) exceptions can still occur. How these are handled by the application is much the same as **update** lock scenarios.

- Object locked exceptions

Lock timeouts are reported as normal object locked (1027) exceptions. This allows exception handlers to determine the object involved, to retry the lock, and to abort or continue as required. An **exclusive** lock should be retried for the object involved. When a lock exception is continued, the commit action carries on allowing further deferred operations to be processed. If the lock exception handler attempts to continue without successfully retrying the lock, a lock cannot be continued (1225) exception is raised. If the lock exception is not successfully continued, the transaction will be automatically aborted.

- Deadlock exceptions

Deadlock (1081) exceptions can be raised with the usual deadlock exception information. This exception will trigger a transaction abort.

Deferred Update Visibility

A potential downside to deferred operations you need to consider is that the effects are not visible to the calling process until after the enclosing transaction has committed. In the Jade 2022 release, we provided collection query methods that take into account the net effect of deferred operations for the process. The **<xxx>WithDeferred** methods are summarized below.

- **getAtKeyWithDeferred** method

```
getAtKeyWithDeferred(keys: KeyType): MemberType;
```

The **getAtKeyWithDeferred** method of the **Dictionary** class returns a reference to an object in the receiver collection at the specified **keys** parameter value, taking account of deferred operations visible to the calling process. If an entry with the specified **keys** parameter value is not found, the method returns a **null** value.

Applies to Version: 2022.0.01 and higher.

- **includesKeyWithDeferred** method

```
includesKeyWithDeferred(keys: KeyType): Boolean;
```

The **includesKeyWithDeferred** method of the **Dictionary** class returns **true** if the receiver contains an entry at the specified **keys** parameter value, taking account of deferred operations visible to the calling process. If an entry with the specified **keys** parameter value is not found, the method returns **false**.

Applies to Version: 2022.0.01 and higher.

- **includesWithDeferred** method

```
includesWithDeferred(value: MemberType): Boolean;
```

The **includesWithDeferred** method of the **Collection** class returns **true** if the collection contains the object specified in the **value** parameter, taking account of deferred operations visible to the process. This method returns **false** if a null reference is passed to the **value** parameter. For a dictionary, the method returns **true** if the object is contained in the dictionary at its current key value, taking account of deferred operations visible to the process.

Applies to Version: 2022.0.01 and higher.

Deferred Inverse Maintenance

Deferred inverse maintenance covers the ability to specify that a collection inverse is maintained using a deferred execution strategy at an individual property level. Deferred execution is applicable to the **Automatic** and **Man/Auto** update modes on the Define Reference dialog, resulting in two additional update modes.

- **automaticDeferred**
- **manualAutomaticDeferred**

The following **Process** instance methods are provided to support enabling or disabling deferred execution for all inverse collection properties with the **Automatic** or **Man/Auto** option set for the current process.

- **useDeferredInverseMaintenance** method

```
useDeferredInverseMaintenance(enable: Boolean): Boolean, updating;
```

When called with the **enable** parameter set to **true**, this method enables the use of a deferred execution strategy for all automatically maintained collection properties for the current process, overriding the execution strategy for each property. When called with the **enable** parameter set to **false**, this method restores the schema-defined behavior for each property and returns the value of the prior enabled state.

Use Case

Evaluating, testing, and benchmarking the impact of using a deferred execution strategy before permanently applying its use in the schema.

Applies to Version: 2020.0.01 and higher.

■ **overrideDeferredInverseMaintenance** method

```
overrideDeferredInverseMaintenance(disable: Boolean): Boolean, updating;
```

When called with the **disable** parameter set to **true**, this method disables the use of a deferred execution strategy for all automatically maintained collection properties for the current process, overriding the execution strategy for each property. When called with the **disable** parameter set to **false**, this method restores the schema-defined behavior for each property and returns the value of the prior disabled state.

Use Case

Deferred execution has been specified at the property level in the schema because this was deemed to be appropriate for standard online processing. However, there is also a need to avoid the impact (particularly memory consumption) from a batch processing or bulk data load workload that is known to generate many collection updates. The **overrideDeferredInverseMaintenance** method provides a means to disable the schema-defined behavior for the duration of the batch or bulk load execution window.

Applies to Version: 2020.0.01 and higher.

Implementing Collection Concurrency

You can make use of any of the methods discussed in this document in your Jade application logic.

When considering the use of deferred update methods, it is important to check whether your logic calls methods that read collections in the same transaction that updates them. This is because reading and updating collections in the same transaction statement is prone to encountering deadlocks due to the implicit locking performed by Jade. For details, see the deadlock scenario described under "[Conditional Collection Methods](#)", earlier in this document.

On the other hand, if you have coded an **exclusive** lock before reading a collection within transaction state as a deadlock avoidance measure, you will need to rework this logic before you start to make effective use of deferred collection updates. This is best achieved by doing collection lookups in a read and validation phase before entering transaction state where you do your update processing.

The deferred execution model is a good choice when applied to collections that are updated but not read within the transaction. Here are some pros and cons to consider.

■ Pros

- The proposed methods can be called at any point within the transaction and since a deferred execution does not lock the collection at all, it means multiple processes can execute the deferred operations concurrently, whereas only one process at a time can hold an **update** lock on a given collection.
- If application logic does not read the collection in the updating transaction, a **shared to exclusive** lock upgrade does not happen.

- Con
 - The deferred add and remove operations are not visible to the calling process until after the enclosing transaction has committed. This is mitigated in Jade version 2022.0.01 by the **includesWithDeferred**, **getAtKeyWithDeferred**, and **includesKeyWithDeferred** methods.

When considering or implementing the deferred collection updates feature, utilize the **Process** class **useDeferredInverseMaintenance** method to facilitate the evaluation, testing, and benchmarking of the deferred execution strategy's impact before making a permanent commitment in a schema or code.

Note We are actively working on an enhancement, which will allow collection query methods to take into account the effects of deferred operations by default. We are aiming to make this enhancement available in a Jade 2022 feature release.

If, after implementing deferred collection updates, an increased memory consumption becomes an issue for certain workloads, consider overriding the deferred inverse maintenance strategy for automatically maintained collection properties. This could help avoid the impact, especially on memory consumption, caused by numerous collection updates during heavy batch or bulk data workloads. The **overrideDeferredInverseMaintenance** method is available for this purpose.

Deferred inverse maintenance for specific references is configured on the Define Reference dialog of the reference, as shown in the following example.

The screenshot shows the "Define Reference" dialog box. It is configured for a reference between the "Current Class" `BankAccount` and the "Related Class" `Customer`. The relationship is defined as `∞` to `1`. The "Property" section is named `myCustomer` and has a type of `Customer`. The "Multi Valued Property" section is named `allBankAccounts` and has a type of `BankAccountByNumberDict`. Both sections have "Deferred Execution" checked. The "Deferred Execution" checkbox in the Multi Valued Property section is highlighted with a red box. The status bar at the bottom indicates that the reference `'Customer::allBankAccounts'` is now updated automatically and its maintenance is deferred.

For more details, see "Defining an Inverse Reference Property", in Chapter 4 of the *Development Environment User's Guide*.

.NET API Collection Concurrency Methods

The collection concurrency methods are also provided in the Jade .NET API, with subtle differences in some cases to conform to a .NET idiom.

Tip For documentation about the .NET classes and other components, including those for collection concurrency, that comprise the Jade .NET API, see the **JadeDotNetAPI.chm** file, which is located in the installed Jade **documentation** directory (for example, **C:\Jade\JADE docs\documentation**).

■ TryAdd method

```
public bool TryAdd(  
    T item  
)
```

This method attempts to add the specified item to the collection. It returns **true** if the item was successfully added or returns **false** if the collection already contains the item.

Namespace: JadeSoftware.Joob

Applies to Version: 2020.0.01 and higher

■ TryRemove method

```
public TValue TryRemove(  
    TKey key  
)
```

This method attempts to remove the specified key from the collection. It returns **true** if the key was successfully removed or returns **false** if the collection does not contain the key.

This method also supports the functionality of the **tryRemoveKey** and **tryRemoveKeyEntry** Jade methods.

□ tryRemoveKey method

This method attempts to remove a single (key, value) pair with the specified key or keys from the dictionary. It returns the member value if a single (key, value) pair was successfully removed or returns null if the dictionary does not contain the specified key.

Note No subclass of the RootSchema **Dictionary** class allows the insertion of a null object reference.

□ tryRemoveKeyEntry method

This method attempts to remove the specified (key, value) pair from the dictionary. It returns **true** if the (key, value) pair was successfully removed or returns **false** if the dictionary does not contain the specified (key, value) pair.

Namespace: JadeSoftware.Joob

Applies to Version: 2020.0.01 and higher

■ TryPutAtKey method

```
public virtual bool TryPutAtKey(  
    TKey key,  
    TValue value  
)
```

This method attempts to add the specified (key, value) pair to the dictionary if it is not already present. It returns **true** if the (key, value) pair was successfully added or returns **false** if the dictionary already contains the (key, value) pair.

Namespace: JadeSoftware.Joob

Applies to Version: 2020.0.01 and higher

Exception Handling

Dictionaries with a no-duplicates constraint raise a duplicated key (1310) exception when the collection already contains the member key or keys with a different value.

■ TryAddDeferred method

```
bool TryAddDeferred(  
    JoobObject item  
)
```

This method attempts to add the value specified by the **item** parameter to the collection if it is not already present. For persistent collections, a **TryAdd** operation is queued and executed when the database transaction commits. For transient collections, the attempt is executed immediately.

Notes:

- For persistent collections, the receiver is not fetched or locked.
- For transient collections, a **TryAdd** operation is executed immediately.
- Returns **true** if the item is persistent, or was transient and added to the collection; otherwise it returns **false**.

Namespace: JadeSoftware.Joob

Applies to Version: 2020.0.01 and higher

■ TryAddIfNotNull method

```
bool TryAddIfNotNull(  
    JoobObject item  
)
```

This method attempts to add the specified item if it is not present in the collection. For persistent collections, the attempt is queued and executed when the transaction commits. For transient collections, the attempt is executed immediately. This method returns **true** if the item was successfully added; otherwise it returns **false**.

Namespace: JadeSoftware.Joob

Applies to Version: 2020.0.02 and higher

■ TryPutAtKeyDeferred method

```
public virtual bool TryPutAtKeyDeferred(  
    TKey key,  
    TValue value  
)
```

This method attempts to add the specified (key, value) pair to the dictionary if it is not already present. For persistent dictionaries, the attempt is queued and executed when the transaction commits. For transient dictionaries, the attempt is executed immediately.

If the receiver has a persistent lifetime:

- The receiver is not fetched or locked.
- A request to execute a **TryPutAddKey** operation is queued and the method returns **true**.
- The queued **TryPutAddKey** operation is executed when the transaction successfully commits.

If the receiver has a non-persistent lifetime, the method is executed directly by calling **TryPutAddKey**.

Namespace: JadeSoftware.Joob

Applies to Version: 2020.0.01 and higher

■ **TryRemoveDeferred** method

```
public bool TryRemoveDeferred(  
    TKey key,  
    TValue value  
)
```

This method attempts to remove the specified (key, value) pair if it is present. For persistent dictionaries, the attempt is queued and executed when the transaction commits. For transient dictionaries, the attempt is executed immediately. The method returns **true** if the dictionary is persistent or if the value was removed from the dictionary; otherwise it returns **false**.

If the receiver has a persistent lifetime:

- The receiver is not fetched or locked.
- A request to execute a **TryRemove** operation is queued and the method returns **true**.
- The queued **TryRemove** operation is executed when the transaction successfully commits.

If the receiver has a non-persistent lifetime, the method is executed directly by calling **TryRemove**.

This method also supports the functionality of the **tryRemoveKeyDeferred** and **tryRemoveKeyEntryDeferred** Jade methods.

- **tryRemoveKeyDeferred** method

If the receiver has a persistent lifetime:

- The receiver is not fetched or locked.
- A request to execute a **tryRemoveKey** operation is queued and the method returns **true**.
- The queued **tryRemoveKey** operation is executed when the transaction successfully commits.

If the receiver has a non-persistent lifetime, the method is executed directly by calling **tryRemoveKey**.

- **tryRemoveKeyEntryDeferred** method

If the receiver has a persistent lifetime:

- The receiver is not fetched or locked.
- A request to execute a **tryRemoveKeyEntry** operation is queued and the method returns **true**.
- The queued **tryRemoveKeyEntry** operation is executed when the transaction successfully commits.

If the receiver has a non-persistent lifetime, the method is executed directly by calling **tryRemoveKeyEntry**.

Namespace: JadeSoftware.Joob

Applies to Version: 2020.0.01 and higher

- **TryRemoveIfNotNull** method

```
public bool TryRemoveIfNotNull(  
    T item  
)
```

The **TryRemoveIfNotNull** method of the **Collection** class attempts to remove the specified item if it is not null and it is present in the collection. It returns **true** if the item was successfully removed; otherwise it returns **false**.

Namespace: JadeSoftware.Joob

Applies to Version: 2020.0.02 and higher

- **UseDeferredInverseMaintenance** method

```
public bool UseDeferredInverseMaintenance(  
    bool enable  
)
```

When called with the **enable** parameter set to **true**, this method enables the use of a deferred execution strategy for all automatically maintained collection properties for the current process, overriding the execution strategy for each property. When called with the **enable** parameter set to **false**, this method restores the schema-defined behavior for each property and returns the value of the prior enabled state.

Use Case

Evaluating, testing, benchmarking the impact of using a deferred execution strategy before permanently applying its use in the schema.

Namespace: JadeSoftware.Joob.Management

Applies to Version: 2020.0.01 and higher

- **OverrideDeferredInverseMaintenance** method

```
public bool OverrideDeferredInverseMaintenance(  
    bool disable  
)
```

When called with the **disable** parameter set to **true**, this method disables the use of a deferred execution strategy for all automatically maintained collection properties for the current process, overriding the execution strategy for each property. When called with the **disable** parameter set to **false**, this method restores the schema-defined behavior for each property and returns the value of the prior disabled state.

Use Case

Deferred execution has been specified at the property level in the schema because this was deemed to be appropriate for standard online processing. However, there is also a need to avoid the impact (particularly memory consumption) from a batch processing or bulk data load workload that is known to generate many collection updates. The **OverrideDeferredInverseMaintenance** method provides a means to disable the schema-defined behavior for the duration of the batch or bulk load execution window.

Namespace: JadeSoftware.Job.Management

Applies to Version: 2020.0.01 and higher

Benchmarking

Benchmarking of deferred operations was performed against a sizeable database with collections of a size between 1 million and 2 million entries using a 10 millisecond (10ms) workload. Batteries of tests were run, simulating both interactive user load and batch/bulk load processing. The testing was done on a server running Windows Server 2016, with 8 logical CPUs, so the server was running above 50 percent (%) CPU during some of the tests, especially when deferring updates.

Benchmark Scenarios

Benchmark scenarios were run to gauge improvements in user response time for interactive workloads and in throughput for batch workloads. The tables in "[Benchmark Results](#)", later in this document, illustrate that improvements were achieved in both scenarios.

The sample user transaction for the interactive workload benchmark scenario is described in the following table.

Interactive Workload Transaction	Duration (ms)
Do work	10
Shared lock/unlock a collection	
Work	10
beginTransaction;	
Update the collection	
Work	10
commitTransaction;	

The *work* in this scenario is a pure CPU load (calculations), to simulate doing other processing. The shared lock and unlock of a collection is to simulate accessing a collection partway through the transaction. The same collection is updated during the persistent transaction, so the shared locks are sometimes queued. The collection is locked early in the database transaction, so the exclusive locks are held for a while. This is the primary use case for this feature. The performance characteristics are the same whether the **tryAddDeferred** and **tryRemoveDeferred** methods are used or inverses are changed while using the **process.useDeferredInverseMaintenance** method call.

For the interactive workload scenario, there are five fat client users, who perform these transactions in a loop. Each user accesses randomly selected customer objects in a class. There is a pair of transactions; the first adds the object into the collection and the second removes it from the collection. The add and remove transactions are both in the format shown above, and the performance of each is virtually identical.

A multi-worker bulk load scenario was simulated by having five fat clients performing transactions containing many collection updates. The sample user transaction for the batch workload benchmark scenario is described in the following table.

Batch Workload Transaction	Details
reserveLock	For non-deferred transactions - required to prevent deadlocks
beginTransaction;	
Update 100 objects with inverse	10ms
commitTransaction;	

As with the online simulation, the *work* in this scenario is a pure CPU load (calculations), to simulate doing other processing. The **reserveLock** is needed if setting an inverse without the deferred updates, to prevent deadlocks. This was not used with deferred updates, as one of the design features is intended to help avoid deadlocks. The **reserveLock** was not required for the explicit add and remove test, as each user locked the collections in the same sequence. Again, the performance characteristics are virtually the same whether the **tryAddDeferred** and **tryRemoveDeferred** methods are used or inverses are changed while using the **process.useDeferredInverseMaintenance** method call.

As for the interactive workload scenario, there are five fat client users, who perform these transactions in a loop. Each user accesses selected customer objects in a class. There is a pair of transactions; the first adds the object into the collections and the second removes it from the collections. The add and remove transactions are both in the format shown in the previous table.

Benchmark Results

This section covers the benchmark results for the interactive workload and batch workload test cases.

The benchmark results for the interactive workload test case are listed in the following table.

Interactive Workload Test Case	Non-Deferred Elapsed Time (ms)	Deferred Elapsed Time (ms)	Change (%)
Five users	127	72	43
Five users, with no initial read (shared lock) access	110	68	38
Five users, collection updated at end of transaction	71	70	1.41
Single user	60	61	-1.67

The value in the **Change** column is expressed as the percentage of improvement in performance.

For:

- Five users, non-deferred elapsed time was 127ms, deferred elapsed time was 72ms, indicating a 43% improvement with deferral.
- Five users with no initial read access (shared lock), non-deferred time was 110ms, deferred time was 68ms, showing a 38% improvement with deferral.
- Five users with a collection update at the end of the transaction, non-deferred time was 71ms, deferred time was 70ms, resulting in a marginal 1.41% change.
- A single user, non-deferred time was 60ms, deferred time was 61ms, indicating a slight -1.67% change.

The benchmark results for the batch workload test case are listed in the following table.

Batch Workload Test Case	Non-Deferred Elapsed Time (seconds)	Deferred Elapsed Time (seconds)	Change (%)
Five processes, four collections updated	7.2	2.3	68
Five processes, three collections updated	5.6	2.1	62.5

The value in the **Change** column is expressed as the percentage of improvement in performance.

With:

- Five processes and four collections updated, non-deferred time was 7.2 seconds, deferred time was 2.3 seconds, reflecting a significant 68% improvement with deferral.
- Five processes and three collections updated, non-deferred time was 5.6 seconds, deferred time was 2.1 seconds, showing a 62.5% improvement with deferral.

Your application logic may be structured so that it's not possible to just flick a switch and start using a deferred collection update strategy for all your collection updates without reworking existing logic.

It is always useful to look at your current locking strategy with a view to taking advantage of deferred collection updates.

Locking Strategy

You should establish an effective locking strategy as part of your system design for every multiuser application.

An effective locking strategy should meet the following general objectives, although in some circumstances these objectives can conflict. When this occurs, you should typically aim to maximize multithreading over other objectives.

The locking strategy should support the greatest possible degree of multithreading (unless you are effectively in a dedicated single user mode; for example, where a solitary batch process is running offline).

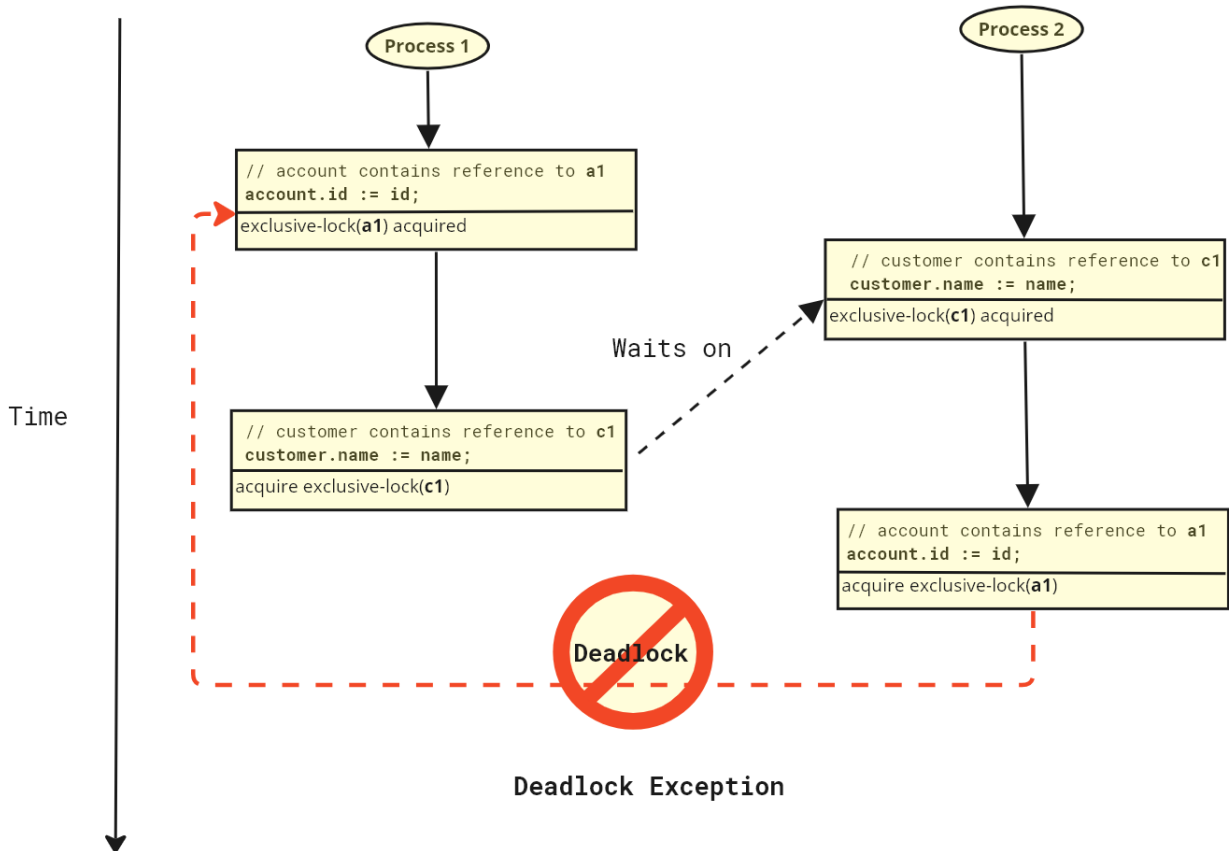
- Objects should remain locked for the minimum time necessary.
- Objects (and classes) should be locked in a consistent order, to reduce deadlocks.
- Objects that are going to be updated should be locked as soon as that is known, with a **reserve** lock, **update** lock, or **exclusive** lock.

For more details about designing and building an effective locking strategy, read Chapter 6, "Jade Locking", in the *Developer's Reference*, and "Locking" in Part 4, "Design Considerations", in the *Erewhon Demonstration System Reference* included with the Example schemas and also available on the Jade website.

Deadlock Exceptions and How to Avoid Them

A deadlock is a situation where one or more processes wait indefinitely for each other to release locks they need to be able to proceed. If left unresolved, none of the waiting processes would progress, leading to a standstill or impasse.

Consider the example in the following diagram, which shows what happens when object updates and their required locks overlap in a specific manner in transaction state.



In the previous example, **Process 1** acquires an **exclusive** lock on account instance **a1** to update it. Subsequently, **Process 2** updates and acquires an **exclusive** lock on customer instance **c1**. When **Process 1** updates customer instance **c1**, it needs to acquire an **exclusive** lock and must wait on **Process 2** to release its lock. If **Process 2** then updates account instance **a1**, it needs to acquire an **exclusive** lock on **a1**. If we were to allow **Process 2** to wait for **Process 1**, both processes would end up waiting for each other to release their locks and both processes would come to a standstill.

JOM doesn't allow the deadlock to manifest. As soon as JOM identifies a deadlock situation, the process that triggered the deadlock is given a deadlock exception and the action is aborted, which ultimately means the enclosing database transaction is aborted. In the previous example, **Process 2** is given the deadlock exception.

As another example of a deadlock situation, if two processes have shared locks on the same object and they both try to upgrade the lock to **reserve** or **exclusive**, a deadlock occurs. In this case, the second process to attempt the upgrade receives the deadlock exception. If two processes have shared locks on the same object and they both try to upgrade the lock to **update**, a deadlock will not occur. This is because the shared lock is released before the **update** lock is requested, even if in transaction state.

The more deadlocks you have occurring in your system, the greater the impact will be on the performance of the system, let alone the irritation to the users. It is therefore sensible to take practical steps to reduce the number of deadlocks that occur. There are a number of simple steps to take to assist in this; for example:

- Control the order in which classes are locked. For example, always lock the customer first, then lock the customer's accounts, and then lock the customer account's transactions.

This is probably best enforced by encapsulating the locking activity.

- Within a class, control the order in which objects are locked. For example, when locking multiple customers, always try to lock in one specific order (for example, the customer number order).

You can use the **Process::setPersistentDeadlockPriority** method to control which user will receive the deadlock exception. You could give higher priority to online users, for example, so that background reports receive the deadlocks or give higher priority to the background reports so that they don't hold locks as long.

If all processes involved in a deadlock have the same deadlock priority, the exception is usually given to the process whose action caused the deadlock.

You can set the **DoubleDeadlockException** parameter to **true** in the [JadeServer] section of the Jade initialization file to help diagnose deadlocks. With this setting, two of the processes involved receive the deadlock exception, which enables you to get two stack dumps to diagnose the contention.