Transactional Web Feature Server Design

Submitted To: Program Manager
GeoConnections
Victoria, BC, Canada

Submitted By: Jody Garnett
Brent Owens
Refractions Research Inc.
Suite 400, 1207 Douglas Street
Victoria, BC, V8W-2E7
jgarnett@refractions.net
Phone: (250) 885-0632
Fax: (250) 383-2140

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

This document outlines the changes required to provide strong transaction support for GeoServer. GeoServer is the reference implementation of the Open GIS Consortium's Web Feature Server Specification.

GeoServer currently uses In-Process locks that are not persistent and do not protect against modification by other applications. To provide strong transaction support we need provide a persistent, database supported Locking mechanism.

This document explains the required changes for strong transaction support to GeoServer and Geotools2. The existing designs of Geotools2 and Geoserver are discussed, several locking schemes are proposed, and our changes detailed.

Our changes will:

- Extend geotools2 DataSource API to allow Database backed locking
- Change GeoServer to use extended DataSource API in tandem with it's existing locks
- Transition to only using the GeoServer's In-Process locks where required
- Refactor the GeoServer in-Process locks into Geotools2

2 WEB FEATURE SERVER OVERVIEW

GeoServer is an open source implementation of the Open GIS Consortium's Web Feature Server Specification. The project's goal is to be the reference implementation of this specification.

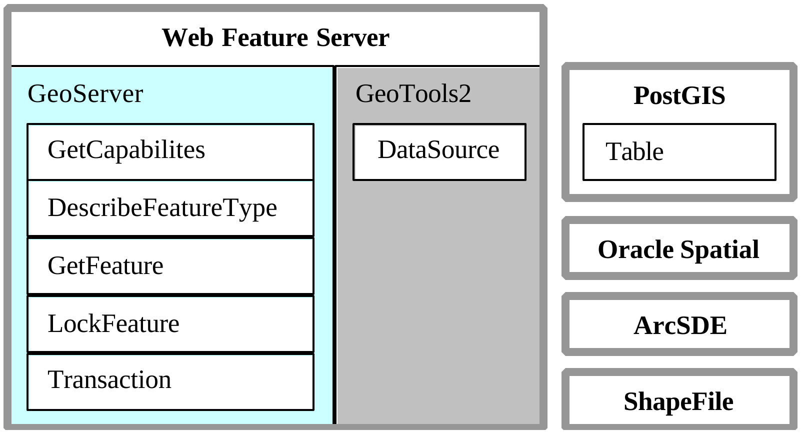


Figure 1 - GeoServer Layer Diagram

This diagram shows the main layers in the GeoServer application:

- GeoServer – implements the WFS Specification
- GeoTools2 – the DataSource API provides access to feature information
- Data – PostGIS, OracleSpatial and Shapefile are used to store feature information.

2.1 Web Feature Server Operations

The Web Feature Server Specification mandates that the following operations be supported:

- **GetCapabilities:**
  Describes the Operations supported by the WFS and the available Feature Types
- **DescribeFeatureType:**
  Returns an XML Schema document describing the requested FeatureType
- **GetFeature:**
  Returns feature information, from multiple FeatureTypes, that will validate against the DescribeFeatureType supplied XML Schema

The WFS Specification makes use of a Feature ID (or fid) to track features across operations.

2.2 Transaction Web Feature Server

In order to support transactions a WFS must support the following optional operation:

- **Transaction:**
  Operation support modification of a feature information.
  - Multiple Sub-Operations:
    The transaction operation consists of a number of DELETE, INSERT and UPDATE elements in a single request.
  - Partial Results:
    A request may ask a result of both successful modifications and failures.
  - Multiple FeatureTypes:
    Modifications on a number of different FeatureTypes from different data sources.

Web Feature Servers that support this operation are called Transaction Web Feature Servers.

2.3 Web Feature Server Long Transaction Support

In order to support long transactions a Web Feature Server provide the following operations:

- **GetLock:**
  Requests a LockID for a number of features specified by fid or via a filter.
  Interesting properties of this request:
  - Partial Results:
    A lock may be returned that lists both features it was able to lock , and features it could not lock
  - Duration:
    Lock requests can be requested for a duration, after which they are no longer valid
  - LockIDs:
    The returned LockID can be used for later Transaction operations.
- **GetFeature:** (extended with **GetFeatureWithLock**)
  The GetFeature operation can be extended to simultaneously perform a locking operation
- **GetTransaction:** (LockID Element)
  Transaction Operations can now use a number of user supplied LockID

This long-term locking facility is needed to provide concurrency for data modification. With a web feature server, it is likely that several users will try to access the same data at once. It is important to ensure that a user's transactions do not interfere with another's.

2.4 Example Locking Workflow

The following locking workflow assumes:

- Road Feature Type: road1, road2, and road3
- River Feature Type: river 1
- Landmark Feature Type: landmark 1
- No current Locks

1. A LockFeature Request asks for a lock on road1 and road2 with the expiry date of February 24, 2008 14:15.
2. A LockFeature Document is returned with lock id “A” reporting success. The Document lists “road.fid1” and “road.fid2” as being locked.
3. A Second LockFeature Request asks for road2, road3, river1 and landmark1 with expiry date of March 12th, 2008 6:00
4. A LockFeature Document is returned with lock id “B”. The Document lists “road.fid3”, “river.fid1” and “landmark.fid1” as obtained, and “road.fid2” as unavailable.
5. A Transaction Request asks for modification on roads1, road3, river1 and landmark1. The Transaction Request supplies the Lock Ids “A” and “B” as part of the request. The Request also says that these locks are to be released after the modification.
6. A Transaction Document is returned.

At the end of this workflow:

- Road1, road3, river1 and landmark1 have been modified
- Lock “A” still exists on road2

2.5 GeoServer Type Repository

The TypeRepository acts as a Facade for the GeoServer Application. It is used by the GeoServer operations to access low-level services. It is responsible for maintaining Application State, including Feature Locking.

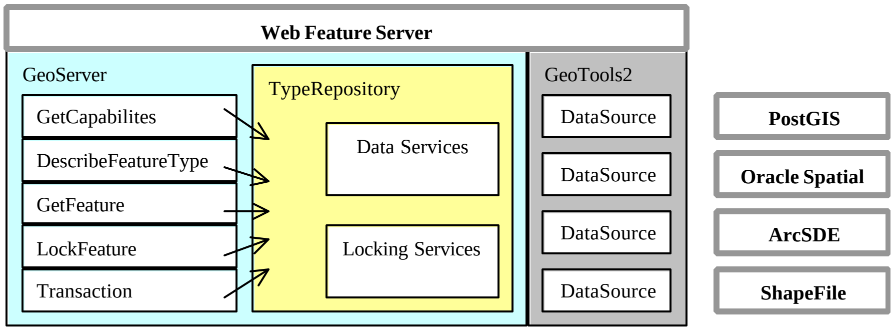


Figure 2 - GeoServer TypeRepository

All GeoServer locking schemes will need to modify the following core locking API provided by TypeRepository:

```
String addToLock( typeName, filter, lockAll, lockId)
String lock( typeName, filter, lockAll, expiry)
boolean unlock(lockId)
boolean unlock(completedTransactionRequest)
Set getLockedFeatures(lockId)
Set getNotLockedFeatures(lockId)
```

This API provides the following capabilities:

- Adding features, specified by a filter, to a new lock specified by an expiry
- Adding features, specified by a filter, to an existing lock
- Partial Locking support
- Locking Release and Partial Locking Release
- Access to both the locked and unlocked features for a given Lock Id

3 IN-PROCESS LOCKING SCHEME

Currently locking for GeoServer is performed in process. In-Process locking has the limitation of not invoking the existing database locking mechanisms.

In-Process Locking:

- Does not support persistence
- Does not provide strong transaction support

We will not be using this locking scheme.

3.1 In-Process Locks

GeoServer implements in-process locks using a class called InternalLock.

```
private class InternalLock {
     public Set getLockedFeatures();
     public Set getNotLockedFeatures();
     public String getId();
     public boolean addFeatures(boolean lockAll, List addFeatures);
     public boolean unlock(List releaseFeatures, boolean releaseAll);

     private int expiryTime;
     private String lockId;
     private Set features;
     private Set notLockedFeatures;
}
```

Internal Lock:

- Implemented as an inner class of TypeRepository
- Represents the results of one LockFeature request
- Maintains an expiry date and time
- Holds a list of locked feature Ids
- Hold a list of “unlocked” feature Ids
  (features that it tried to lock and could not acquire)

GeoServer In-Process locks are not persistent and do not offer protection against other applications concurrently modifying information.

3.2 GeoServer Integration of In-Process Locks

TypeRepository uses InternalLocks for locking support in the GeoServer layer. The locks and locked features are stored in hash maps for lookup by the TypeRepository Locking API.

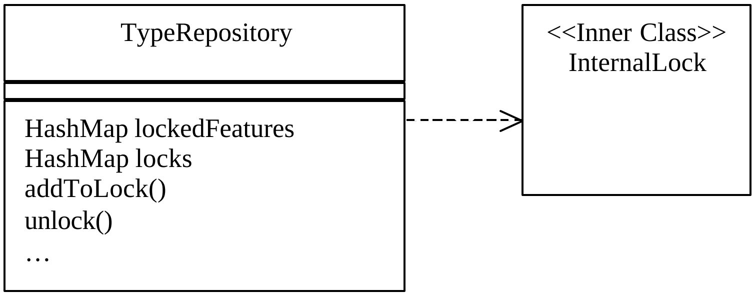


Figure 3 - In-Process Locks

The TypeRepository Locking API obtains InternalLocks instances using the data structures ‘lockedFeatures’ and ‘locks’ above.

AddToLock method example:

```
public synchronized String addToLock(String typeName,
                                     Filter filter,
                                     boolean lockAll,
                                     String lockId) throws WfsException
{
   InternalLock lock = (InternalLock)locks.get(lockId);
   lock.addFeatures(lockAll, getFidFeatures(typeName, filter));
   return lockId;
}
```

3.3 In-Process Locking Walkthrough

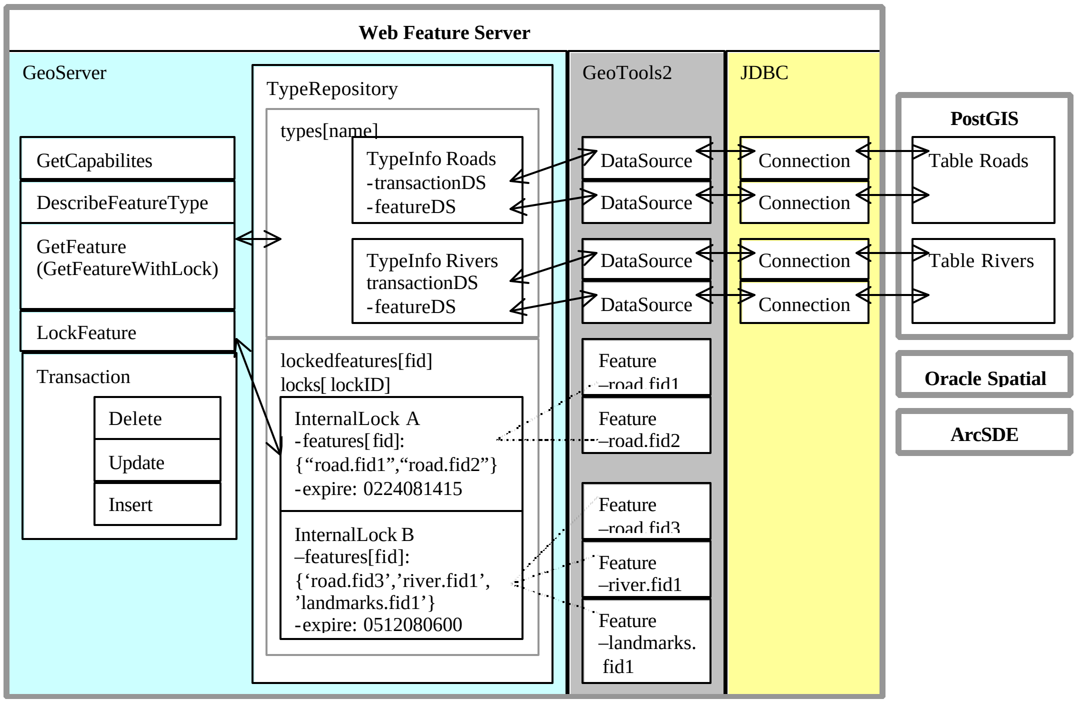


Figure 4 - Execution Diagram with In-Process Locks

1. A LockFeature Request asks for a lock on road1 and road2 with the expiry date of February 24, 2008 14:15.
   - An InternalLock “A” with expiry 0224081415 is created with road.fid1 and road.fid2.
   - InternalLock A is added to the locks Map referenced by “A”
   - InternalLock A is added to the features Map referenced by “roads.fid1” and “roads.fid2”
2. A LockFeature Document is returned with lock id “A” reporting success. The Document lists “road.fid1” and “road.fid2” as being locked.
   - GeoServer uses the InternalLock A to construct the response.
3. A Second LockFeature Request asks for road2, road3, river1 and landmark1 with expiry date of March 12th, 2008 6:00
   - An InternalLock “B” with expiry 0512080600 is created with “road.fid3”, “river.fid1” and “landmark.fid1”. The Lock records “road.fid2” as not locked.
   - InternalLock A is added to the locks Map referenced by “A”
   - InternalLock A is added to the features Map referenced by “roads.fid1” and “roads.fid2”

4. A LockFeature Document is returned with lock id “B”. The Document lists “road.fid3”, “river.fid1” and “landmark.fid1” as obtained, and “road.fid2” as unavailable.

   - GeoServer uses the InternalLock B to construct the response.

5. A Transaction Request asks for modification on roads1, road3, river1 and landmark1. The Transaction Request supplies the Lock Ids “A” and “B” as part of the request. The Request also says that these locks are to be released after the modification.

   - Modification operations are authorized with InternalLocks “A” and “B”.
   - InternalLock A releases the lock on “road.fid1”. This removed “road.fid1” from the features lookup table.
   - InternalLock B releases the locks on “road.fid3”,”river.fid1” and “landmark.fid1” and is removed from features lookup table and lock lookup table.

6. A Transaction Document is returned.

4 ROW LOCKING SCHEME

Our initial locking scheme was designed to use database row locking. Row locking is supported by may databases. Row locking provides protection against other application concurrently modifying information

Row Locking:

- Does not support persistence
- Supports strong transactions

We will not be using this locking scheme: it is complicated and does not work.

4.1 Postgis Row Locking

Postgis makes use of the SELECT * FOR UDPATE command to provide a row locking mechanism.

```
SELECT FROM roads WHERE fid=1 FOR UDPATE;
```

The problem with this approach is that the locks maintained are only available for the duration of a transaction.

```
BEGIN;
  SELECT FROM roads WHERE fid=1 FOR UDPATE;
  UPDATE roads SET name=”Road One” WHERE fid=1;
COMMIT;
```

This solution does not provide support for persistent transactions. Obtaining a row-lock is a blocking operation.

4.2 Geotools2 Row Locking Extension

The Geotools2 DataSource Class is used by GeoServer to access feature information.

The following extension would be required to support Row-Locking:

```
class TransactionalDataSource extends DataSource {
   public lockFeatures();
   public lockFeatures(Filter);
   public lockFeatures(Query);
}
```

There are no facilities to unlock Features individually – locks are only held for the duration of a transaction.

4.3 GeoServer Integration of Row Locking

Since row locks are only held for a transaction, each lock ID would have to cache a Database connection with a the row lock already performed.

A WFS Lock can lock features from multiple sources. So a Lock will have to cache several Database connections, and release them when the expiry date has been reached.

Transaction Operation issues with Row Locking:

- Transaction Operations do not use a Lock ID (the feature may be unlocked)
- Transaction Operations require partial lock release. Partial lock release will require releasing the entire lock and reacquiring a lock on a subset of features.
- Transaction Operations disseminate modifications across connections based on Lock and feature type, rather than just feature type.

The benefits to this approach would be cross DataSource locking support, which would not require additional work by DataSource providers.

AddToLock method example:

```
public String addToLock( String typeName,
                         Filter filter,
                         boolean lockAll,
                         String id){
   Lock lock = (Lock)locks.get(id);
   DataSource ds = lock.getDataSource( typeName );
   ds.addFeatures( filter );
}
```

In this limited example partial locking is not supported. Obtaining a row lock is a blocking operation.

4.4 Example Walkthrough Using Row Locks

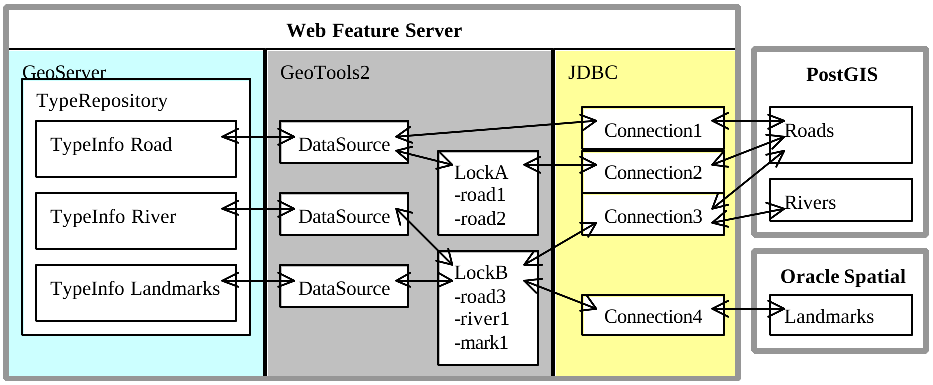


Figure 5 - Execution Diagram with Row-Locks

1. A LockFeature Request asks for a lock on road1 and road2 with the expiry date of February 24, 2008 14:15.

   - LockA is created with an expiry
   - A Connection2 is created with:

   ```
   BEGIN;
     SELECT fid=1 OR fid=2 FROM ROADS FOR UDPATE;
   ```

   - The lock will need to remember that road.fid1 and road.fid2 are locked.

2. A LockFeature Document is returned with lock id “A” reporting success. The Document lists “road.fid1” and “road.fid2” as being locked.

3. A Second LockFeature Request asks for road2, road3, river1 and landmark1 with expiry date of March 12th, 2008 6:00

   - LockB is created with an expiry
   - A Connection3 is created with: (road.fid2 already locked)

   ```
   BEGIN;
    SELECT fid=3 FROM ROADS FOR UDPATE;
    SELECT fid=1 FROM RIVER FOR UPDATE;
   ```

   - A Connection4 is created with:

   ```
   BEGIN;
    SELECT fid=1 FROM LANDMARK FOR UDPATE;
   ```

   - The lock will need to remember that road.fid3, river.fid1 and landmark.fid1 are locked.

4. A LockFeature Document is returned with lock id “B”. The Document lists “road.fid3”, “river.fid1” and “landmark.fid1” as obtained, and “road.fid2” as unavailable.

5. A Transaction Request asks for modification on roads1, road3, river1 and landmark1. The Transaction Request supplies the Lock Ids “A” and “B” as part of the request. The Request also says that these locks are to be released after the modification.

- road1 modification sent on LockA’s Connection2:

```
  UPDATE roads SET name=”Road One” WHERE fid=1;
COMMIT;
```

- road3 and river1 modification are sent on LockB’s Connection3:

```
  UPDATE roads SET name=”Road Three” WHERE fid=3;
  UPDATE river SET name=”River One” WHERE fid=1;
COMMIT;
```

- landmark1 modification are sent on LockB’s Connection4:

```
  UPDATE landmark SET name=”Landmark One” WHERE fid=1;
COMMIT;
```

- LockA reacquires a lock on road2 using Connection2:

```
BEGIN;
  SELECT fid=2 FROM ROADS FOR UDPATE;
```

6. A Transaction Document is returned.

5 LOCKING TABLE SCHEME

Our Locking Table scheme is designed to use a separate locking table. The locking table will be consulted by the database before authorizing any modification operations.

The addition of a Locking Table will need to account for database row-locking mechanisms that may be in use by other applications.

Using a Locking Table:

- Supports persistence
- Supports strong transactions

We will be using this locking scheme.

Our initial prototype will operate in conjunction with the existing GeoServer In-Process locking. Using both locking mechanisms provides a smooth upgrade path and will serve as verification of locking behavior.

5.1 Postgis Locking Table

Postgis does not directly support locking tables. This functionality will need to be added to Postgis.

Postgis Locking Table requirements:

- Require a locking table holding a tableID, rowID, Authorization code, and expiry date for each locked feature

```
CREATE TABLE locks ( table text, fid text, lock text, expires date );
```

- Require a lock checking function
- Require the lock checking function be associated with BEFORE UPDATE and BEFORE DELETE

```
CREATE TRIGGER locktrig_roads BEFORE UPDATE OR DELETE
       ON roads FOR EACH ROW
       EXECUTE PROCEDURE lockcheck('id','locks’);
```

- Require that a row lock be acquired before a locking table lock:

```
BEGIN;
     SELECT roads WHERE id=1 FOR UPDATE;
     INSERT INTO locks VALUES ('roads','1','A','2008-2-24');
COMMIT;
```

- Require a transaction provide authorization before modifying a locked row:

```
BEGIN;
     select have_lock_for('A');
     select roads where id=1 for update;
     update roads set name = 'Road One' where id=1;
COMMIT;
```

5.2 Geotools2 Locking Table Integration

The following extension to DataSource will be required to support locking:

```
class TransactionalDataSource extends DataSource {
   setCurrentLock(Lock)
   lockFeatures();
   lockFeatures(Filter);
   lockFeatures(Query);

   setAuthorization(String[])
   unLockFeatures()
   unLockFeatures(Filter)
   unLockFeatures(Query)
}
```

Where Lock a geotools2 class required to:

- Encapsulate an Authorization ID and an expiry date.
- Allows for cross DataSource Authorization IDs
- Allows for the generation of Authorization IDs to ensure uniqueness.

The Lock provided using setCurrentLock is used for subsequent lockFeature requests. When no lock is provided the class will revert to a simple Row-Locking scheme.

Authorization IDs are supplied, and are used for subsequent unLockFeature requests. Authorization IDs are also used for the DataSource addFeatures, modifyFeatures, removeFeatures and setFeatures requests.

This is a complete Locking API for Geotools2 and will not need further extension.

5.3 GeoServer Locking Table Integration

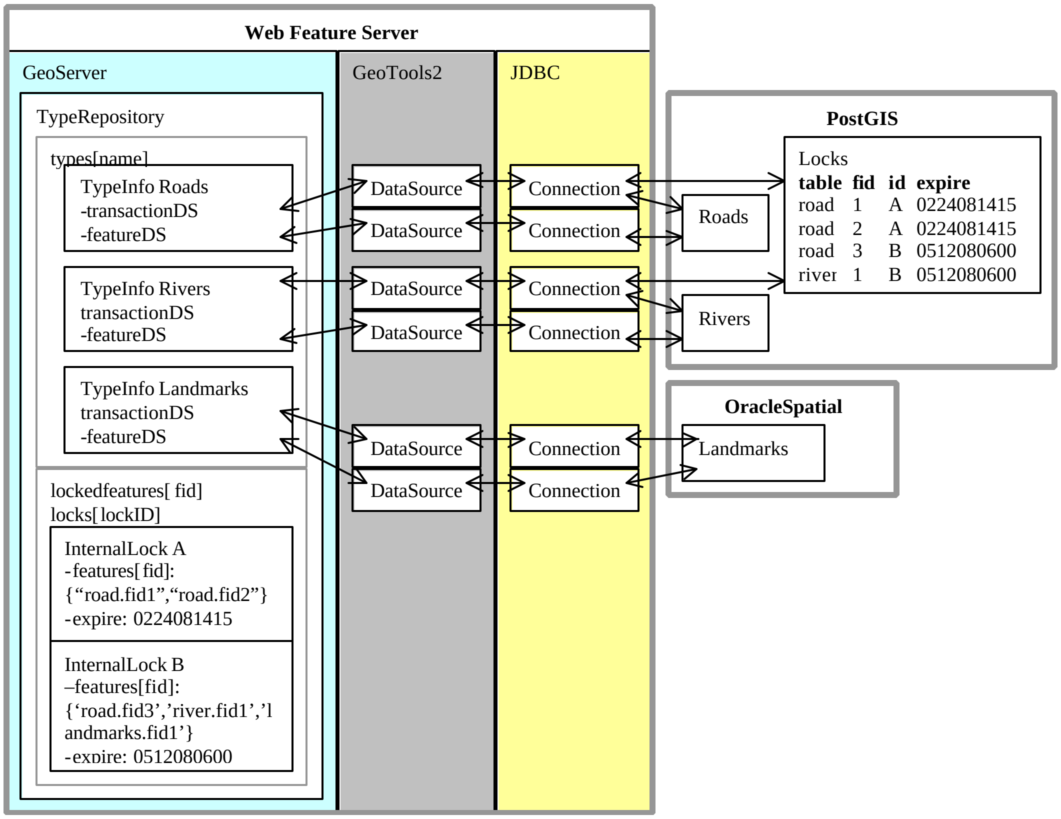
The GeoServer In-Process locking scheme will be extended to use TransactionalDataSource.

The TypeRepository Locking API will be modified to check if the features to be locked are from a TransactionalDataSource. If so the TransactionalDataSource locking API will also be used.

AddToLock method example:

```
public synchronized String addToLock(String typeName,
                                     Filter filter,
                                     boolean lockAll,
                                     String lockId) throws WfsException
{
   InternalLock lock = (InternalLock)locks.get(lockId);
   lock.addFeatures(lockAll, getFidFeatures(typeName, filter));
   DataSource ds = getType(typeName).getTransactionalDataSource();
   if( ds instanceof TransactionalDataSource){
      Transaction tds = (TransactionalDataSource) ds;
      tds.setLock( lock.getDataLock() );
      tds.addFeatures( filter );
   }
   return lockId;
}
```

5.4 Locking Table Walkthrough



Locks

table	fid	id	expire
road	1	A	0224081415
road	2	A	0224081415
road	3	B	0512080600
river	1	B	0512080600

Figure 6 - Execution Diagram with Locking Table

1. A LockFeature Request asks for a lock on road1 and road2 with the expiry date of February 24, 2008 14:15.

   - An InternalLock “A” with expiry 0224081415 is created with road.fid1 and road.fid2.

   - InternalLock A is added to TypeRepository for later reference.

   - The Postgis DataSource used for Roads is a TransactionalDataSource API. Its setLock and addFeatures methods are called:

```
 Transaction tds = (TransactionalDataSource) ds;
 tds.setLock( lock.getDataLock() );
 tds.addFeatures( filter );
```

   - The TransactionalDataSource issues the following SQL:

```
BEGIN;
     SELECT roads WHERE id=1 OR id=2 FOR UPDATE;
     INSERT INTO locks VALUES ('roads','1','A','2008-2-24')
                              ('roads','2','A','2008-2-24');
COMMIT;
```

2. A LockFeature Document is returned with lock id “A” reporting success. The Document lists “road.fid1” and “road.fid2” as being locked.

   - GeoServer uses the InternalLock A to construct the response.

3. A Second LockFeature Request asks for road2, road3, river1 and landmark1 with expiry date of March 12th, 2008 6:00

   - An InternalLock “B” with expiry 0512080600 is created with “road.fid3”, “river.fid1” and “landmark.fid1”. The Lock records “road.fid2” as not locked.
   - InternalLock A is added to TypeRepository for later reference.
   - The Postgis Roads DataSource is a TransactionalDataSource API. It’s setLock and addFeatures methods are called.
   - The Road TransactionalDataSource issues the following SQL:

     ```
     BEGIN;
          SELECT roads WHERE id=2 OR id=3 FOR UPDATE;
          INSERT INTO locks VALUES ('roads','2','B','2008-5-12')
                                   ('roads','3','B','2008-5-12');
     COMMIT;
     ```

     This SQL query returns (0,1) indicating that the road 2 lock was not obtained.

   - The Postgis River DataSource is a TransactionalDataSource API and its setLock and addFeatures methods are called.
   - The River TransactionalDataSource issues the following SQL:

     ```
     BEGIN;
          SELECT river WHERE id=1 FOR UPDATE;
          INSERT INTO locks VALUES ('river','1','A','2008-5-12');
     COMMIT;
     ```

   - The Oracle DataSource is not a TransactionalDataSource.

4. A LockFeature Document is returned with lock id “B”. The Document lists “road.fid3”, “river.fid1” and “landmark.fid1” as obtained, and “road.fid2” as unavailable.

   - GeoServer uses the InternalLock B to construct the response.

5. A Transaction Request asks for modification on roads1, road3, river1 and landmark1. The Transaction Request supplies the Lock Ids “A” and “B” as part of the request. The Request also says that these locks are to be released after the modification.

   - Modification operations are authorized with InternalLocks “A” and “B”.
   - InternalLock A releases the lock on “road.fid1”. This removed “road.fid1” from the features lookup table.
   - InternalLock B releases the locks on “road.fid3”,”river.fid1” and “landmark.fid1” and is removed from features lookup table and lock lookup table.

6. A Transaction Document is returned.

6 GeoServer and GeoTools2 Modifications

The Locking Table Scheme described previously is complete and offers conformance with the Locking specification provided by the Open GIS Consortium.

The GeoServer/GeoTools2 integration plan is:

1. Implement a Postgis TransactionalDataSource
2. Implement the Table Locking in duplication of In-Process Locking.
3. Choose Table Locking for supporting DataSources, with In-Process locking as a fallback for DataSources that cannot support locking.
4. Move the In-Process Locking code to Geotools2 as a facility that DataSource that cannot support locking can use.

At the end of this process all DataSource implementations will some form of locking. In addition, databases will have an opportunity to provide seamless strong transaction support.

6.1 Prototype Locking Table Scheme

The initial prototype will operate in conjunction with the existing GeoServer In-Process locking scheme. The Prototype's functionality, and required modifications, are adequately documented in "Section 5: Locking Table Scheme" where it is used as a reference implementation.

The prototype provides true long transaction support for GeoServer and a robust locking API to GeoTools2.

AddToLock method example:

```
public synchronized String addToLock(String typeName,
                                     Filter filter,
                                     boolean lockAll,
                                     String lockId) throws WfsException
{
   InternalLock lock = (InternalLock)locks.get(lockId);
   lock.addFeatures(lockAll, getFidFeatures(typeName, filter));
   DataSource ds = getType(typeName).getTransactionalDataSource();
   if( ds instanceof TransactionalDataSource){
      Transaction tds = (TransactionalDataSource) ds;
      tds.setLock( lock.getDataLock() );
      tds.addFeatures( filter );
   }
   return lockId;
}
```

The prototype will also require the addition of TransactionalDataSource to the geotools2 library. The complete API is listed in Appendix A.

6.2 Transition Locking Table Scheme

The Transition Locking Table Scheme focuses on eliminating duplicated effort within GeoServer.

AddToLock method example:

```
public synchronized String addToLock(String typeName,
                                     Filter filter,
                                     boolean lockAll,
                                     String lockId) throws WfsException
{
   DataSource ds = getType(typeName).getTransactionalDataSource();
   if( ds instanceof TransactionalDataSource){
      Transaction tds = (TransactionalDataSource) ds;
      tds.setLock( getDataLock( lockId ) );
      tds.addFeatures( filter );
   }
   else {
      InternalLock lock = (InternalLock)locks.get(lockId);
      lock.addFeatures(lockAll, getFidFeatures(typeName, filter));
   }
   return lockId;
}
```

The Transition Locking Table Scheme eliminates duplication of effort and provides GeoServer with a more efficient implementation.

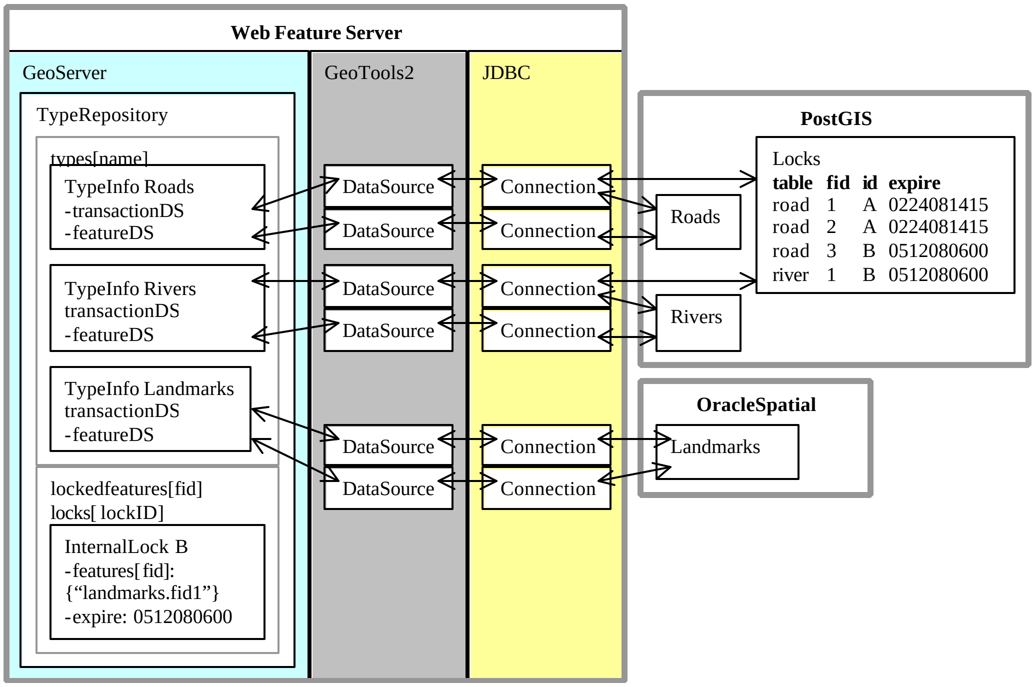


table	fid	id	expire
road	1	A	0224081415
road	2	A	0224081415
road	3	B	0512080600
river	1	B	0512080600

Figure 7 - Execution Diagram with Transition Locking Scheme

6.3 Final Locking Table Scheme

The final stage will move all locking code to the TransactionalDataSource API. In order to accomplish this we will need to move the GeoServer In-Process locking implementation into the geotools2 library. This allows DataSource implementations, such as Shapefile, that cannot natively support locking the ability to implement TransactionalDataSource.

AddToLock method example:

```
public synchronized void addToLock(String typeName,
                                   Filter filter,
                                   boolean lockAll,
                                   Lock lock ) throws WfsException
{
   DataSource ds = getType(typeName).getTransactionalDataSource();
   ds.setLock( lock );
   ds.addFeatures( filter );
}
```

The GeoServer implementation becomes simplified as locking is now completely delegated.

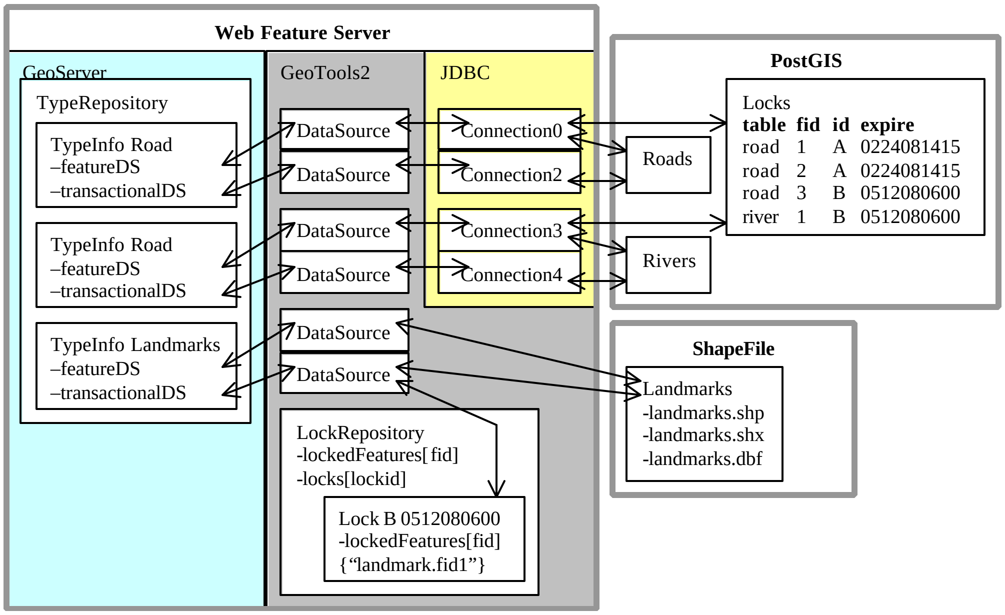


table	fid	id	expire
road	1	A	0224081415
road	2	A	0224081415
road	3	B	0512080600
river	1	B	0512080600

Figure 8 - Execution Diagram with Final Locking Scheme

7 References

Web Feature Service Implementation Specification, http://www.opengis.org/techno/specs/02-058.pdf

Martin Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, 2003

APPENDIX A: GEOTOOLS2 TRANSACTIONALDATASOURCE

TransactionDataSource.java

```
package org.geotools.data;

import org.geotools.filter.Filter;

public interface TransactionalDataSource extends DataSource {
  public void setCurrentLock( Lock lock );
  public void lockFeatures( Query query ) throws DataSourceException;
  public void lockFeatures( Filter filter) throws DataSourceException;
  public void lockFeatures() throws DataSourceException;
  public void setAuthorization( String[] locksIds );
  public void unLockFeatures() throws DataSourceException;
  public void unLockFeatures( Filter filter ) throws DataSourceException;
  public void unLockFeatures( Query query ) throws DataSourceException;
}
```

This extension is required to provide locking support to DataSource.

setCurrentLock

```
public void setCurrentLock( Lock lock );
```

All locking operations will operate against the provided Lock.

This in in keeping with the stateful spirit of DataSource in which operations are issued against the "current" transaction. If a Lock is not provided lock operations will only be applicable for the current transaction (they will expire on the next commit or rollback).

That is lockFeatures() operations will:

- Be recorded against the provided Lock
- Be recorded against the current transaction if no Lock is provided.

Calling this method with setCurrentLock(null), will revert to per transaction operation.

This design allows for the following:

- cross DataSource Lock usage
- not having pass in the same Lock object multiple times

setFeatures

```
public void lockFeatures( Query query ) throws DataSourceException;
```

Lock features described by Query.

This is an All or nothing operation - to implement partial Locking retrieve the features with a query operation first before trying to lock them individually.

```
public void lockFeatures( Filter filter) throws DataSourceException;
```

Lock features described by Filter.

This is an All or nothing operation - to implement partial Locking retrieve the features with a Filter operation first before trying to lock them individually.

```
public void lockFeatures() throws DataSourceException;
```

Lock all Features.

The method does not prevent addFeatures() from being used (we could add a lockDataSource() method if this functionality is required

setAuthorization

```
public void setAuthorization( String[] locksIds );
```

Provides a set of Lock ids this Transaction can use for authorization.

All proceeding modifyFeatures and removeFeatures operations will make use of the provided authorization.

That is operations will only succeed if affected features are:

- unlocked
- locked with one of the provided lockIds

LockIds are provided as a String, rather than Lock objects, to account for across process lock use.

Authorization must be provided before calling:

- unLockFeatures
- modifyFeatures
- addFeatures
- removeFeatures

Example:

```
void releaseLock( String lockId, TransactionDataSource tds ){
   tds.setAuthorization( new String[]{"LOCK1234","LOCK534"} );
   tds.unLockFeatures();
}
```

unLockFeatures

```
public void unLockFeatures() throws DataSourceException;
```

Unlocks all Features. Authorization must be provided prior before calling this method.

```
public void unLockFeatures( Filter filter ) throws DataSourceException;
```

Unlock Features denoted by provided filter. Authorization must be provided prior before calling this method.

```
public void unLockFeatures( Query query ) throws DataSourceException;
```

Unlock Features denoted by provided query. Authorization must be provided prior before calling this method.

Lock.java

```
public class Lock {
    public String getID(){
        return id;
    }
    public long getExpire(){
        return date.getTime();
    }
    static public Lock create( int expire );
}
```

This class is responsible for supplying a unique ID for TransactionDataSource locking