Assignment 5 Solution

Version Locking

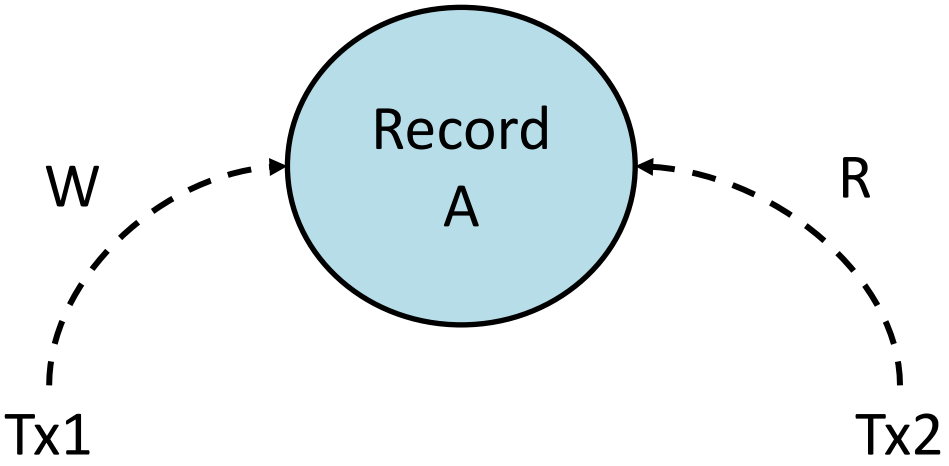
Database Systems

DataLab, CS, NTHU

Spring, 2018

2V2PL

- Acquire **shadow exclusive lock** before writing
- Acquire shared lock before reading
- Shadow exclusive lock and shared lock are **compatible**



LockTable

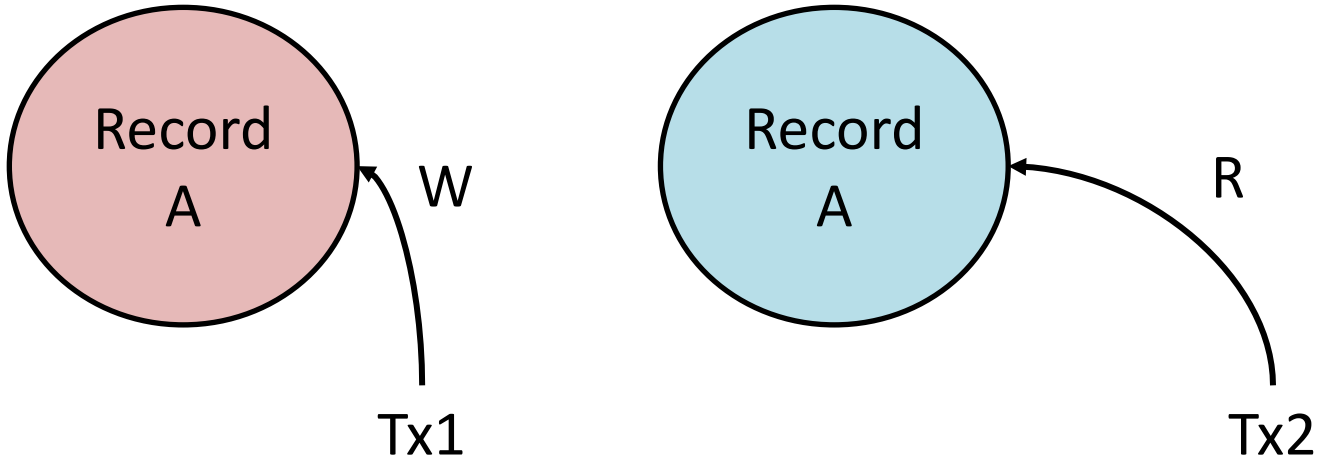
```
class Lockers {
    Set<Long> sLockers, ixLockers, isLockers, requestSet;
    // only one tx can hold xLock(sixLock) on single item
    long sixLocker, xLocker, shadowXLocker;
```

add a shadow exclusive lock

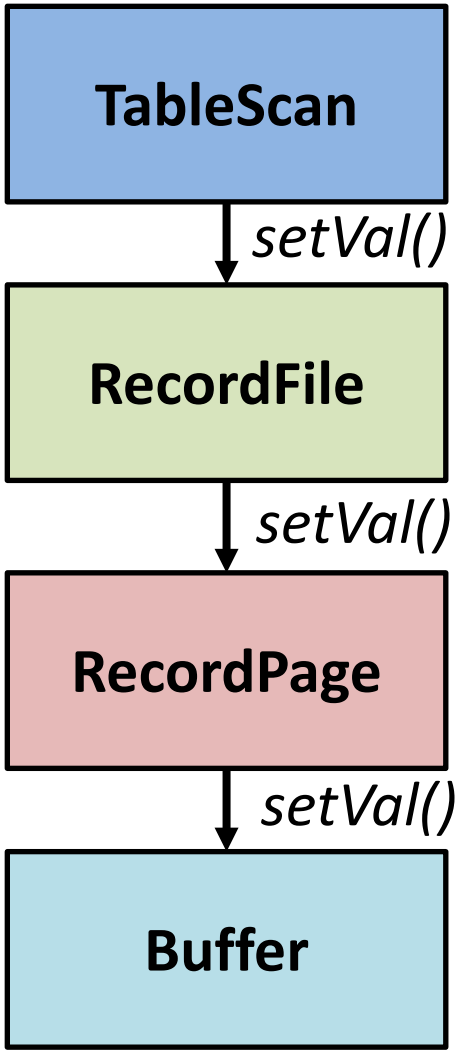
```
private boolean shadowXLockable(Lockers lks, long txNum) {
    return (!sixLocked(lks) || hasSixLock(lks, txNum))
            && (!ixLocked(lks) || isTheOnlyIxLocker(lks, txNum))
            && (!xLocked(lks) || hasXLock(lks, txNum))
            && (!shadowXLocked(lks) || hasShadowXLock(lks, txNum));
}
```

Shadow Modification

- Modify records in private workspace



update in Tx1's private workspace


TableScan
setVal()
RecordFile
setVal()
RecordPage
setVal()
Buffer

RecordFile

```
public void setVal(String fldName, Constant val) {
    if (tx.isReadOnly() && !isTempTable())
        throw new UnsupportedOperationException();
    Type fldType = ti.schema().type(fldName);

    Constant v = val.castTo(fldType);
    if (Page.size(v) > Page.maxSize(fldType))
        throw new SchemaIncompatibleException();
    if (!tx.certified() && !isTempTable()) {
        tx.concurrencyMgr().shadowModifyRecord(currentRecordId());
        tx.putVal(ti.tableName(), currentRecordId(), fldName, v);
    } else {
        rp.setVal(fldName, v);
    }
}
```

path 1: modify in private workspace

RecordFile

```
public void setVal(String fldName, Constant val) {
    if (tx.isReadOnly() && !isTempTable())
        throw new UnsupportedOperationException();
    Type fldType = ti.schema().type(fldName);

    Constant v = val.castTo(fldType);
    if (Page.size(v) > Page.maxSize(fldType))
        throw new SchemaIncompatibleException();
    if (!tx.certified() && !isTempTable()) {
        tx.concurrencyMgr().shadowModifyRecord(currentRecordId());
        tx.putVal(ti.tableName(), currentRecordId(), fldName, v);
    } else {
        rp.setVal(fldName, v);
    }
}
```

a transaction is certified if it can acquire all the required exclusive locks

```
public void setVal(String fldName, Constant val) {
    if (tx.isReadOnly() && !isTempTable())
        throw new UnsupportedOperationException();
    Type fldType = ti.schema().type(fldName);

    Constant v = val.castTo(fldType);
    if (Page.size(v) > Page.maxSize(fldType))
        throw new SchemaIncompatibleException();
    if (!tx.certified() && !isTempTable()) {
        tx.concurrencyMgr().shadowModifyRecord(currentRecordId());
        tx.putVal(ti.tableName(), currentRecordId(), fldName, v);
    } else {
        rp.setVal(fldName, v);
    }
}
```

RecordFile

path 2: in-place modify

```
public Constant getVal(String fldName) {
    Constant val;
    val = tx.getVal(ti.tableName(), currentRecordId(), fldName);
    if (val != null)
        return val;
    return rp.getVal(fldName);
}
```

RecordFile

handling read-after-write for consistency

Private Workspace

- Create a per-transaction private workspace: *HashMap*
  - key : RecordId (which record) + String (which field)
  - value : Constant (new value)

Private Workspace

```
public Transaction(TransactionMgr txMgr, TransactionLifecycleListener
        TransactionLifecycleListener recoveryMgr, TransactionLifecycle
        long txNum) {
    this.concurMgr = (ConcurrencyMgr) concurMgr;
    this.recoveryMgr = (RecoveryMgr) recoveryMgr;
    this.bufferMgr = (BufferMgr) bufferMgr;
    this.txNum = txNum;
    this.readOnly = readOnly;
    this.workspace = new HashMap<RecordField, Constant>();
    this.certified = false;
```

Transaction

Private Workspace

```
public Transaction(TransactionMgr txMgr, TransactionLifecycleListener
        TransactionLifecycleListener recoveryMgr, TransactionLifecycle
        long txNum) {
    this.concurMgr = (ConcurrencyMgr) concurMgr;
    this.recoveryMgr = (RecoveryMgr) recoveryMgr;
    this.bufferMgr = (BufferMgr) bufferMgr;
    this.txNum = txNum;
    this.readOnly = readOnly;
    this.workspace = new HashMap<RecordField, Constant>();
    this.certified = false;
```

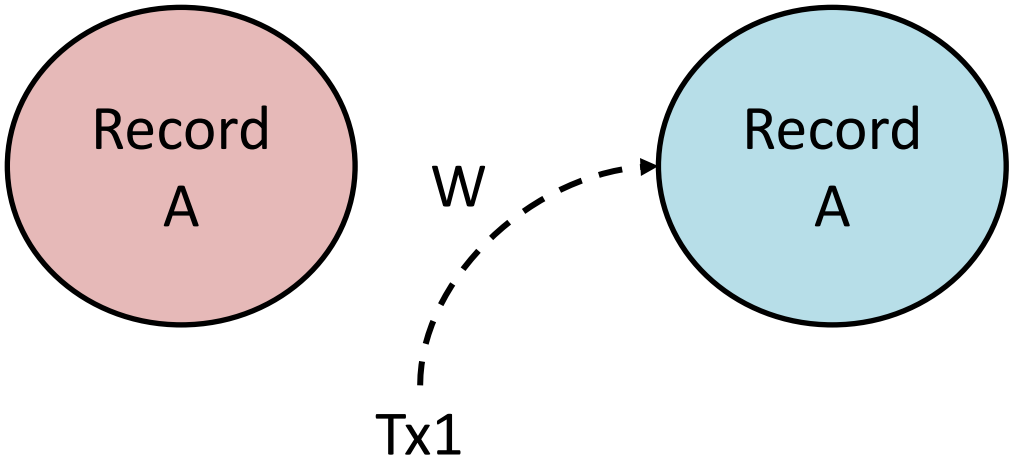
Transaction

```
public void putVal(String tblName, RecordId rid, String fldName, Constant val) {
    workspace.put(new RecordField(tblName, rid, fldName), val);
}

public Constant getVal(String tblName, RecordId rid, String fldName) {
    return workspace.get(new RecordField(tblName, rid, fldName));
}
```

Upgrade to Exclusive Lock

- Before a transaction commit, it needs to acquire exclusive lock for every record it modified



Transaction

```
public void commit() {
    upgradeWriteLock();
    certify();
    commitWorkspace();
    for (TransactionLifecycleListener l : lifecycleListeners)
        l.onTxCommit(this);

    if (logger.isLoggable(Level.FINE))
        logger.fine("transaction " + txNum + " committed");
}
private void upgradeWriteLock() {
    for (RecordField rf: workspace.keySet())
        this.concurMgr.modifyRecord(rf.rid);
}
```

Transaction

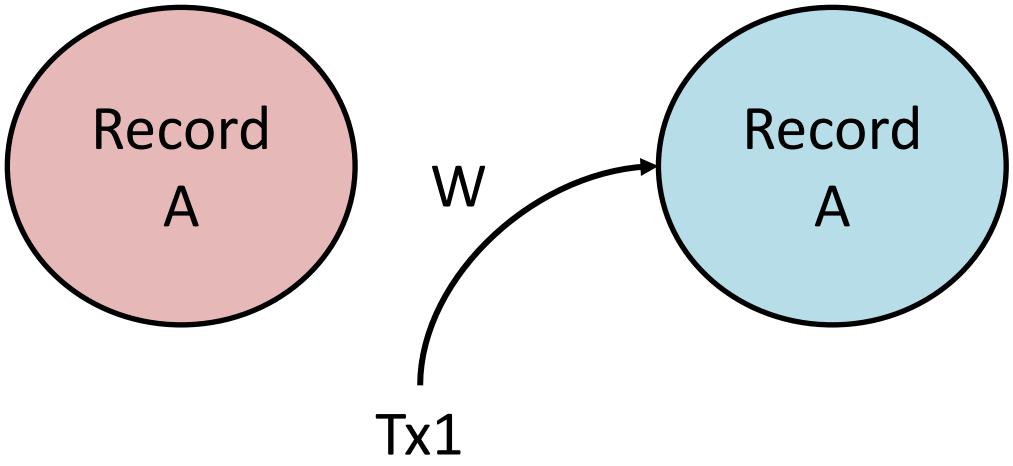
```
public void commit() {
    upgradeWriteLock();
    certify();
    commitWorkspace();
    for (TransactionLifecycleListener l : lifecycleListeners)
        l.onTxCommit(this);

    if (logger.isLoggable(Level.FINE))
        logger.fine("transaction " + txNum + " committed");
}
```

```
public void certify() {
    this.certified = true;
}
```

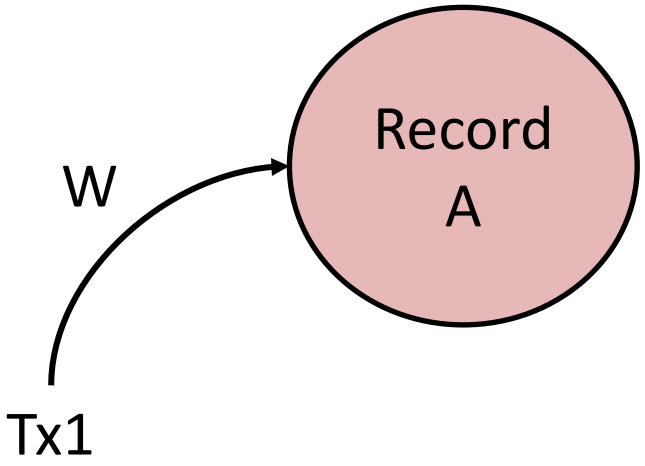
In-Place Update

- After the tx collecting all the required exclusive locks, copy the content in its private workspace to the public table



In-Place Update

- After the tx collecting all the required exclusive locks, copy the content in its private workspace to the public table



```
public void commit() {
    upgradeWriteLock();
    certify();
    commitWorkspace();
    for (TransactionLifecycleListener l : lifecycleListeners)
        l.onTxCommit(this);

    if (logger.isLoggable(Level.FINE))
        logger.fine("transaction " + txNum + " committed");
}
```

Transaction

```
private void commitWorkspace() {
    for (Map.Entry<RecordField, Constant> entry: workspace.entrySet()) {
        RecordField rfield = entry.getKey();
        Constant val = entry.getValue();
        TableInfo ti = VanillaDb.catalogMgr().getTableInfo(rfield.tblName, this);
        RecordFile rfile = ti.open(this, true);
        rfile.moveToRecordId(rfield.rid);
        rfile.setVal(rfield.fldName, val);
        rfile.close();
    }
}
```

get a RecordFile of the table to be modified and move to the specified position

Transaction

```
private void commitWorkspace() {
    for (Map.Entry<RecordField, Constant> entry: workspace.entrySet()) {
        RecordField rfield = entry.getKey();
        Constant val = entry.getValue();
        TableInfo ti = VanillaDb.catalogMgr().getTableInfo(rfield.tblName, this);
        RecordFile rfile = ti.open(this, true);
        rfile.moveToRecordId(rfield.rid);
        rfile.setVal(rfield.fldName, val);
        rfile.close();
    }
}
```

modify through the original path

Transaction

```
private void commitWorkspace() {
    for (Map.Entry<RecordField, Constant> entry: workspace.entrySet()) {
        RecordField rfield = entry.getKey();
        Constant val =
        TableInfo ti =
        RecordFile rfil
        rfile.moveToRec
        rfile.setVal(rf
        rfile.close();
    }
}
```

RecordFile

```
public void setVal(String fldName, Constant val) {
    if (tx.isReadOnly() && !isTempTable())
        throw new UnsupportedOperationException();
    Type fldType = ti.schema().type(fldName);

    Constant v = val.castTo(fldType);
    if (Page.size(v) > Page.maxSize(fldType))
        throw new SchemaIncompatibleException();
    if (!tx.certified() && !isTempTable()) {
        tx.concurrencyMgr().shadowModifyRecord(currentRecordId());
        tx.putVal(ti.tableName(), currentRecordId(), fldName, v);
    } else {
        rp.setVal(fldName, v);
    }
}
```