15-410 Operating Systems

Atomic Transactions

December 1, 2008

Jeffrey L. Eppinger
Professor of the Practice
School of Computer Science

So Who *Is* This Guy?

Jeff Eppinger (eppinger@cmu.edu, EDSH 229)

- Ph.D. Computer Science (CMU 1988)
- Asst Professor of Computer Science (Stanford 1988-1989)
- Co-founder of Transarc Corp. (Bought in 1994 by IBM)
  - Transaction Processing Software
  - Distributed File Systems Software
- IBM Faculty Loan to CMU eCommerce Inst. (1999-2000)
- Joined SCS Faculty in 2001
- Lecture Style: ¿Questioning?

What Do Transactions Do?

- They ensure the ***consistency*** of data
  - In the face of ***concurrency***
  - In the face of ***failure***
- They ***improve performance***
  - In many cases
    - In many common cases
  - But not always

Do You Do ACID?

- What is ACID?
- The ACID properties are the guarantees provided by the transaction system:
  - Atomicity: all or none
  - Consistency: if consistent before transaction, so too after
  - Isolation: despite concurrent execution, $\exists$ serial ordering
  - Durability: committed transaction cannot be undone

When Are Transactions Used?

- When you use:
  - File Systems
    - Remember fsck, chkdsk, scandisk?
    - Before File Systems used transactions it could take hours for a large file system to recover from a crash
  - Databases
  - Applications build on databases
    - Banking Applications
    - Web Applications
    - BeanFactory

Who Invented Atomic Transactions?

- The guys that built TP Monitors
- Most notable advocate: Jim Gray
  - The guru of transactions systems
  - Berkeley, Ph.D.
  - Famously worked at IBM, then Tandem, finally Microsoft
  - Presumed lost at sea in January 2007
  - Wrote the bible on transaction systems:

    Transaction Processing: Concepts and Techniques, 1992

Outline

- ✓ ***What*** Do Transactions Do?
- ✓ ***When*** Are Transactions Used?
- ✓ ***Who*** Invented Atomic Transactions?
- ➢ ***How***
  - – How do you use transactions?
  - – How do you implement them?

How do I use transactions?

```
public void deposit(int acctNum, double amount)
    throws RollbackException
{
    Transaction.begin();
    Acct a = acctFactory.lookup(acctNum);
    a.setBalance(a.getBalance()+amount);
    Transaction.commit();
}
```


Accounts are JavaBeans

```
public class Acct {
    private int    acctNum;
    private double balance;

    public Acct(int acctNum) { this.acctNum = acctNum; }

    public int    getAcctNum()  { return acctNum; }
    public double getBalance()  { return balance; }

    public void setBalance(double x) { balance = x; }
}
```

BeanFactory

```
public interface BeanFactory<B> {
    public B    create(Object... priKeyValues) throws RollbackExce…
    public void delete(Object... priKeyValues) throws RollbackExce…
    public int  getBeanCount()                 throws RollbackExce…
    public B    lookup(Object... priKeyValues) throws RollbackExce…
    public B[]  match(MatchArg... constaints)  throws RollbackExce…
    …
}
```

- BeanFactory uses Java Reflection to obtain the bean properties
- Methods throw RollbackException in case of any failure
  - (The transaction is rolled back before throwing the exception)
- BeanFactory implementations use the Abstract Factory pattern
  - There are multiple implementations of BeanFactory:
    - Using a relational database
    - Using files

Transactions

- Transactions are associated with threads
- When called in a transaction, beans returned by create(), lookup(), and match() are tracked and their changes are “saved” at commit time

```
public class Transaction {
    public static void begin()  throws RollbackException {…}
    public static void commit() throws RollbackException {…}
    public static boolean isActive() {…}
    public static void rollback() {…}
}
```

The classic debit/credit example

```
public void xfer(int fromAcctNum,
                 int toAcctNum,
                 double amount) throws RollbackException
{
    Transaction.begin();
    Acct t = acctFactory.lookup(toAcctNum);
    t.setBalance(t.getBalance()+amount);
    Acct f = acctFactory.lookup(fromAcctNum);
    f.setBalance(f.getBalance()-amount);
    Transaction.commit();
}
```

- Error cases not addressed (acct not found, low balance)

Remember the ACID Properties?

- ✓ Atomicity: all or none
- ✓ Consistency: if before than after
- ✓ Isolation: serial ordering
- ✓ Durability: cannot be undone

```
public void xfer(int fromAcctNum,
                 int toAcctNum,
                 double amount)
    throws RollbackException
{
    Transaction.begin();
    Acct t = acctFactory.lookup(toAcctNum);
    t.setBalance(t.getBalance()+amount);
    Acct f = acctFactory.lookup(fromAcctNum);
    f.setBalance(f.getBalance()-amount);
    Transaction.commit();
}
```

How Are ACID Properties Enforced?

- A *simple, low-performance* implementation
  - One (CSV) file holds contains all the data
  - ***Atomicity*** – write a new file and then use rename to replace old version (slow)
  - ***Consistency*** – app's problem
  - ***Isolation*** – locking w/ one mutex (slow)
  - ***Durability*** – trust the file system (weak)

How Are ACID Properties Enforced?

- A *high-performance* implementation
  - Complex disk data structures (B-trees in MySQL)
  - ***Atomicity*** – write-ahead logging
  - ***Consistency*** – app's problem
  - ***Isolation*** – two-phase locking
  - ***Durability*** – write-ahead logging

Write-ahead Logging

- Provides atomicity & durability
- Buffer database disk pages in a memory buffer cache
- Log (on disk) all changes to DB before they are written (out to disk)
  - When changing data pages, queue (to log) records that describe changes
  - When committing, queue "commit-record" into log, flush log (to disk)
  - Before writing out cached DB pages, ensure relevant log recs flushed
- Recover from the log
  - When restarting after a failure, scan the log:
    - (Case 1) Redo transactions with commit records, as necessary
    - (Case 2) Undo transactions without commit records, as necessary
  - When handling user or system initiated rollbacks:
    - (Case 3) Scan the log and undo all the work

How Do You Describe Changes?

- Value Logging
  - E.g., old value = 4, new value = 5
- Operation Logging
  - E.g., increment by 1,
  - E.g., insert file 436 into directory 123

`Transaction.begin();`

Sample Log

Disk Storage

<toAcctNum>
`balance:  $3`

<fromAcctNum>
`balance:$100`

Memory Buffer Cache

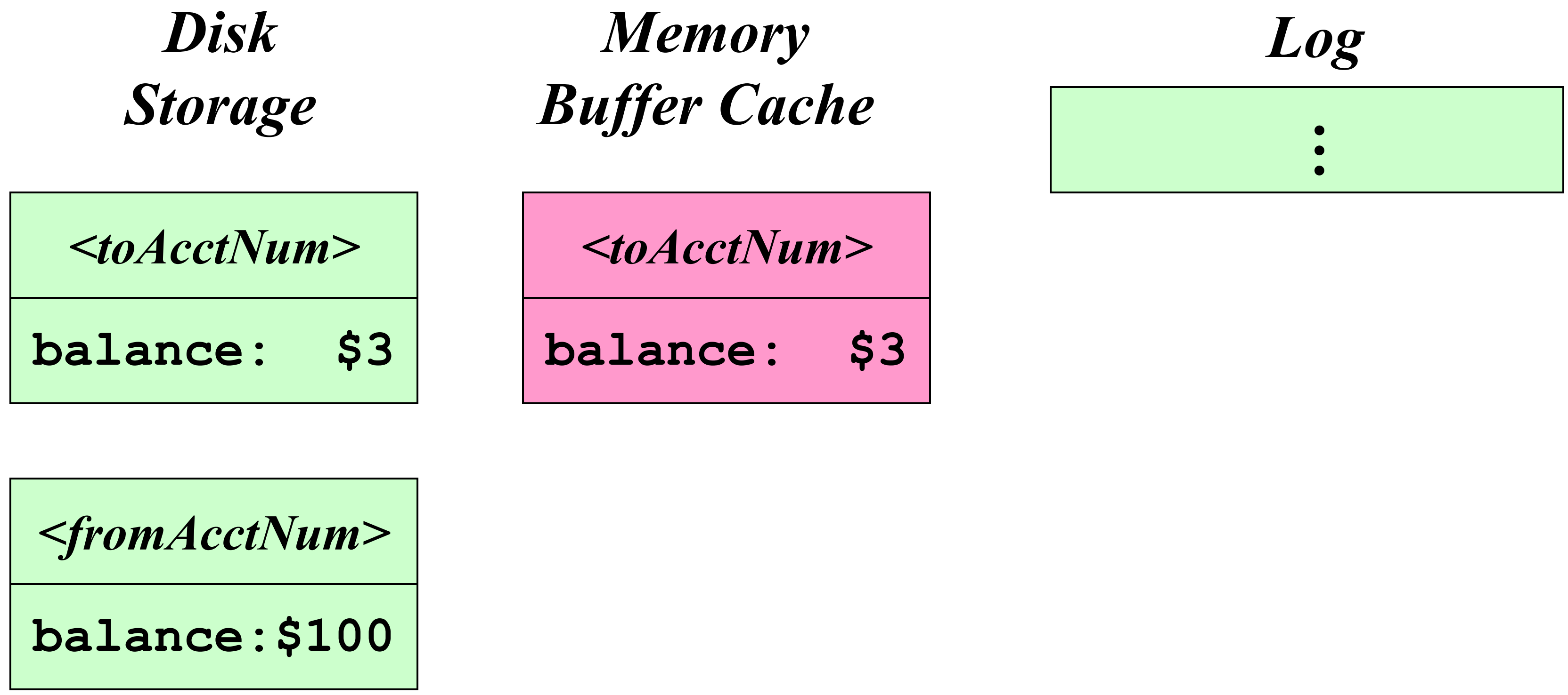
Log

⋮

Green log records have been flushed to disk

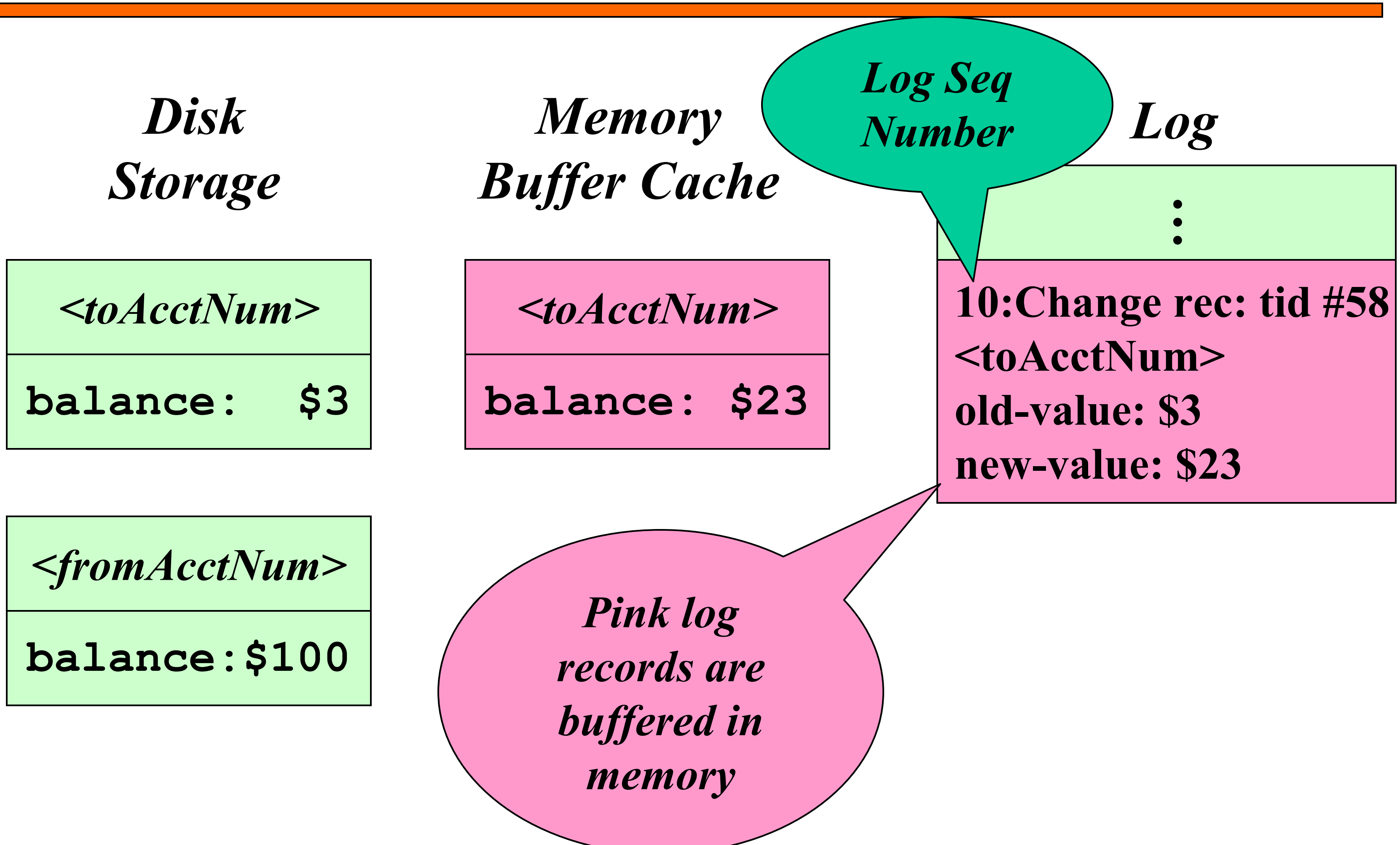
```
Transaction.begin();
Acct t = factory.lookup(toAcctNum);
…t.getBalance()…
```

Sample Log



```
Transaction.begin();
Acct t = factory.lookup(toAcctNum);
t.setBalance(t.getBalance()+20);
```

Sample Log



Sample Log

```
Transaction.begin();
Acct t = factory.lookup(toAcctNum);
t.setBalance(t.getBalance()+20);
Acct f = factory.lookup(fromAcctNum);
…f.getBalance()…
```

Disk Storage

<toAcctNum>

`balance:  $3`

<fromAcctNum>

`balance:$100`

Memory Buffer Cache

<toAcctNum>

`balance: $23`

<fromAcctNum>

`balance:$100`

Log

⋮

10:Change rec: tid #58
<toAcctNum>
old-value: $3
new-value: $23

Sample Log

```
Transaction.begin();
Acct t = factory.lookup(toAcctNum);
t.setBalance(t.getBalance()+20);
Acct f = factory.lookup(fromAcctNum);
f.setBalance(f.getBalance()-20);
```

Disk Storage

<toAcctNum>
`balance:  $3`

<fromAcctNum>
`balance:$100`

Memory Buffer Cache

<toAcctNum>
`balance: $23`

<fromAcctNum>
`balance: $80`

Log

⋮

10:Change rec: tid #58
<toAcctNum>
old-value: $3
new-value: $23

⋮

12:Change rec: tid #58
<fromAcctNum>
old-value: $100
new-value: $80

Sample Log

```
Transaction.begin();
Acct t = factory.lookup(toAcctNum);
t.setBalance(t.getBalance()+20);
Acct f = factory.lookup(fromAcctNum);
f.setBalance(f.getBalance()-20);
Transaction.commit();
```

Disk Storage

<toAcctNum>
`balance: $3`

<fromAcctNum>
`balance:$100`

Memory Buffer Cache

<toAcctNum>
`balance: $23`

<fromAcctNum>
`balance: $80`

Log

⋮
10:Change rec: tid #58 <toAcctNum> old-value: $3 new-value: $23
⋮
12:Change rec: tid #58 <fromAcctNum> old-value: $100 new-value: $80
13:Commit: tid #58

To Commit:

1) Append "Commit" rec.

2) Flush log buffer

¡Performance Improvement!

- You do not need to flush the memory buffer cache to commit a transaction
  - Only need to flush the buffered log records
  - Great locality…all those disparate buffer cache data pages can be written out later…writes of hot pages will contain changes from many transactions
- All transactions share one log
  - You can commit several transactions with one log write
- The log is append only and rarely read
  - So it's very efficient to write…great locality
  - Optimizations abound for increasing throughput

Recovery after System Failure: *Crash after commit* (Case 1)

Disk Storage
<toAcctNum>
`balance:  $3`

<fromAcctNum>
`balance:$100`

Memory Buffer Cache

Log
⋮
10:Change rec: tid #58 <toAcctNum> old-value: $3 new-value: $23
⋮
12:Change rec: tid #58 <fromAcctNum> old-value: $100 new-value: $80
13:Commit: tid #58

Recovery after System Failure: Redo committed transactions (Case 1)

Disk Storage

<toAcctNum>

`balance: ~~$3~~` *23*

<fromAcctNum>

`balance: ~~$100~~` *80*

Memory Buffer Cache

Log

⋮

10:Change rec: tid #58
<toAcctNum>
old-value: $3
new-value: $23

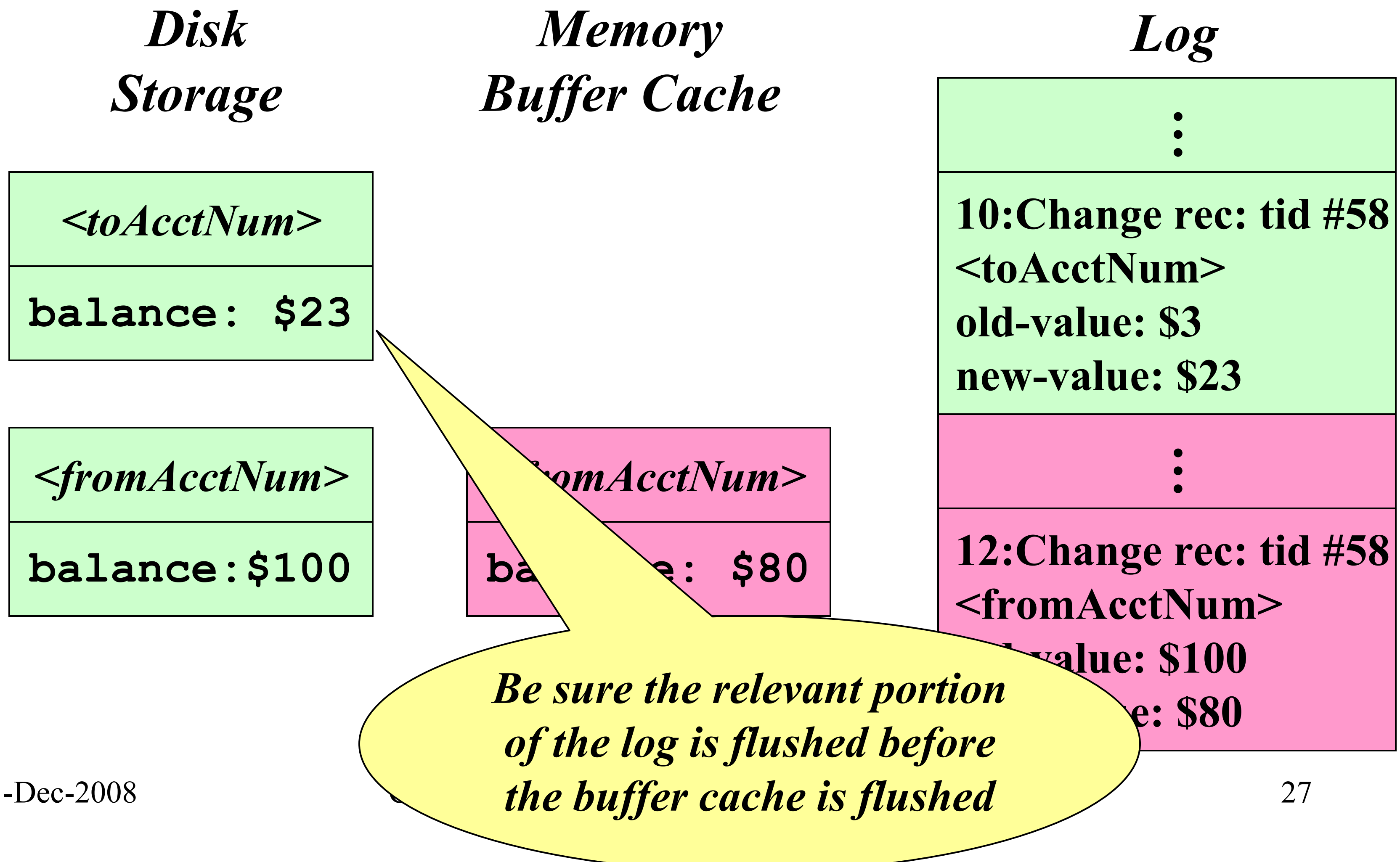
⋮

12:Change rec: tid #58
<fromAcctNum>
old-value: $100
new-value: $80

13:Commit: tid #58

Buffer Cache Can Be Flushed Mid-Transaction

```
Transaction.begin();
Acct t = factory.lookup(toAcctNum);
t.setBalance(t.getBalance()+20);
Acct f = factory.lookup(fromAcctNum);
f.setBalance(f.getBalance()-20);
```



Recovery after System Failure: Undo partial work of uncommitted transactions (Case 2)

Disk Storage

<toAcctNum>
`balance: $23` ~~$23~~ *3*

<fromAcctNum>
`balance:$100`

Memory Buffer Cache

Log

⋮

10:Change rec: tid #58
<toAcctNum>
old-value: $3
new-value: $23

Rollback using the log (Case 3)

```
Transaction.begin();
Acct t = factory.lookup(toAcctNum);
t.setBalance(t.getBalance()+20);
Acct f = factory.lookup(fromAcctNum);
f.setBalance(f.getBalance()-20);
Transaction.rollback();
```

Disk Storage
<toAcctNum>
`balance:  $3`
<fromAcctNum>
`balance:$100`

Memory Buffer Cache	
<toAcctNum>	
`balance:` ~~`$23`~~	*3*
<fromAcctNum>	
`balance:` ~~`$80`~~	*100*

Log
⋮
10:Change rec: tid #58 <toAcctNum> old-value: $3 new-value: $23
⋮
12:Change rec: tid #58 <fromAcctNum> old-value: $100 new-value: $80

What else is in the log?

- You cannot afford to process the whole log at system restart
  - You need to come up quickly
- Many optimizations and special cases
  - Periodically checkpoint records are written describing the state of the buffer cache
  - Rollback records written to the log
  - Long running transactions are rolled back
  - Storing Log Sequence Numbers (LSNs) on data pages
  - Page flush records written to the log

How Are ACID Properties Enforced?

- ✓ ***Atomicity*** – write-ahead logging
- ✓ ***Consistency*** – app's problem
- ¿ ***Isolation*** – two-phase locking ?
- ✓ ***Durability*** – write-ahead logging

Different Types of “Locks”

Certainly you are familiar with:

- Exclusive Locks
  - E.g., Mutex Locks
- Shared/Exclusive Locks
  - E.g., Read/Write Locks

Alone the above does not guarantee Isolation

- Why? Because of relocking & rollbacks

Debit/Credit with Error Checks

```
public void xfer(int fromAcctNum,
                 int toAcctNum,
                 double amount) throws RollbackException {
{
    try {
        Transaction.begin();

        Acct t = acctFactory.lookup(toAcctNum);
        if (t == null) throw new RollbackException("No acct: "+toAcctNum);
        t.setBalance(t.getBalance()+amount);

        Acct f = acctFactory.lookup(fromAcctNum);
        if (f == null) throw new RollbackException("No acct: "+fromAcctNum);
        if (f.getBalance() < amount) throw new RollbackException("Not enough…
        f.setBalance(f.getBalance()-amount);

        Transaction.commit();
    } finally {
        if (Transaction.isActive()) Transaction.rollback();
    }
}
```

xfer()

```
Transaction.begin();
Acct t = factory.lookup(toAcctNum);
exclusiveLock(t);
t.setBalance(t.getBalance()+200);
unlock(t);

Acct f = factory.lookup(fromAcctNum);
exclusiveLock(f);
if (f.getBalance() < 200))
   ... Transaction.rollback();
```

debit()

```
Transaction.begin();
Acct t = factory.lookup(toAcctNum);
exclusiveLock(t);
if (t.getBalance() < 100) throw ...;
t.setBalance(t.getBalance()-100);
unlock(t);
Transaction.commit();
```

Broken Locking Example

<toAcctNum>
balance: $3

<toAcctNum>
balance:$103

<fromAcctNum>
balance:$100

<fromAcctNum>
balance:$100

Log

⋮
20:Change rec: tid #68 <toAcctNum> old-value: $3 new-value: $203
21:Change rec: tid #69 <toAcctNum> old-value: $203 new-value: $103
22:Commit: tid #69

Problems with Previous Example

1. Debit transaction (#69) sees a balance that will never exist when transactions execute in isolation
2. Transfer transaction (#68) cannot rollback because we cannot undo it’s work but leave #69s work!

Use Two-Phase Locking

Phase 1: grab locks; Phase 2: drop locks

- You're not allowed to get any new locks after you start dropping your locks
- To execute rollback you must hold locks
- Usually, we hold all locks until commit or rollback has completed
  - E.g., there is a lock() method, but no unlock()…locks are dropped by commit() or rollback() methods

xfer()

```
Transaction.begin();
Acct t = factory.lookup(toAcctNum);
exclusiveLock(t);
t.setBalance(t.getBalance()+200);
unlock(t);

Acct f = factory.lookup(fromAcctNum);
exclusiveLock(f);
if (f.getBalance() < 200))
  ... Transaction.rollback();
```

debit()

```
Transaction.begin();
Acct t = factory.lookup(toAcctNum);
exclusiveLock(t);
if (t.getBalance() < 100) throw ...;
t.setBalance(t.getBalance()-100);
unlock(t);
Transaction.commit();
```

<toAcctNum>
balance: $3

<toAcctNum>
balance:$203

<fromAcctNum>
balance:$100

<fromAcctNum>
balance:$100

Log

⋮
20:Change rec: tid #68 <toAcctNum> old-value: $3 new-value: $203

xfer()

```
Transaction.begin();
Acct t = factory.lookup(toAcctNum);
exclusiveLock(t);
t.setBalance(t.getBalance()+200);
```
~~unlock(t);~~ **X**

```
Acct f = factory.lookup(fromAcctNum);
exclusiveLock(f);
if (f.getBalance() < 200))
   ... Transaction.rollback();
```

debit()

```
Transaction.begin();
Acct t = factory.lookup(toAcctNum);
exclusiveLock(t);
if (t.getBalance() < 100) throw ...;
t.setBalance(t.getBalance()-100);
```
~~unlock(t);~~
```
Transaction.commit();
```

<toAcctNum>	*<toAcctNum>*
balance: $3	balance: ~~$203~~ *3*

<fromAcctNum>	*<fromAcctNum>*
balance:$100	balance:$100

Log

⋮
20:Change rec: tid #68 <toAcctNum> old-value: $3 new-value: $203
21:Rollback: tid #68
22:Rollback: tid #69

Alternate Locking Schemes

- Many locking optimizations and fancy schemes have been devised
  - E.g., Increment lock and operation logging
    - Increment locks are compatible with each other
    - Increment locks not compat with read or write locks

xfer()

```
Transaction.begin();
Acct t = factory.lookup(toAcctNum);
incrementLock(t);
t.setBalance(t.getBalance()+200);

Acct f = factory.lookup(fromAcctNum);
exclusiveLock(f);
if (f.getBalance() < 200))
   ... Transaction.rollback();
```

debit()

```
Transaction.begin();
Acct t = factory.lookup(toAcctNum);
incrementLock(t);
if (t.getBalance() < 100) throw ...;
t.setBalance(t.getBalance()-100);
Transaction.commit();
```

<toAcctNum>	*<toAcctNum>*
balance: $3	balance: ~~$103~~ *-97*

<fromAcctNum>	*<fromAcctNum>*
balance:$100	balance:$100

Log

⋮
20:Change rec: tid #68 <toAcctNum> increment-by: $200
21:Change rec: tid #69 <toAcctNum> increment-by: –$100
22:Commit: tid #69
23:Rollback: tid #68

Avoiding Lock-out

- Locks are held on specific portions of the data
- Avoid dead-lock: E.g.,ordering: if all transactions (threads) grab locks in “alphabetical” order (or any specific ordering)
  - Alternatively, deal with it using timeout
    - Timeout transactions are rolled back by the “system”
- Avoid live-lock: E.g., waiting writers prevent new transactions from getting read locks

How Does Data Get Written to Disk?

- Does the OS buffer the writes?
  - Not for DB files
- Does the disk write happen atomically?
  - Manufacturers use NV memory
  - Recovery gurus add check bits & LSNs to headers

What About Disasters

- Power failure?
- Data disk failure?
- Log disk failure?
- Machine room failure?
  - Fire, flood, explosions, etc

What About Disasters

- Power failure: write-ahead logging
- Data disk failure: backup tapes & log
- Log disk failure: mirror the log
- Machine room failure: mirror the log elsewhere

Why Is This Relevant to OS?

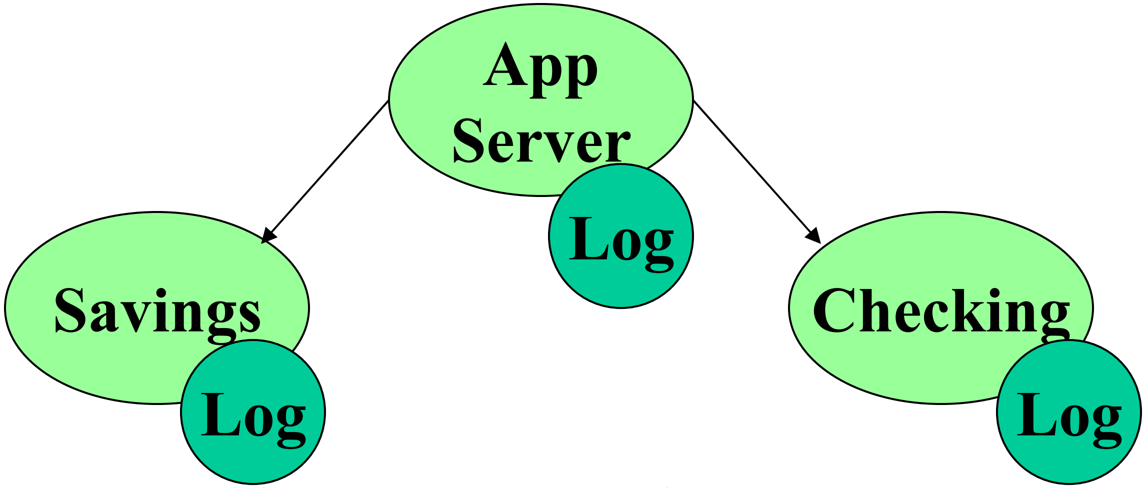
- Databases stole all this from operating systems and transaction systems
- Some OS services are better implemented using ACID properties
  - Journaling file systems

History

- First, atomic transactions were added on at application-level (in TP Monitors)
- Then they were added to OS (mostly research OSs)
- Then they were back in the app with RBDs
- Then they were generalized to create DTP

Distributed Two-Phase Commit

- You can have distributed transactions
  - RPC, access multiple databases, etc
  - DTP: Prepare Phase (subs flush), Commit Phase (coord flush)



File Systems & Transactions

- RDBs are happy to manage whole disks
- There is more to life than relational data
  - HTML, Images, Office Docs, Source, Binaries
- If you don’t otherwise need a RDB, put your files in a file system
- The file system doesn’t allow user-level apps to compose transactions so implementation is easier
- File system operations that require ACID properties:
  - For sure: create, delete, rename, modify properties
  - Often: write

How File Systems Implement ACID?

- Older/low-tech file systems are not log-based
  - Carefully writing to the disk
  - scandisk, chkdsk, fsck
- Newer file systems are log-based
  - E.g., NTFS, Network Appliance's NFS, JFS
  - Transactions are (usually) specialized
    - Not running general, user provided transactions
      - creat(), rename()
    - Allows specialized locking and logging
  - Note: Microsoft allows DTC with NTFS

Conclusion

- The file system is a fine place for static application data
  - HTML, Images, Application Code
- The database is the right place for application data
  - You have to maintain the database
    - In particular backup and fail-over
  - The higher volume your app, the more you need a DB
    - You need transactional consistency (ACID) guarantees
  - For low volume, non-critical applications, you can store data in the file system
    - You just need to back up the file system and be prepared to deal with the very small chance you'll have inconsistencies after a failure

Any Questions?