15-410

Atomic Transactions

December 6, 2006

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, ∃ serial ordering
  - Durability: committed transaction cannot be undone

When Are Transactions Used?

- When you use:
  - Databases
  - File Systems
- Applications built on the above
  - 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
  - Now at Microsoft Research in San Francisco
  - 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 Rollback…
    public void delete(Object... priKeyValues) throws Rollback…

    public int  getBeanCount()                  throws Rollback…
    public B    lookup(Object... priKeyValues) throws Rollback…

    public B[]  match(MatchArg... constaints)  throws Rollback…
}
```

- BeanFactory uses introspection 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 f = acctFactory.lookup(fromAcctNum);
    f.setBalance(f.getBalance()-amount);
    Acct t = acctFactory.lookup(toAcctNum);
    t.setBalance(t.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 f = acctFactory.lookup(fromAcctNum);
    f.setBalance(f.getBalance()-amount);
    Acct t = acctFactory.lookup(toAcctNum);
    t.setBalance(t.getBalance()+amount);
    Transaction.commit();
}
```

How Are ACID Properties Enforced?

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

How Are ACID Properties Enforced?

- A ***high-performance*** implementation
  - Complex disk data structures (B-trees)
  - ***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 all changes in a log before they are written to disk
  - When changing data pages, describe changes in log records
  - When committing, write commit-record into log, flush log
  - Before flushing cached pages, check ensure log was 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	*Memory Buffer Cache*	*Log*
<fromAcctNum> `balance:$100`		⋮
<toAcctNum> `balance:  $3`		

Green log records have been flushed to disk

Sample Log

```
Transaction.begin();
Acct f = factory.lookup(fromAcctNum);
…f.getBalance()…
```

Disk Storage

<fromAcctNum>

balance:$100

<toAcctNum>

balance: $3

Memory Buffer Cache

<fromAcctNum>

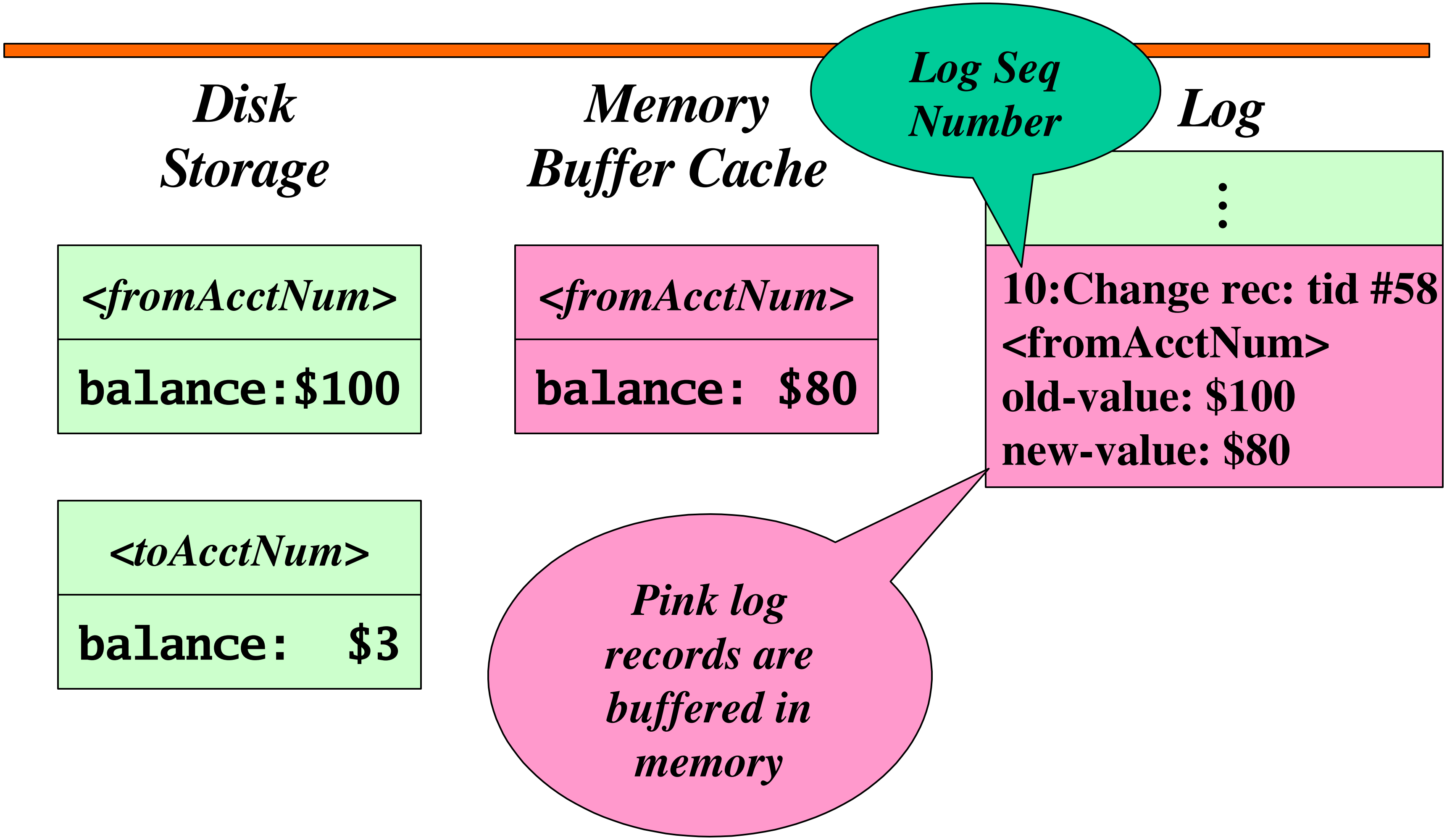
balance:$100

Log

⋮

Sample Log

```
Transaction.begin();
Acct f = factory.lookup(fromAcctNum);
f.setBalance(f.getBalance()-amount);
```



Sample Log

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

Disk Storage	*Memory Buffer Cache*	*Log*
<fromAcctNum>	*<fromAcctNum>*	⋮
balance:$100	**balance: $80**	**10:Change rec: tid #58 <fromAcctNum> old-value: $100 new-value: $80**
<toAcctNum>	*<toAcctNum>*	
balance: $3	**balance: $3**	

Sample Log

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

Disk Storage

<fromAcctNum>

balance:$100

<toAcctNum>

balance: $3

Memory Buffer Cache

<fromAcctNum>

balance: $80

<toAcctNum>

balance: $23

Log

⋮

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

⋮

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

Sample Log

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

Disk Storage	*Memory Buffer Cache*
<fromAcctNum>	*<fromAcctNum>*
balance:$100	**balance: $80**
<toAcctNum>	*<toAcctNum>*
balance: $3	**balance: $23**

Log

⋮

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

⋮

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

13:Commit: tid #58

Performance Improvement

- You do not need to flush the 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
- 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	*Memory Buffer Cache*	*Log*
<fromAcctNum> **balance:$100**		⋮
		10:Change rec: tid #58 **<fromAcctNum>** **old-value: $100** **new-value: $80**
<toAcctNum> **balance: $3**		⋮
		12:Change rec: tid #58 **<toAcctNum>** **old-value: $3** **new-value: $23**
		13:Commit: tid #58

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

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

Log

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

Buffer Cache Can Be Flushed Mid-Transaction

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

Disk Storage

<fromAcctNum>

balance: $80

<toAcctNum>

balance: $3

Memory Buffer Cache

<toAcctNum>

ba e: $23

Log

⋮

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

⋮

12:Change rec: tid #58
<toAcctNum>
value: $3
e: $23

Be sure the relevant portion of the log is flushed before the buffer cache is flushed

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

Disk Storage

<fromAcctNum>
balance: ~~$80~~ *100*

<toAcctNum>
balance: $3

Memory Buffer Cache

Log

⋮

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

Rollback using the log (Case 3)

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

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

Log

⋮

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

⋮

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

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 record are written describing the state of the buffer cache
  - Rollback records written to the log
  - Log 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

Isolation via Locking

- Lock portions of the data (e.g., records, rows, tables)
- Only one transaction can hold a write lock
  - Just use a mutex or semaphore
  - But limits concurrency

		Lock Held by Other Trans				
		None	W	R	RIW	Incr
Lock Requested	W	Grant	Wait			
	R					
	RIW					
	Incr					

Isolation via Locking

- Multiple transactions can hold read locks
  - Improves concurrency
  - But readers can starve writers
  - Two transactions trying to upgrade (R => W) can cause deadlock

	Lock Held by Other Trans				
Lock Requested	None	W	R	RIW	Incr
W	Grant	Wait	Wait		
R	Grant	Wait	Grant		
RIW					
Incr					

Fancy Locks (1)

- Read-intent-write allows one transaction to lock-out other writers, while others are still reading

* To prevent readers from staving writers, make new readers wait when RIW transaction wants to upgrade to W

		Lock Held by Other Trans				
		None	W	R	RIW	Incr
Lock Requested	W	Grant	Wait	Wait	Wait	
	R	Grant	Wait	Grant	*	
	RIW	Grant	Wait	Grant	Wait	
	Incr					

Fancy Locks (2)

- Increment locks allow concurrent writes.
  Example: increment by *x*. Use operation logging:
  Redo: +*x*. Undo: –*x*.

Lock Requested	Lock Held by Other Trans: None	W	R	RIW	Incr
W	Grant	Wait	Wait	Wait	Wait
R	Grant	Wait	Grant	*	Wait
RIW	Grant	Wait	Grant	Wait	Wait
Incr	Grant	Wait	Wait	Wait	Grant

Two-phase Locking

- Grab locks and keep then until until end-of-transaction, so others won't see uncommitted changes

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)
- 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?
- Does the disk write happen atomically?

What is the Atomicity of Disk Writes?

- When you write to the disk, does it all go out?
  - Sector = 512 bytes
  - Track = n Sectors
  - Block (or page) = m Sectors
- OS writes blocks/pages
- Disk has ECC codes…can detect partial sector
  - Often there is hardware support (NV memory buffer)
- We steal a few bits on each sector to detect partial blocks / pages
  - Often there are extra bits in the sector header
  - Often we will store LSN in the sector/header or block

Bad blocks

- A block is bad if it's partially written
  - ECC detects sector error
  - Our tags on the sectors don't match
- If a log block is bad…it had better be part of the last write…good idea: mirror the log
- If data block (page) is bad…restore from backup and apply all committed changes

Remind You of Something?

- A Relational Database
  - Any database

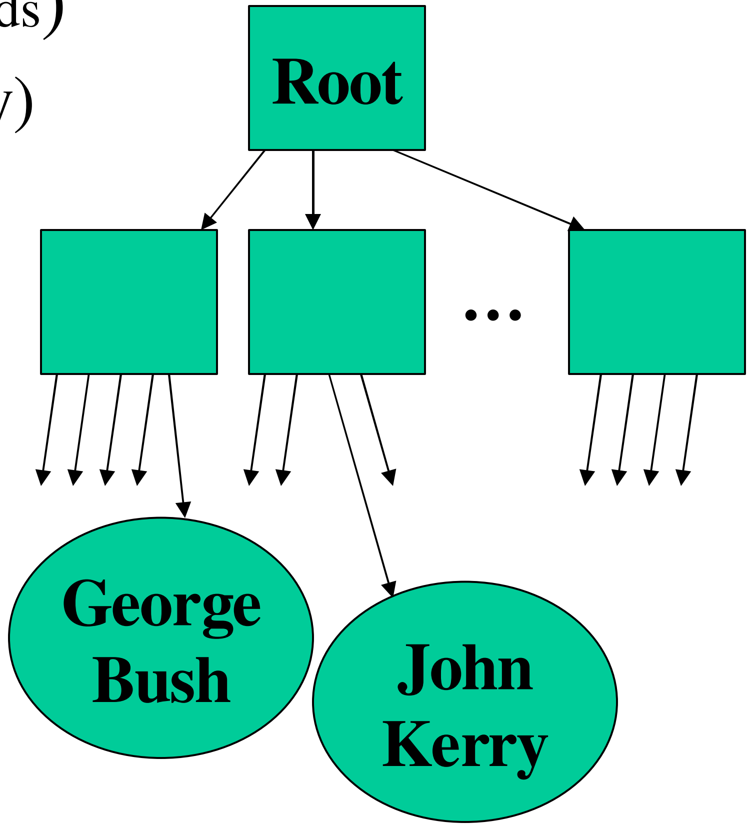
Why Is This Relevant to OS?

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

- Let's start in the beginning…

In the Old Days: OS provided

- Structured files (containing records)
  - Entry-sequenced (append-only)
  - Relative (array)
  - B-tree clustered (hash table)
- Secondary access methods
- Many field types
  - Character data
  - Integers
  - Floats
  - Dates



In the Old Days: TPM provided

- ACID properties for the OS files
  - Transactions
  - Logging
  - Recovery

Today: Relational Databases

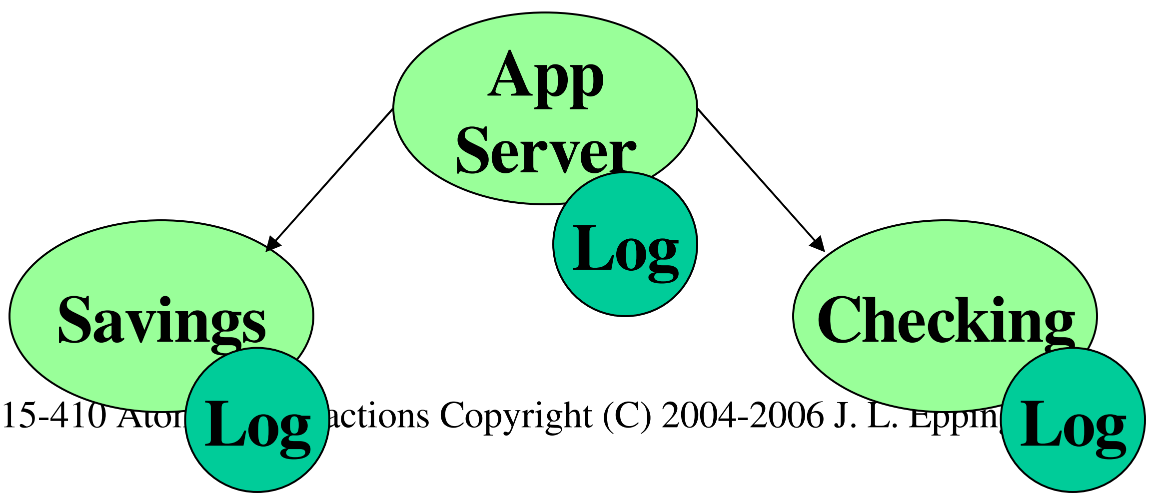
- Structured files
- ACID properties
- SQL Interface

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)



Why Do You Care?

- 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

File Systems & Transactions

- If you don't allow user-level apps to compose transactions, implementation is easier
- FS Ops 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 specialized
    - Not running general, user provided transactions
      - creat(), rename()
    - Allows specialized locking and logging

Any Questions?