15-410

Atomic Transactions

December 5, 2005

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***

When Are Transactions Used?

- When you use:
  - Databases
  - File Systems
- Things built on the above
  - Banking Applications
  - BeanFactories

Who Invented Atomic Transactions?

- The guys that built TP Monitors

Next

- We'll talk about the details
  - The guarantees
  - How to provide them

Do You Do ACID?

- What is ACID?
- The ACID Properties of a Transaction:
  - Atomicity: all or none
  - Consistency: if consistent before transaction, so too after
  - Isolation: despite concurrent execution, ∃ serial ordering
  - Durability: committed transaction cannot be undone

What is a BBoard?

- So you know what a BBoard is…here's a BBoard story
- Nico had a secretary, Suzanna
- CS Ph.D. students not nice on the BBoard
- Suzanna makes post – calls students profane names
- My Ph.D. student friend Dan responds
- Suzanna gets facilities to delete her message
- Dan says – whoa, what about the ACID properties?

Remember the ACID Properties?

Atomicity: all or none

Consistency: if before than after

Isolation: serial ordering

Durability: cannot be undone

- Atomicity: No partial messages/updates
  - Whole messages, index refers to all messages, etc
- Consistency: BBoard transactions (apps) do “all work”
  - App must update all relevant data and do it correctly
- Isolation: no showing of “uncommitted” work
  - If concurrent postings…
- Durability: No unposting
  - Also, no cascading “aborts”

What is a Transaction?

Let's be more concrete

- A group of sub-operations that as a whole conform to the ACID properties

```
private BankAccount savings = new BankAccount(…);
private BankAccount checking = new BankAccount(…);
public void transferStoC(double amount) throws … {
    savings.write(savings.read()-amount);
    checking.write(checking.read()+amount);
}
public void transferCtoS(double amount) throws … { }
```

- You want these transfers to be ACID

Remember the ACID Properties?

Atomicity: all or none

Consistency: if before than after

Isolation: serial ordering

Durability: cannot be undone

- Let’s consider the ACID properties for these transfer “transactions”:

```
private BankAccount savings = new BankAccount(…);
private BankAccount checking = new BankAccount(…);
public void transferStoC(double amount) throws … {
    savings.write(savings.read()-amount);
    checking.write(checking.read()+amount);
}
public void transferCtoS(double amount) throws … { }
```

- So how do you make this work?

Example Implementation

```
public class BankAccount {
    private double balance;

    public double getBalance() {
        return balance;
    }

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

Remember the ACID Properties?

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

```
private BankAccount savings = new BankAccount(…);
private BankAccount checking = new BankAccount(…);
public void transferStoC(double amount) throws … {
    savings.write(savings.read()-amount);
    checking.write(checking.read()+amount);
}
public void transferCtoS(double amount) throws … { }
```

```
public class BankAccount {
    private double balance;
    public double getBalance()       { return balance; }
    public void setBalance(double x) { balance = x; }
}
```

How About This One?

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

```
public class BankAccount {
    private static RandomAccessFile f = new Ra…("…","rws");
    private long myPosInFile = …;
    public double getBalance() throws IOException {
        synchronized (f) {
            f.seek(myPosInFile);
            return f.readDouble();
        }
    }
    public void setBalance(double x) throws IOException {
        synchronized (f) {
            f.seek(myPosInFile);
            f.writeDouble(x);
        }
    }
}
```

How Does Data Get Written to Disk?

- Does the OS buffer the writes?
- Does the disk write happen atomically?

Let's Address Isolation

You Need to Delineate the Transaction

```
public void transferStoC(double amount) throws … {
    Transaction.begin();
    savings.write(savings.read()-amount);
    checking.write(checking.read()+amount);
    Transaction.commit();
}
```

```
public class Transaction {
    private static ThreadLocal tid = new ThreadLocal();
    public static void begin()    { tid.set(nextTid()); }
    public static void commit()   { /* hard work goes here */ }
    public static void rollback() { /* hard work goes here */ }
}
```

How Are ACID Properties Enforced?

```
public void transferStoC(double amount) throws … {
    Transaction.begin();
    savings.write(savings.read()-amount);
    checking.write(checking.read()+amount);
    Transaction.commit();
}
```

- Atomicity – logging
- Consistency – app’s problem
- Isolation – locking
- Durability – logging

Remind You of Something?

- A Relational Database
  - Any database

How Does a Relational DB Do It? (1)

- Consistency
  - Code must be correct
- Isolation
  - Two-phased read-write locking
  - Read-intent-write lock & ordering

		Lock Held by other Trans				
		None	R	W	RIW	Incr
	R	√	√		√	
Lock Requested	W	√				
	RIW	√	√			
	Incr	√				√

More on Locking (1)

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

		Lock Held by other Trans				
		None	R	W	RIW	Incr
Lock Requested	R	√	√		√	
	W	√				
	RIW	√	√			
	Incr	√				√

More on Locking (2)

- Avoiding Lock-out
  - Ordering to avoid deadlocks … if all transactions (threads) grab locks in "alphabetical" order (or any specific ordering)
  - Read-intent-write lock … keeps a stream of readers from livelocking our writers

		Lock Held by other Trans				
		None	R	W	RIW	Incr
Lock Requested	R	√	√		√	
	W	√				
	RIW	√	√			
	Incr	√				√

How Does a Relational DB Do It? (2)

- Atomicity & Durability
  - Buffer database disk pages in memory
  - Log all changes in a write-ahead log
    - When changing data pages, describe in log recs
    - When flushing data pages, check that log flushed
    - When committing, commit-record into log, flush log
  - Recover from the log
    - When rolling back, scan log and undo
    - When restarting after a failure, scan the log
      - Undo transactions without commit records, as necessary
      - Redo transactions with commit records, as necessary

How Do You Describe Changes

- Value Logging
  - E.g., old value = 4, new value = 5
- Operation Logging
  - E.g., increment by 1

Sample Log

⋮
Change rec: **tid #584** **acct <savings>** **old-value: $100** **new-value: $80**
⋮
Change rec: **tid#584** **acct <checking>** **old-value: $3** **new-value: $23**
Commit rec: **tid#584**
⋮

How Does a Relational DB Do It? (2)

- Atomicity & Durability
  - Buffer database disk pages in memory
  - Log all changes in a write-ahead log
    - When changing data pages, describe in log recs
    - When flushing data pages, check that log flushed
    - When committing, commit-record into log, flush log
  - Recover from the log
    - When rolling back, scan log and undo
    - When restarting after a failure, scan the log
      - Undo transactions without commit records, as necessary
      - Redo transactions with commit records, as necessary

How Does a Relational DB Do It? (3)

- More on Atomicity & Durability
  - Databases are very careful when they write to disk
  - They control the buffering of pages in memory
  - The log is append-only, order of records counts
    - If commit rec present, preceded by descrip. of changes…
    - If descrip of changes present, without commit rec …
    - We track the last log rec # that applies to ea data page…
      - Log recs describing changes, go out before the page w/changes
      - Often, we put the last log rec # on each data page

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
- Disk has ECC codes…can detect partial sector
- How do you detect if you have a partial 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

Caveat

- This is just a basic example of how a database really works
- There are many, many optimizations
  - E.g., checkpointing the log limits recovery scan
  - E.g., operation logging permits add’l locking modes
    - E.g., increment locks

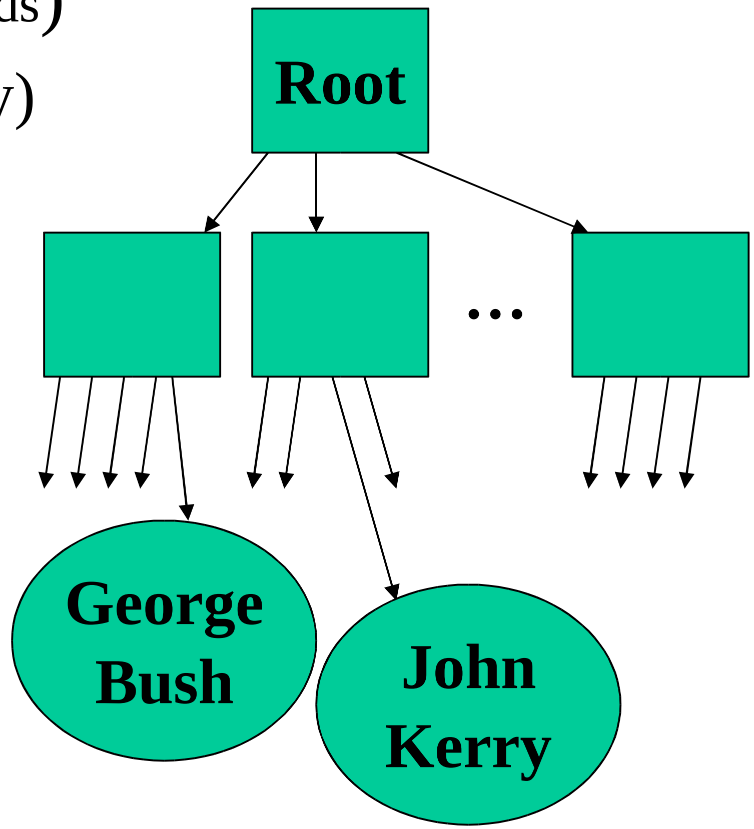
Why Is This Relevant to OS?

- Databases stole all this from operating systems
- Some OS services require ACID properties

- Let’s start in the beginning…

In the Old Days

- 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



Today we have Relational Databases

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

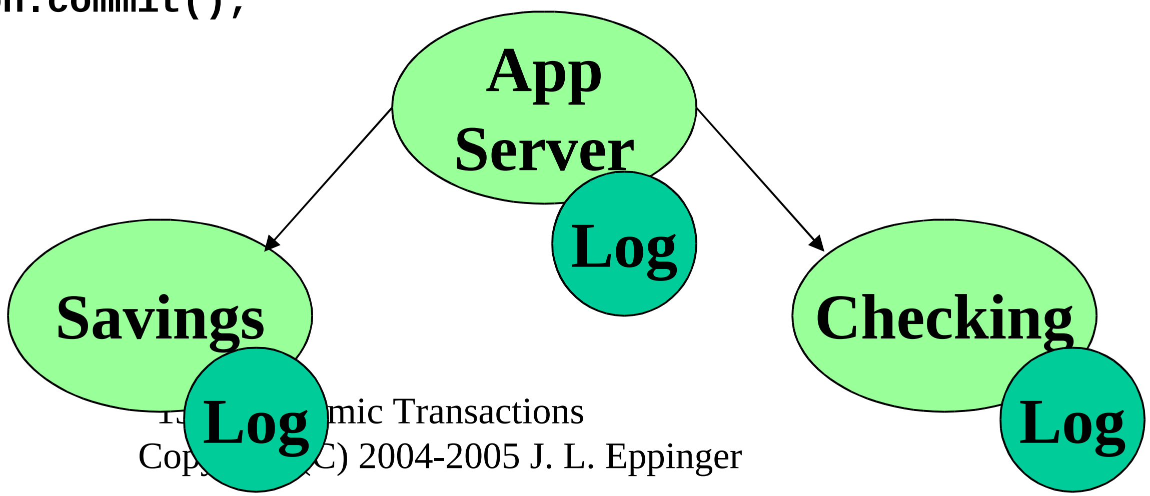
In the Old Days

- 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)

```
public void transferStoC(double amount) throws … {
   Transaction.begin();
   savings.write(savings.read()-amount);
   checking.write(checking.read()+amount);
   Transaction.commit();
}
```



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?

- Carefully writing to the disk
  - DBs are careful, too
- Older/cheaper file systems are not log-based
  - scandisk, chkdsk, fsck
- Newer file systems are log-based
  - E.g., NTFS, Network Appliance’s NFS, JFS

How Do I Use Transactions?

- JavaBeans
  - BeanFactories
    - Exclusive locks
    - When backed by database:
      - DB provided ACID properties
    - When backed by file systems
      - One big lock for concurrency control
      - Rename (aka pointer swap) for atomicity
      - Backup for durability (weak)

Any Questions?