3. Concurrency Control for Transactions

Part One

CSEP 545 Transaction Processing
Philip A. Bernstein

Copyright ©2012 Philip A. Bernstein
Outline

1. A Simple System Model
2. Serializability Theory
3. Synchronization Requirements for Recoverability
4. Two-Phase Locking
5. Preserving Transaction Handshakes
6. Implementing Two-Phase Locking
7. Deadlocks
3.1 A Simple System Model

- Goal - Ensure serializable (SR) executions
- Implementation technique - Delay operations that may lead to non-SR results (e.g. set locks on shared data)
- For good performance minimize *overhead* and *delay* from synchronization operations
- First, we’ll study how to get correct (SR) results
- Then, we’ll study performance implications (mostly in Part Two)
Assumption - Atomic Operations

• We will synchronize Reads and Writes.
• We must therefore assume they’re atomic
  – else we’d have to synchronize the finer-grained operations that implement Read and Write
• Read(x) - returns the current value of x in the DB
• Write(x, val) overwrites all of x (the whole page)
• This assumption of atomic operations allows us to abstract executions as sequences of reads and writes (without loss of information).
  – Otherwise, what would $w_k[x]$ $r_i[x]$ mean?
• Also, commit ($c_i$) and abort ($a_i$) are atomic
System Model

Transaction 1
Transaction 2
Transaction N

Start, Commit, Abort
Read(x), Write(x)

Data Manager

Database
3.2 Serializability Theory

• The theory is based on modeling executions as histories, such as
  \[ H_1 = r_1[x] \ r_2[x] \ w_1[x] \ c_1 \ w_2[y] \ c_2 \]

• First, characterize a concurrency control algorithm by the properties of histories it allows

• Then prove that any history having these properties is SR

• Why bother? It helps you understand why concurrency control algorithms work
Equivalence of Histories

• Two operations conflict if their execution order affects their return values or the DB state.
  – A read and write on the same data item conflict.
  – Two writes on the same data item conflict.
  – Two reads (on the same data item) do not conflict.

• Two histories are equivalent if they have the same operations and conflicting operations are in the same order in both histories.
  – Because only the relative order of conflicting operations can affect the result of the histories.
Examples of Equivalence

• The following histories are equivalent

\[ H_1 = r_1[x] r_2[x] w_1[x] c_1 w_2[y] c_2 \]
\[ H_2 = r_2[x] r_1[x] w_1[x] c_1 w_2[y] c_2 \]
\[ H_3 = r_2[x] r_1[x] w_2[y] c_2 w_1[x] c_1 \]
\[ H_4 = r_2[x] w_2[y] c_2 r_1[x] w_1[x] c_1 \]

• But none of them are equivalent to

\[ H_5 = r_1[x] w_1[x] r_2[x] c_1 w_2[y] c_2 \]
which reverses the order of \( r_2[x] w_1[x] \) in \( H_1 \), because \( r_2[x] \) and \( w_1[x] \) conflict and \( r_2[x] \) precedes \( w_1[x] \) in \( H_1 - H_4 \), but \( r_2[x] \) follows \( w_1[x] \) in \( H_5 \).
Serializable Histories

• Definition: A history is \textit{serializable} (SR) if it is equivalent to a serial history

• For example,

\begin{align*}
H_1 &= r_1[x] \ r_2[x] \ w_1[x] \ c_1 \ w_2[y] \ c_2 \\
H_4 &= r_2[x] \ w_2[y] \ c_2 \ r_1[x] \ w_1[x] \ c_1
\end{align*}

is equivalent to

\begin{align*}
H_1 &= r_1[x] \ r_2[x] \ w_1[x] \ c_1 \ w_2[y] \ c_2 \\
H_4 &= r_2[x] \ w_2[y] \ c_2 \ r_1[x] \ w_1[x] \ c_1
\end{align*}

(Because \(H_1\) and \(H_4\) have the same operations and the only conflicting operations, \(r_2[x]\) and \(w_1[x]\), are in the same order in \(H_1\) and \(H_4\).)

• Therefore, \(H_1\) is serializable.
Another Example

- $H_6 = r_1[x] \ r_2[x] \ w_1[x] \ r_3[x] \ w_2[y] \ w_3[x] \ c_3 \ w_1[y] \ c_1 \ c_2$
  is equivalent to a serial execution of $T_2 \ T_1 \ T_3$,
- $H_7 = r_2[x] \ w_2[y] \ c_2 \ r_1[x] \ w_1[x] \ w_1[y] \ c_1 \ r_3[x] \ w_3[x] \ c_3$
- Each conflict implies a constraint on any equivalent serial history:

\[
H_6 = r_1[x] \ r_2[x] \ w_1[x] \ r_3[x] \ w_2[y] \ w_3[x] \ c_3 \ w_1[y] \ c_1 \ c_2
\]
Serialization Graphs

• A serialization graph, SG(H), for history H tells the effective execution order of transactions in H.
• Given history H, SG(H) is a directed graph whose nodes are the committed transactions and whose edges are all $T_i \rightarrow T_k$ such that at least one of $T_i$’s operations precedes and conflicts with at least one of $T_k$’s operations.

$$H_6 = r_1[x] \ r_2[x] \ w_1[x] \ r_3[x] \ w_2[y] \ w_3[x] \ c_3 \ w_1[y] \ c_1 \ c_2$$

$$SG(H_6) = T_2 \rightarrow T_1 \rightarrow T_3$$
The Serializability Theorem

A history is SR if and only if SG(H) is acyclic.

Proof: (if) SG(H) is acyclic. So let \( H_s \) be a serial history consistent with SG(H). Each pair of conflicting ops in \( H \) induces an edge in SG(H). Since conflicting ops in \( H_s \) and \( H \) are in the same order, \( H_s \equiv H \), so \( H \) is SR.

(only if) \( H \) is SR. Let \( H_s \) be a serial history equivalent to \( H \). We claim that if \( T_i \rightarrow T_k \) in SG(H), then \( T_i \) precedes \( T_k \) in \( H_s \) (else \( H_s \not\equiv H \)). If SG(H) had a cycle, \( T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1 \), then \( T_1 \) would precede \( T_1 \) in \( H_s \), a contradiction. So SG(H) is acyclic.
How to Use the Serializability Theorem

• Characterize the set of histories that a concurrency control algorithm allows.
• Prove that any such history must have an acyclic serialization graph.
• Therefore, the algorithm guarantees SR executions.
• We’ll use this soon to prove that locking produces serializable executions.
3.3 Synchronization Requirements for Recoverability

• In addition to ensuring serializability, synchronization is needed to implement abort easily.

• When a transaction T aborts, the data manager wipes out all of T’s effects, including
  – Undoing T’s writes that were applied to the DB
    • Remember before-images of writes
  – Aborting transactions that read values written by T (these are called cascading aborts)
    • Remember which transactions read T’s writes
Recoverability Example

• Example -  $w_1[x] \ r_2[x] \ w_2[y]$
  
  – To abort $T_1$, we must undo $w_1[x]$ and abort $T_2$ (a cascading abort).
  
  – System should keep before image of $x$ in case $T_1$ aborts
    • We may even need to remember other before images.
  
  – System should make $T_2$ dependent on $T_1$
    • If $T_1$ aborts $T_2$ aborts.

• We want to avoid some of this bookkeeping.
Recoverability

- If $T_k$ reads from $T_i$ and $T_i$ aborts, then $T_k$ must abort
  - Example - $w_1[x] \text{ r}_2[x] a_1$ implies $T_2$ must abort
- But what if $T_k$ already committed? We’d be stuck.
  - Example - $w_1[x] \text{ r}_2[x] c_2 a_1$
  - $T_2$ can’t abort after it commits

- Executions must be recoverable:
  A transaction T’s commit operation must follow the commit of every transaction from which T read.
  - Recoverable - $w_1[x] \text{ r}_2[x] c_1 c_2$
  - Not recoverable - $w_1[x] \text{ r}_2[x] c_2 a_1$
- Recoverability requires synchronizing operations.
Avoiding Cascading Aborts

• Cascading aborts are worth avoiding to
  – Avoid complex bookkeeping, and
  – Avoid an uncontrolled number of forced aborts

• To avoid cascading aborts, a data manager should ensure transactions read only committed data

• Example
  – Avoids cascading aborts: $w_1[x] \rightarrow c_1 \rightarrow r_2[x]$
  – Allows cascading aborts: $w_1[x] \rightarrow r_2[x] \rightarrow a_1$

• A system that avoids cascading aborts also guarantees recoverability.
Strictness

• It’s convenient to undo a write, w[x], by restoring its before image (x’s value before w[x] executed)

• Example - w_1[x,1] writes the value “1” into x.
  – w_1[x,1] w_1[y,3] c_1 w_2[y,1] r_2[x] a_2
  – Abort T_2 by restoring the before image of w_2[y,1] (i.e. 3)

• But this isn’t always possible.
  – For example, consider w_1[x,2] w_2[x,3] a_1 a_2
  – a_1 & a_2 can’t be implemented by restoring before images
  – Notice that w_1[x,2] w_2[x,3] a_2 a_1 would be OK

• A system is strict if it only reads or overwrites committed data.
Strictness (cont’d)

• More precisely, a system is strict if it only executes $r_i[x]$ or $w_i[x]$ if all previous transactions that wrote $x$ committed or aborted.

• Examples (‘…” marks a non-strict prefix)
  – Strict: $w_1[x]$ $c_1$ $w_2[x]$ $a_2$
  – Not strict: $w_1[x]$ $w_2[x]$ … $c_1$ $a_2$
  – Strict: $w_1[x]$ $w_1[y]$ $c_1$ $r_2[x]$ $w_2[y]$ $a_2$
  – Not strict: $w_1[x]$ $w_1[y]$ $r_2[x]$ … $c_1$ $w_2[y]$ $a_2$
  – To see why strictness matters in the above histories, consider what happens if $T_1$ aborts.

• “Strict” implies “avoids cascading aborts.”
3.4 Two-Phase Locking

• Basic locking - Each transaction sets a lock on each data item before accessing the data
  – The lock is a reservation
  – There are read locks and write locks
  – If one transaction has a write lock on x, then no other transaction can have any lock on x

• Example
  – $rl_i[x], ru_i[x], wl_i[x], wu_i[x]$ denote lock/unlock operations
  – $wl_1[x] w_1[x] rl_2[x] r_2[x]$ is impossible
  – $wl_1[x] w_1[x] wu_1[x] rl_2[x] r_2[x]$ is OK
Basic Locking Isn’t Enough

• Basic locking doesn’t guarantee serializability

\[\text{rl}_1[x] \text{r}_1[x] \text{ru}_1[x] \quad \text{wl}_1[y] \text{w}_1[y] \text{wu}_1[y]c_1\]

\[\text{rl}_2[y] \text{r}_2[y] \text{wl}_2[x] \text{w}_2[x] \text{ru}_2[y] \text{wu}_2[x]c_2\]

• Eliminating the lock operations, we have

\[\text{r}_1[x] \text{r}_2[y] \text{w}_2[x]c_2 \text{w}_1[y]c_1\] which isn’t SR

• The problem is that locks aren’t being released properly.
Two-Phase Locking (2PL) Protocol

- A transaction is *two-phase locked* if:
  - Before reading x, it sets a read lock on x
  - Before writing x, it sets a write lock on x
  - It holds each lock until after it executes the corresponding operation
  - After its first unlock operation, it requests no new locks.

- Each transaction sets locks during a *growing phase* and releases them during a *shrinking phase*.

- Example - on the previous page T₂ is two-phase locked, but not T₁ since ru₁[x] < wl₁[y]
  - use “<” for “precedes”.
2PL Theorem: If all transactions in an execution are two-phase locked, then the execution is SR.

Proof: Let $H$ be a 2PL history and $T_i \rightarrow T_k$ in $SG$.

- Then $T_i$ read $x$ and $T_k$ later wrote $x$,
- Or $T_i$ wrote $x$ and $T_k$ later read or wrote $x$

• If $T_i \rightarrow T_k$, then $T_i$ released a lock before $T_k$ obtained some lock.

• If $T_i \rightarrow T_k \rightarrow T_m$, then $T_i$ released a lock before $T_m$ obtained some lock (because $T_k$ is two-phase).

• If $T_i \rightarrow \ldots \rightarrow T_i$, then $T_i$ released a lock before $T_i$ obtained some lock, breaking the 2-phase rule.

• So there cannot be a cycle in $SG(H)$. By the Serializability Theorem, $H$ is SR.
2PL and Recoverability

- 2PL does not guarantee recoverability
- This non-recoverable execution is 2-phase locked
  \[ \text{wl}_1[x] \; \text{w}_1[x] \; \text{wu}_1[x] \; \text{rl}_2[x] \; r_2[x] \; c_2 \; \ldots \; c_1 \]
  - Hence, it is not strict and allows cascading aborts
- However, holding write locks until after commit or abort guarantees strictness
  - Hence avoids cascading aborts and is recoverable
  - In the above example, \( T_1 \) must commit before its first unlock-write (wu_1):
    \[ \text{wl}_1[x] \; w_1[x] \; c_1 \; \text{wu}_1[x] \; \text{rl}_2[x] \; r_2[x] \; c_2 \]
Automating Locking

• 2PL can be hidden from the application.
• When a data manager gets a Read or Write operation from a transaction, it sets a read or write lock.
• How does the data manager know it’s safe to release locks (and be two-phase)?
• Ordinarily, the data manager holds a transaction’s locks until it commits or aborts. A data manager
  – Can release read locks after it receives commit
  – Releases write locks only after it processes commit, to ensure strictness.
3.5 Preserving Transaction Handshakes

- Read and Write are the only operations the system will control to attain serializability.
- So, if transactions communicate via messages, then implement SendMsg as Write, and ReceiveMsg as Read.
- Else, you could have the following:
  \[ w_1[x] \ r_2[x] \ send_2[M] \ receive_1[M] \]
  - Data manager didn’t know about send/receive and thought the execution was SR.
- Also watch out for brain transport.
Transactions Can Communicate via Brain Transport

T1: Start
    . . .
    Display output
    Commit

User reads output
    ...
    User enters input

T2: Start
    Get input from display
    . . .
    Commit

Brain transport
Brain Transport (cont’d)

• For practical purposes, if the user waits for $T_1$ to commit before starting $T_2$, then the data manager can ignore brain transport.

• This is called a **transaction handshake** ($T_1$ commits before $T_2$ starts).

• Reason - Locking preserves the order imposed by transaction handshakes
  – e.g., it serializes $T_1$ before $T_2$. 
2PL Preserves Transaction Handshakes

• 2PL serializes transactions consistent with all transaction handshakes. I.e. there’s an equivalent serial execution that preserves the transaction order in all transaction handshakes.

• This isn’t true for arbitrary SR executions. E.g.

  – $r_1[x] \ w_2[x] \ c_2 \ r_3[y] \ c_3 \ w_1[y] \ c_1$
  – $T_2$ commits before $T_3$ starts, but the only equivalent serial execution is $T_3 \ T_1 \ T_2$
  – The history can’t occur using 2PL. Try adding lock ops: $rl_1[x] \ r_1[x] \ wl_1[y] \ ru_1[x] \ wl_2[x] \ w_2[x] \ c_2 \ wu_2[x]$ but now we’re stuck, since we can’t set $rl_3[y] \ r_3[y]$. 
How to show whether a given history H was produced by 2PL?

- H could have been produced via 2PL iff you can add lock operations to H, following 2PL protocol.

- First add rl₁[x]: \( \text{rl}_1[x] \) r₁[x] w₂[x] c₂ r₃[y] c₃ w₁[y] c₁
  
  - Next, T₂ must have set wl₂[x] before executing w₂[x]
    - So r₁[x] must have released rl₁[x] before w₂[x] ran
    - Since T₁ is 2PL, it must have write-locked y before unlocking x

- rl₁[x] r₁[x] w₁[y] ru₁[x] wl₂[x] w₂[x] c₂ wu₂[x]
  
  - Now we’re stuck, since T₃ could not have set rl₃[y] before r₃[y], since T₁ could not have unlocked y until after w₁[y].

- Hence, H could not have been produced by 2PL.
2PL Preserves Transaction Handshakes (cont’d)

• Stating this more formally …

• Theorem:

For any 2PL execution $H$, there is an equivalent serial execution $H_s$, such that for all $T_i, T_k$,

if $T_i$ committed before $T_k$ started in $H$,

then $T_i$ precedes $T_k$ in $H_s$. 
Brain Transport — One Last Time

• If a user reads displayed output of $T_i$ and wants to use that output as input to transaction $T_k$, then he/she should wait for $T_i$ to commit before starting $T_k$.

• The user can then rely on transaction handshake preservation to ensure $T_i$ is serialized before $T_k$. 
3.6 Implementing Two-Phase Locking

• Even if you never implement a DB system, it’s valuable to understand locking implementation, because it can have a big effect on performance.

• A data manager implements locking by
  – Implementing a lock manager
  – Setting a lock for each Read and Write
  – Handling deadlocks.
System Model

Transaction 1

Start, SQL Ops, Commit, Abort

Query Optimizer
Query Executor
Access Method (record-oriented files)
Page-oriented Files

Database System

... ...

Transaction N

Database
How to Implement SQL

- Query Optimizer - translates SQL into an ordered expression of relational DB operators (Select, Project, Join)
- Query Executor - executes the ordered expression by running a program for each operator, which in turn accesses records of files
- Access methods - provides indexed record-at-a-time access to files (OpenScan, GetNext, …)
- Page-oriented files - Read or Write (page address)
Which Operations Get Synchronized?

<table>
<thead>
<tr>
<th>Query Optimizer</th>
<th>SQL operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query Executor</td>
<td>Record-oriented operations</td>
</tr>
<tr>
<td>Access Method</td>
<td>Page-oriented operations</td>
</tr>
<tr>
<td>(record-oriented files)</td>
<td></td>
</tr>
<tr>
<td>Page-oriented Files</td>
<td></td>
</tr>
</tbody>
</table>

• It’s a tradeoff between
  – Amount of concurrency and
  – Runtime expense and programming complexity of synchronization
Lock Manager

- A lock manager services the operations
  - Lock(trans-id, data-item-id, mode)
  - Unlock(trans-id, data-item-id)
  - Unlock(trans-id)


<table>
<thead>
<tr>
<th>Data Item</th>
<th>List of Locks</th>
<th>Wait List</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>[T₁,r] [T₂,r]</td>
<td>[T₃,w]</td>
</tr>
<tr>
<td>y</td>
<td>[T₄,w]</td>
<td>[T₅,w] [T₆,r]</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lock Manager (cont’d)

• Caller generates data-item-id, e.g. by hashing data item name
• The lock table is hashed on data-item-id
• Lock and Unlock must be atomic, so access to the lock table must be “locked”
• Lock and Unlock are called frequently. They must be very fast. Average < 100 instructions.
  – This is hard, in part due to slow compare-and-swap operations needed for atomic access to lock table.
Lock Manager (cont’d)

• In MS SQL Server
  – Locks are approx 32 bytes each.
  – Each lock contains a Database-Id, Object-Id, and other resource-specific lock information such as record id (RID) or key.
  – Each lock is attached to lock resource block (64 bytes) and lock owner block (32 bytes).
Locking Granularity

• **Granularity** - size of data items to lock
  – e.g., files, pages, records, fields

• **Coarse granularity implies**
  – Very few locks, so little locking overhead
  – Must lock large chunks of data, so high chance of conflict, so concurrency may be low

• **Fine granularity implies**
  – Many locks, so high locking overhead
  – Locking conflict occurs only when two transactions try to access the exact same data concurrently

• **High performance TP requires record locking**
Multigranularity Locking (MGL)

- Allow different txns to lock at different granularity
  - Big queries should lock coarse-grained data (e.g. tables)
  - Short transactions lock fine-grained data (e.g. rows)
- Lock manager can’t detect these conflicts
  - Each data item (e.g., table or row) has a different id
- Multigranularity locking “trick”
  - Exploit the natural hierarchy of data containment
  - Before locking fine-grained data, set intention locks on coarse grained data that contains it
  - e.g., before setting a read-lock on a row, get an intention-read-lock on the table that contains the row
  - An intention-read-lock conflicts with a write lock on the same item
3.7 Deadlocks

• A set of transactions (txns) is deadlocked if every transaction in the set is blocked and will remain blocked unless the system intervenes
  – Example
    
    $\text{rl}_1[x]$ granted
    $\text{rl}_2[y]$ granted
    $\text{wl}_2[x]$ blocked
    $\text{wl}_1[y]$ blocked and deadlocked
  
• Deadlock is 2PL’s way to avoid non-SR executions
  – $\text{rl}_1[x] \text{ r}_1[x] \text{ rl}_2[y] \text{ r}_2[y] \ldots$ can’t run $\text{w}_2[x] \text{ w}_1[y]$ and be SR

• To repair a deadlock, you must abort a transaction
  – Releasing a txn T’s lock without aborting T breaks 2PL
Deadlock Prevention

• Never grant a lock that can lead to deadlock

• Often advocated in operating systems

• Useless for TP, because it would require running transactions serially
  – Example to prevent the previous deadlock, rl₁[x] rl₂[y] wl₂[x] wl₁[y], the system can’t grant rl₂[y]

• Avoiding deadlock by resource ordering is unusable in general, since it overly constrains applications
  – But may help for certain high frequency deadlocks

• Setting all locks when txn begins requires too much advance knowledge and reduces concurrency
Deadlock Detection

• Detection approach: Detect deadlocks automatically and abort a deadlocked transactions (the **victim**)
• It’s the preferred approach, because it
  – Allows higher resource utilization and
  – Uses cheaper algorithms

• Timeout-based deadlock detection - If a transaction is blocked for too long, then abort it
  – Simple and easy to implement
  – But aborts unnecessarily and
  – Some deadlocks persist for too long
Detection Using Waits-For Graph

- Explicit deadlock detection - Use a Waits-For Graph
  - Nodes = \{transactions\}
  - Edges = \{T_i \rightarrow T_k \mid T_i \text{ is waiting for } T_k \text{ to release a lock}\}
  - Example (previous deadlock) \( T_1 \leftrightarrow T_2 \)
- Theorem: If there’s a deadlock, then the waits-for graph has a cycle
Detection Using Waits-For Graph (cont’d)

- So, to find deadlocks
  - When a transaction blocks, add an edge to the graph.
  - Periodically check for cycles in the waits-for graph.

- Need not test for deadlocks too often.
  - A cycle won’t disappear until you detect it and break it.

- When a deadlock is detected, select a victim from the cycle and abort it.

- Select a victim that hasn’t done much work
  - E.g., has set the fewest locks.
Cyclic Restart

• Transactions can cause each other to abort forever.
  – $T_1$ starts running. Then $T_2$ starts running.
  – They deadlock and $T_1$ (the oldest) is aborted.
  – $T_1$ restarts, bumps into $T_2$ and again deadlocks
  – $T_2$ (the oldest) is aborted ...

• Choosing the youngest in a cycle as victim avoids cyclic restart, since the oldest running transaction is never the victim.

• Can combine with other heuristics, e.g. fewest-locks
MS SQL Server

• Aborts the transaction that is “cheapest” to roll back.
  – “Cheapest” is determined by the amount of log generated.
  – Allows transactions that you’ve invested a lot in to complete.

• SET DEADLOCK_PRIORITY LOW (vs. NORMAL) causes a transaction to sacrifice itself as a victim.
Distributed Locking

• Suppose a transaction can access data at many data managers
• Each data manager sets locks in the usual way
• When a transaction commits or aborts, it runs two-phase commit to notify all data managers it accessed
• The only remaining issue is distributed deadlock
Distributed Deadlock

• The deadlock spans two nodes. Neither node alone can detect it.

Node 1

\[ rl_1[x] \]
\[ wl_2[x] \text{ (blocked)} \]

Node 2

\[ rl_2[y] \]
\[ wl_1[y] \text{ (blocked)} \]

• Timeout-based detection is popular. Its weaknesses are less important in the distributed case:
  – Aborts unnecessarily and some deadlocks persist too long
  – Possibly abort younger unblocked transaction to avoid cyclic restart
Oracle Deadlock Handling

• Uses a waits-for graph for single-server deadlock detection.
• The transaction that detects the deadlock is the victim.
• Uses timeouts to detect distributed deadlocks.
Fancier Dist’d Deadlock Detection

• Use waits-for graph cycle detection with a central deadlock detection server
  – More work than timeout-based detection, and there’s no evidence it performs better
  – Phantom deadlocks? - No, because each waits-for edge is an SG edge. So, WFG cycle => SG cycle (modulo spontaneous aborts)

• Path pushing (a.k.a. flooding) - Send paths $T_i \rightarrow \ldots \rightarrow T_k$ to each node where $T_k$ might be blocked.
  – Detects short cycles quickly
  – Hard to know where to send paths
  – Possibly too many messages
Locking Performance

• The following is oversimplified. We’ll revisit it.
• Deadlocks are rare.
  – Typically 1-2% of transactions deadlock.
• Locking performance problems are not rare.
• The problem is too much blocking.
• The solution is to reduce the “locking load”.
• Good heuristic – If more than 30% of transactions are blocked, then reduce the number of concurrent transactions.
Lock Conversions

• **Lock conversion** - upgrading an r-lock to a w-lock
  – e.g., $T_i = \text{read}(x) \ldots \text{write}(x)$

• This is one place where deadlocks are an issue
  – If two txns convert a lock concurrently, they’ll deadlock (both get an r-lock on x before either gets a w-lock).
  – To avoid the deadlock, a caller can get a w-lock first and down-grade to an r-lock if it doesn’t need to write.
  – We’ll see other solutions later.

• This is step 3 of the course project. Its main purpose is to ensure you understand the lock manager code.
What’s Coming in Part Two?

• Locking Performance
• More details on multigranularity locking
• Hot spot techniques
• Query-Update Techniques
• Phantoms
• B-Trees and Tree locking