Insight into

DBMS:

Design and Implementation

Chapter 5: Transaction Control

Ensure the Data Consistency

孔令波

mlinking@126.com

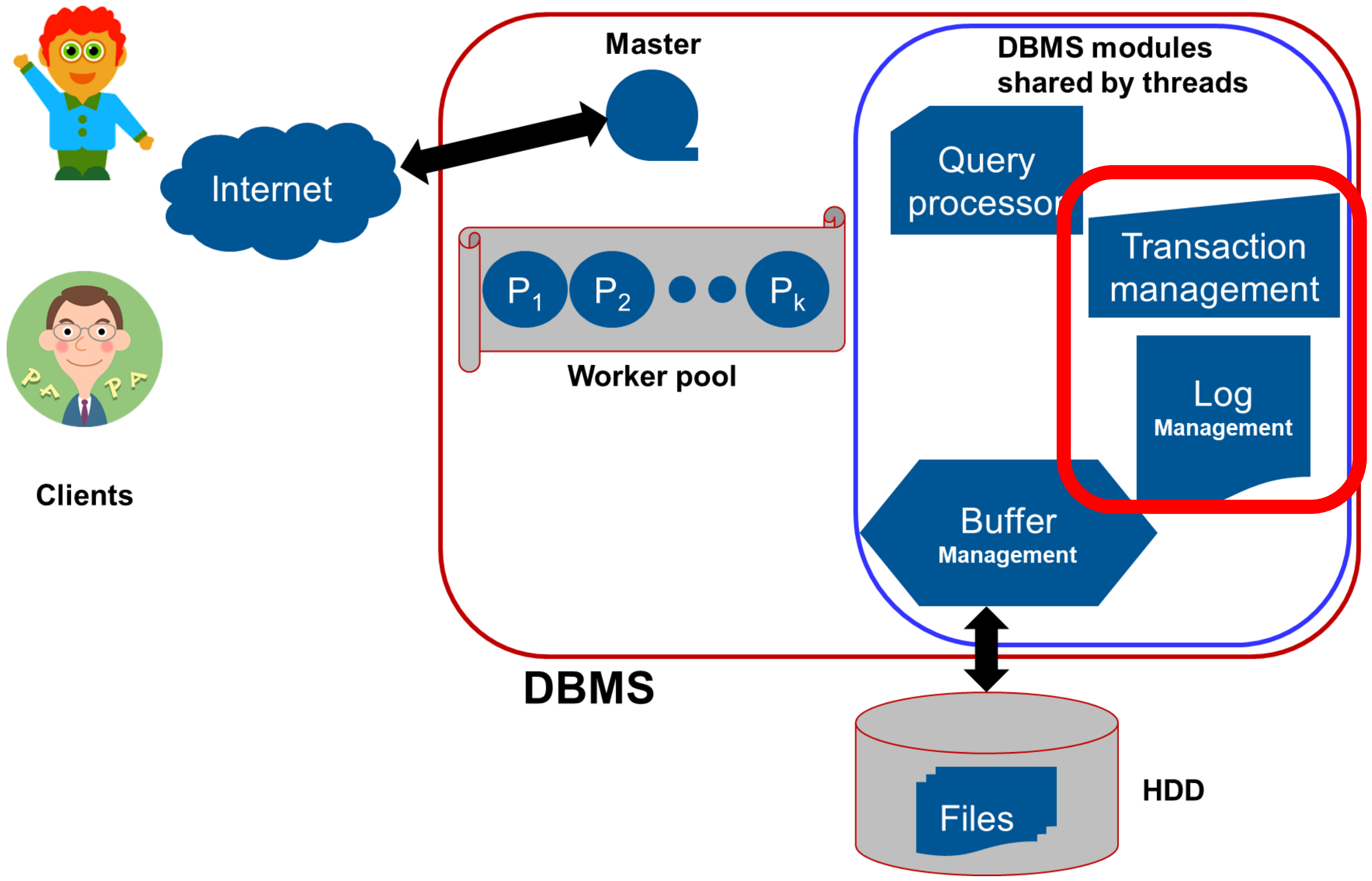
+86 15010255486

Outline of the chapters covered

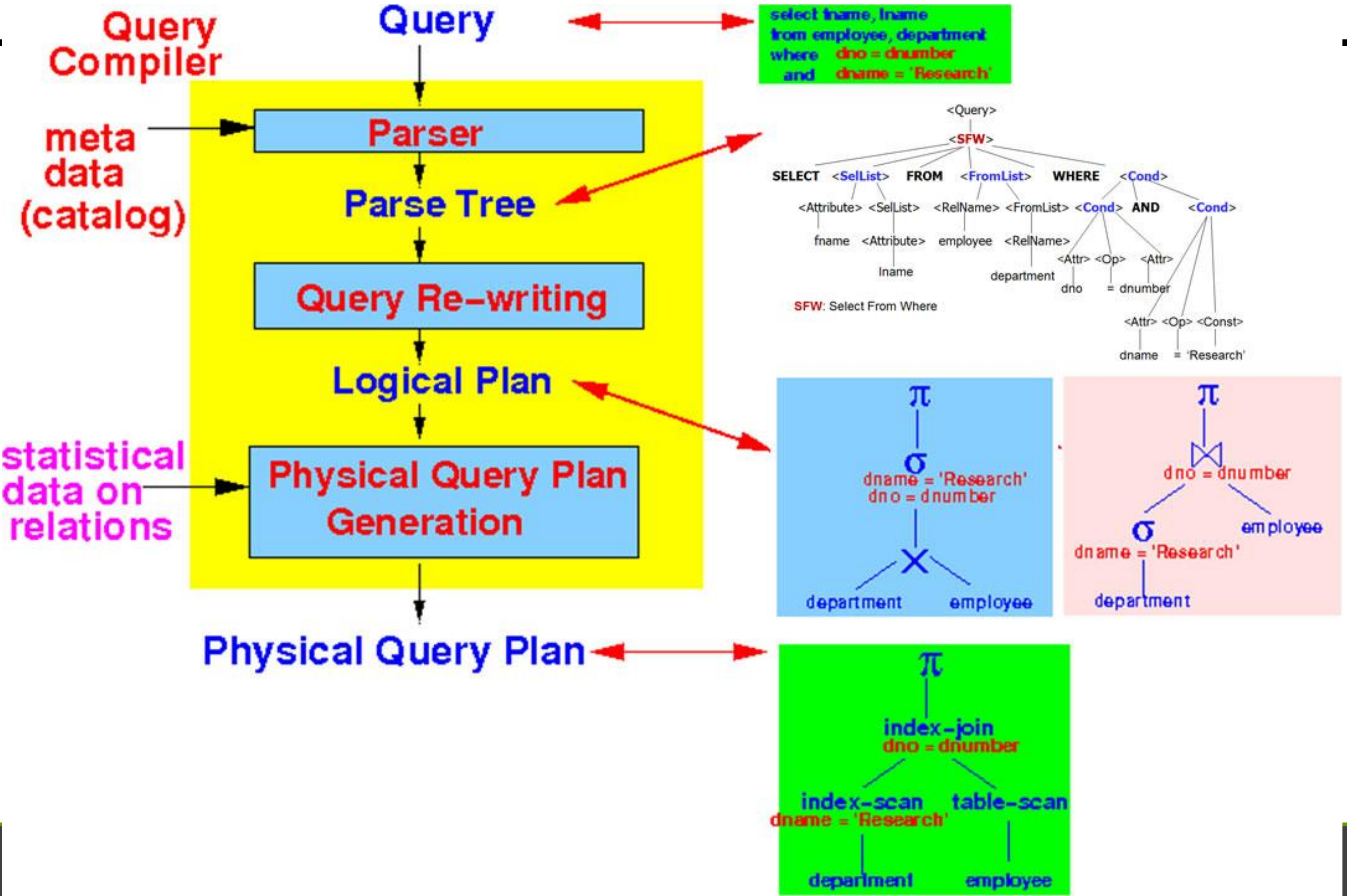
- **Introduction**
- **Overview of the projects**
- **Demonstration of Development environment**
  - Watch and practice by yourself
- **My understanding about (R)DBMS**
  - History and D&I
- **SQL translation with 2 conversions**
  - SQL → RA (Relational Algebra)
  - RA → Sequence of File operations
- **Transaction control**
- **Deeper**
  - File, (R)DBMS, ERP, DW, Big Data (No SQL, SQL again)
  - SQL on MPP and Hadoop (Greenplum, **HAWQ**)

Sit back, but not relax!

hammock [ˈhæmək]
n. 吊床; 吊铺


Master
DBMS modules shared by threads
Internet
Query processor
Transaction management
Log Management
P_1
P_2
P_k
Worker pool
Buffer Management
Clients
DBMS
Files
HDD

In short



```
select fname, lname
from employee, department
where   dno = dnumber
  and   dname = 'Research'
```

Chapter 4: Transaction Control/Management

- **My understanding about**
  - Transactions and Scheduler
    - Serializable schedule
  - Concurrency control
    - How to ensure the serializability?
      - ✓ Lock …
- **3rd project**
- **Advanced project in reading PostgreSQL**
  - Transaction Management in PostgreSQL

What is a Transaction?
- Like Critical Section in OS, but more complicated

- **A *logical* unit of work that must be either entirely completed or aborted – NOT ATOMICALLY**
- **Transactions could be**
  - DBMS SQL Prompt
    - Every SQL SELECT → **READ** action
    - Insert, Delete, Update → **WRITE** action
      - ✓ Maybe followed with COMMIT/ROLLBACK
        - » SQL> DELETE FROM CUSTOMERS WHERE AGE = 25; SQL> COMMIT;
        - » SQL> DELETE FROM CUSTOMERS WHERE AGE = 25; SQL> ROLLBACK;
  - Embedded SQL in HPLs
    - Maybe READ and WRITE together
      - ✓ Local variable ← **READ** by using SELECT-FROM-WHERE
      - ✓ Update locally
      - ✓ **WRITE** back

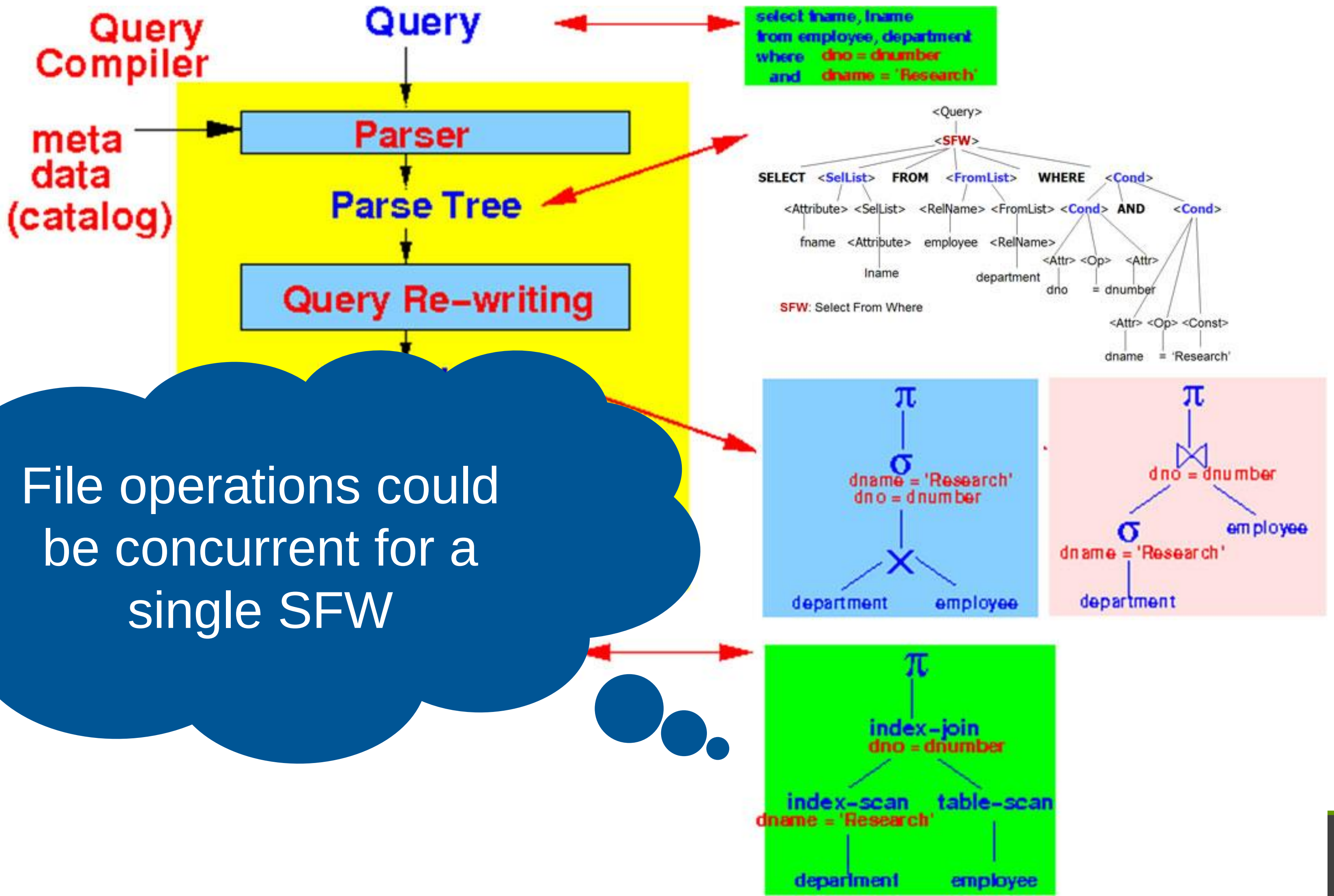
Example: Transaction 101

```
BEGIN;    --BEGIN TRANSACTION

UPDATE accounts SET balance = balance - 100.00 WHERE name =
   'Alice';

UPDATE branches SET balance = balance - 100.00 WHERE name =
   (SELECT branch_name FROM accounts WHERE name = 'Alice');

UPDATE accounts SET balance = balance + 100.00 WHERE name =
   'Bob';

UPDATE branches SET balance = balance + 100.00 WHERE name =
   (SELECT branch_name FROM accounts WHERE name = 'Bob');

 COMMIT;    --COMMIT WORK
```

Transfer $100 from Alice’s account to Bob’s account

Transaction – single SQL command – SFW

- You have learned SFW



```
select fname, lname
from employee, department
where   dno = dnumber
  and   dname = 'Research'
```

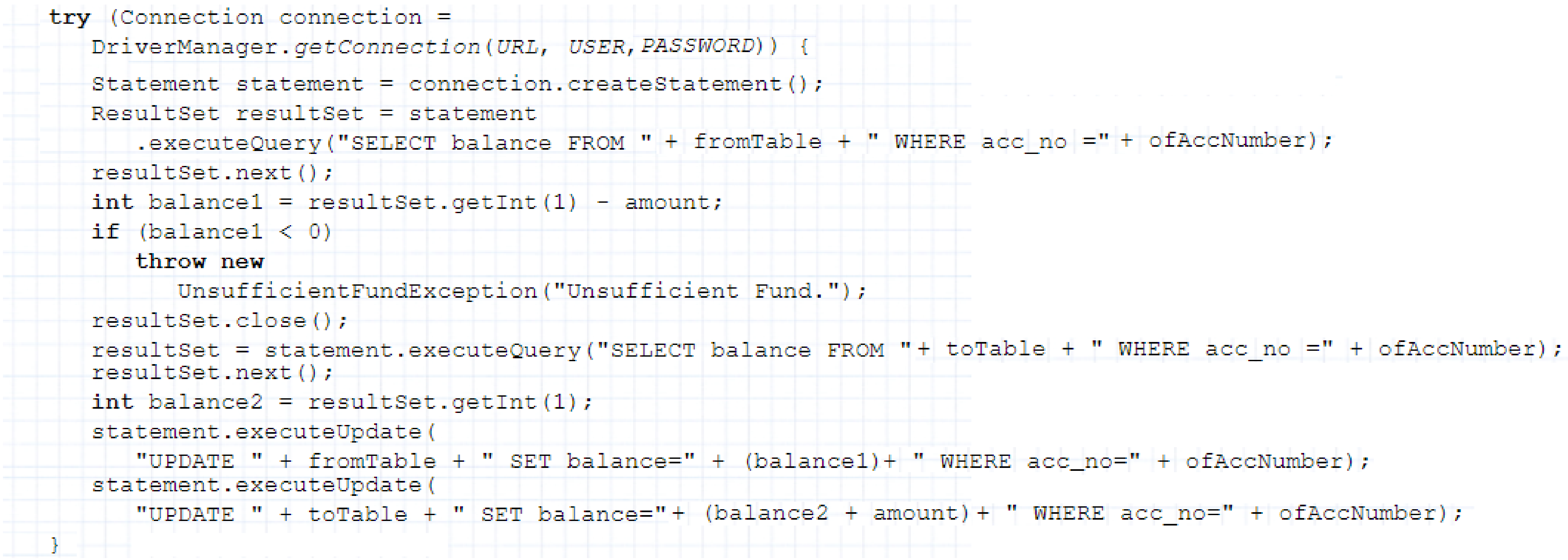
File operations could be concurrent for a single SFW

Transaction – single SQL command – UPDATE/DELETE

- **You could imagine the execution of UPDATE**
  - UPDATE Salary SET salary = salary + 1000 WHERE enrollmentYear >= 2000
  - SCAN the Salary table
    - For enrollmentYear >= 2000
      - ✓ salary = salary + 1000
  - The above UPDATE has to access all the records in the Salary table
  - DELETE may be similar

Transaction – compound SQL commands – Embedded SQL in HPL

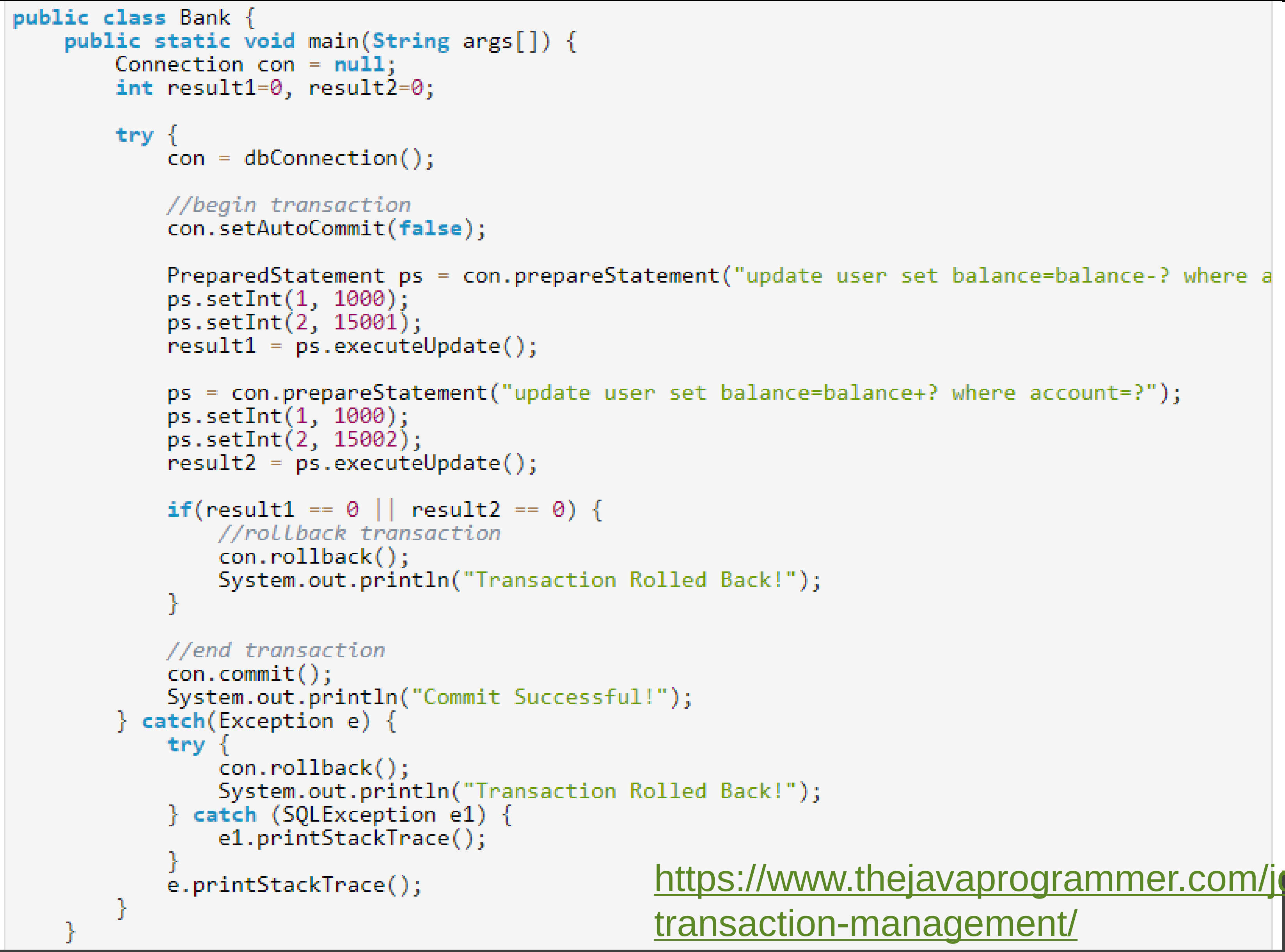
- Embedded SQL in HPL



```
try (Connection connection =
    DriverManager.getConnection(URL, USER,PASSWORD)) {
    Statement statement = connection.createStatement();
    ResultSet resultSet = statement
       .executeQuery("SELECT balance FROM " + fromTable + " WHERE acc_no =" + ofAccNumber);
    resultSet.next();
    int balance1 = resultSet.getInt(1) - amount;
    if (balance1 < 0)
       throw new
          UnsufficientFundException("Unsufficient Fund.");
    resultSet.close();
    resultSet = statement.executeQuery("SELECT balance FROM "+ toTable + " WHERE acc_no =" + ofAccNumber);
    resultSet.next();
    int balance2 = resultSet.getInt(1);
    statement.executeUpdate(
       "UPDATE " + fromTable + " SET balance=" + (balance1)+ " WHERE acc_no=" + ofAccNumber);
    statement.executeUpdate(
       "UPDATE " + toTable + " SET balance="+ (balance2 + amount)+ " WHERE acc_no=" + ofAccNumber);
}
```

https://www.developer.com/java/data/how-to-handle-transactions-with-the-jdbc-api.html

By using JDBC

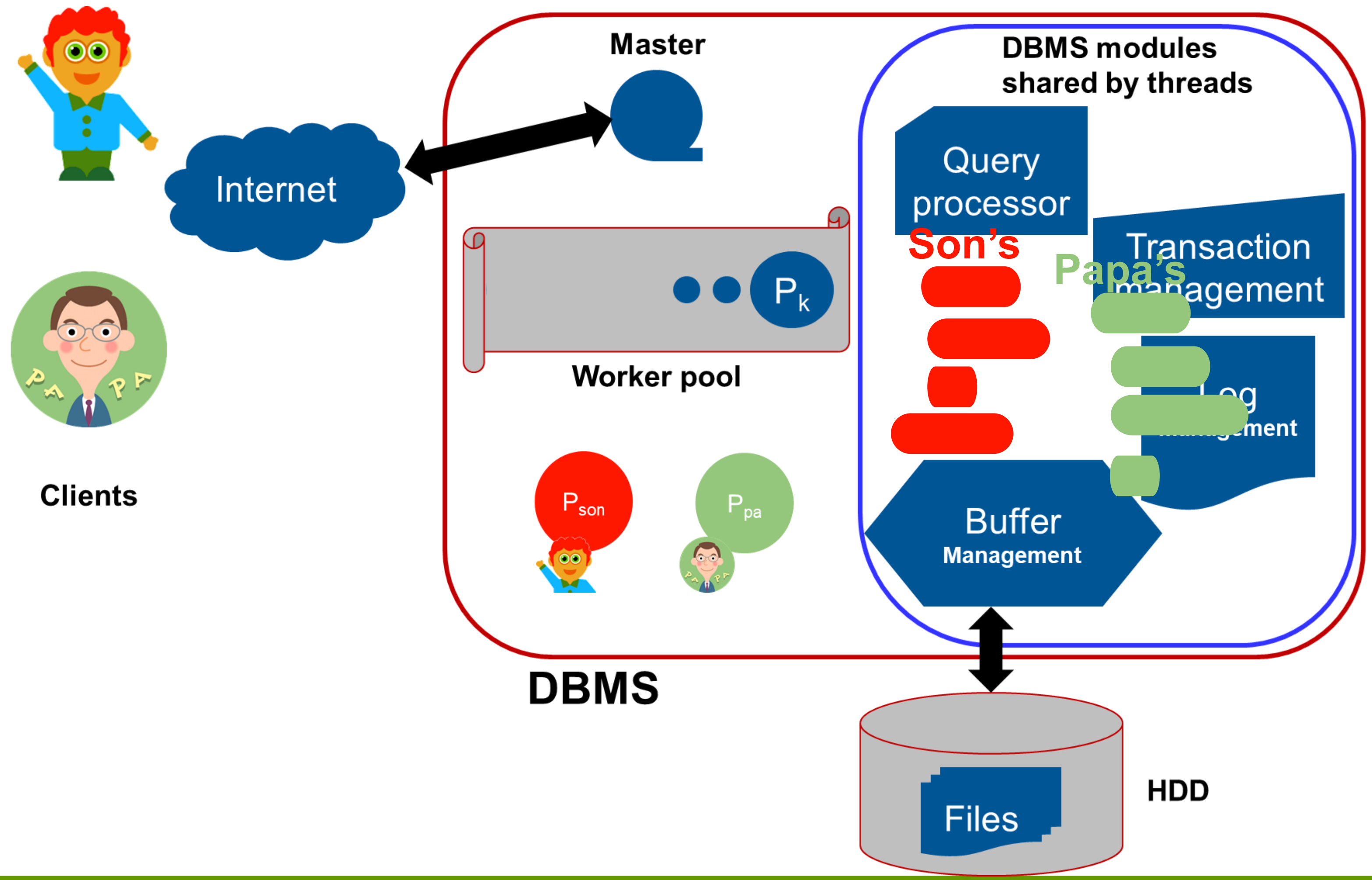


```
public class Bank {
    public static void main(String args[]) {
        Connection con = null;
        int result1=0, result2=0;

        try {
            con = dbConnection();

            //begin transaction
            con.setAutoCommit(false);

            PreparedStatement ps = con.prepareStatement("update user set balance=balance-? where a
            ps.setInt(1, 1000);
            ps.setInt(2, 15001);
            result1 = ps.executeUpdate();

            ps = con.prepareStatement("update user set balance=balance+? where account=?");
            ps.setInt(1, 1000);
            ps.setInt(2, 15002);
            result2 = ps.executeUpdate();

            if(result1 == 0 || result2 == 0) {
                //rollback transaction
                con.rollback();
                System.out.println("Transaction Rolled Back!");
            }

            //end transaction
            con.commit();
            System.out.println("Commit Successful!");
        } catch(Exception e) {
            try {
                con.rollback();
                System.out.println("Transaction Rolled Back!");
            } catch (SQLException e1) {
                e1.printStackTrace();
            }
            e.printStackTrace();
        }
    }
```

https://www.thejavaprogrammer.com/jdbc-transaction-management/

Those file operations (in a single Transaction or among Transactions) could be CONCURRENT

- But file operations of SQLs from concurrent clients/users could be interleaved



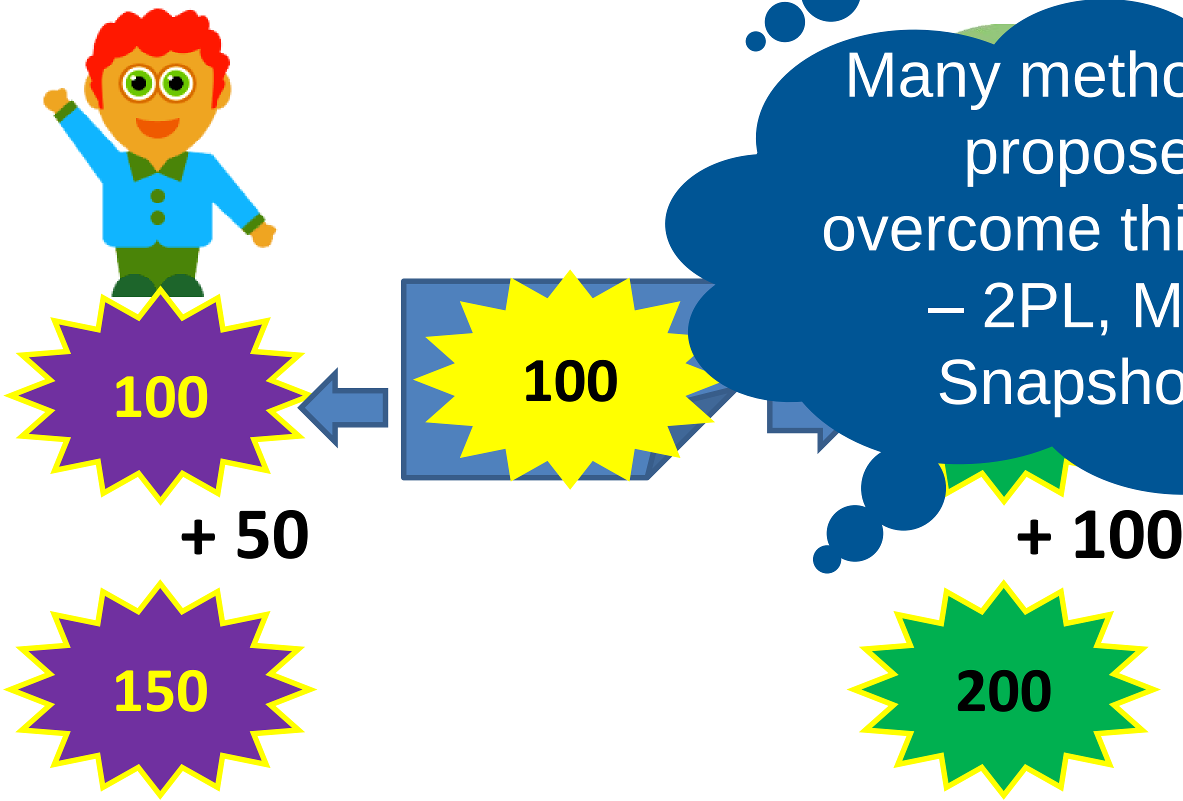
Risk exists …

- Data Inconsistency!

Depositing [

Transaction [事务] concept is used to describe this situation

Many methods were proposed to overcome this trouble! – 2PL, MVCC, Snapshot etc.



Transactions

With the trouble of Lost Update

T1	T2
Read(X)	
X = X - 5	
	Read(X)
	X = X + 5
Write(X)	
	Write(X)
COMMIT	
	COMMIT

- **T1 and T2 read X, both modify it, then both write it out**
  - The net effect of T1 and T2 should be no change on X
  - Only T2’s change is seen, however, so the final value of X has increased by 5

Transactions

With the trouble of Uncommitted Update

T1	T2
`Read(X)`	
`X = X - 5`	
`Write(X)`	
	`Read(X)`
	`X = X + 5`
	`Write(X)`
`ROLLBACK`	
	`COMMIT`

- **T2 sees the change to X made by T1, but T1 is rolled back**
  - The change made by T1 is undone on rollback
  - It should be as if that change never happened

Transactions

With the trouble of Inconsistency

T1	T2
Read(X)	
X = X - 5	
Write(X)	
	Read(X)
	Read(Y)
	Sum = X+Y
Read(Y)	
Y = Y + 5	
Write(Y)	

- **T1 doesn’t change the sum of X and Y, but T2 sees a change**
  - T1 consists of two parts – take 5 from X and then add 5 to Y
  - T2 sees the effect of the first, but not the second

How to ensure Data Consistency? – Serial execution of those transactions

- A *schedule* is a sequence of the operations by a set of concurrent transactions that preserves the order of operations in each of the individual transactions

- A *serial* [连续] schedule is a schedule where operations of each transaction are executed consecutively without any interleaved operations from other transactions (each transaction commits before the next one is allowed to begin)

- **Serial schedules**
  - Serial schedules are guaranteed to avoid interference and keep the database consistent
  - And it seems
    - **Mutual Exclusion** could ensure this by locking the involved tables during the execution of each transaction

- **However databases need concurrent access which means interleaving operations from different transactions**
  - Performance should be guaranteed for providing services for many clients

Serializability [可序列化]

- Two schedules are *equivalent* if they always have the same effect.
- **A schedule is *serializable* if it is equivalent to some serial schedule**.
- For example:
  - if two transactions only read some data items, then the order is which they do it is not important
  - If T1 reads and updates X and T2 reads and updates a different data item Y, then again they can be scheduled in any order.

Serial and Serializable

Interleaved Schedule

T1 Read(X)
T2 Read(X)
T2 Read(Y)
T1 Read(Z)
T1 Read(Y)
T2 Read(Z)

This schedule is serializable:

Serial Schedule

T2 Read(X)
T2 Read(Y)
T2 Read(Z)

T1 Read(X)
T1 Read(Z)
T1 Read(Y)

Conflict Serializable Schedule

Interleaved Schedule

T1 Read(X)
T1 Write(X)
T2 Read(X)
T2 Write(X)
T1 Read(Y)
T1 Write(Y)
T2 Read(Y)
T2 Write(Y)

This schedule is serializable, even though T1 and T2 read and write the same resources X and Y: they have a conflict

Serial Schedule

T1 Read(X)
T1 Write(X)
T1 Read(Y)
T1 Write(Y)

T2 Read(X)
T2 Write(X)
T2 Read(Y)
T2 Write(Y)

Conflict Serializability [冲突可串行化]

- **Two transactions have a conflict:**
  - NO If they refer to different resources
  - NO If they are reads
  - YES If at least one is a write and they use the same resource
- **A schedule is *conflict serializable* [冲突可串行化] if transactions in the schedule have a conflict but the schedule is still serializable**
- **Conflict serializable schedules are the main focus of concurrency control**
  - They allow for interleaving and at the same time they are guaranteed to behave as a serial schedule
- **Important questions:**
  - how to determine whether a schedule is conflict serializable
  - How to construct conflict serializable schedules

Precedence Graphs [前驱图]

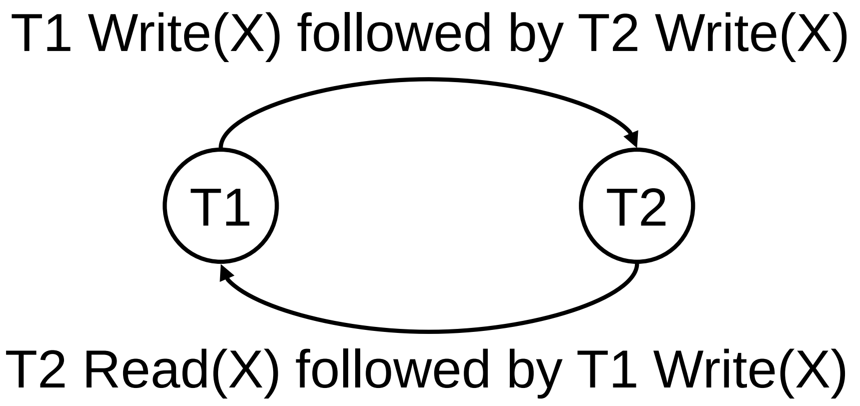
- **To determine if a schedule is conflict serializable we use a precedence graph**
  - Transactions are vertices of the graph
  - There is an edge from T1 to T2 if T1 must happen before T2 in any equivalent serial schedule

- **Edge T1 $\rightarrow$ T2 if in the schedule we have at least one of following situations:**
  1. T1 Read(R) followed by T2 **Write**(R) for the same resource R
  2. T1 **Write**(R) followed by T2 Read(R)
  3. T1 **Write**(R) followed by T2 **Write**(R)

- **The schedule is serializable if there are no cycles**

Precedence Graph Example

- The lost update schedule has the precedence graph:


T1 Write(X) followed by T2 Write(X)
T1
T2
T2 Read(X) followed by T1 Write(X)

T1	T2
Read(X)	
X = X - 5	
	Read(X)
	X = X + 5
Write(X)	
	Write(X)
COMMIT	
	COMMIT

Precedence Graph Example

- **No cycles: conflict serializable schedule**

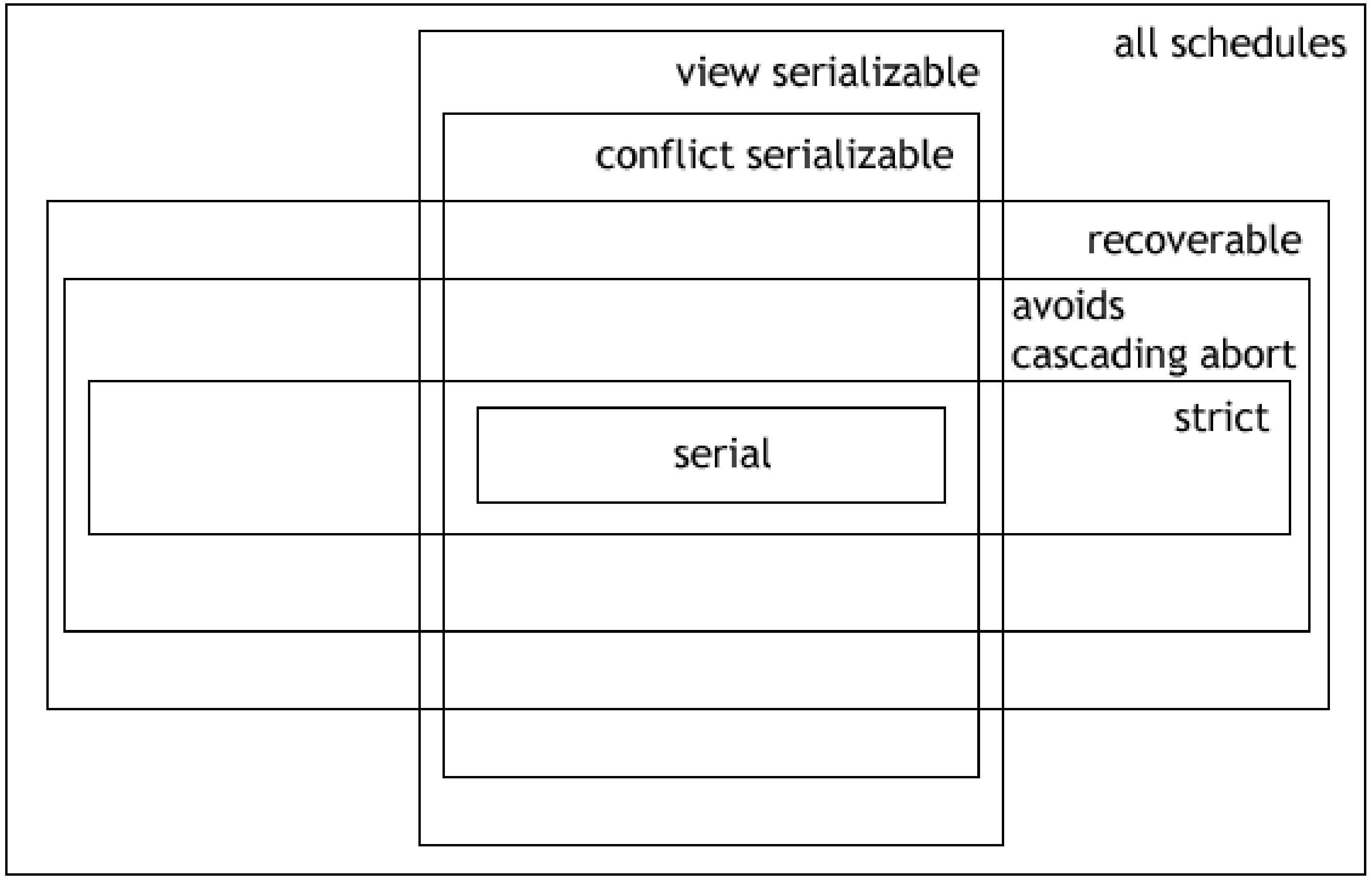
T1 reads X before T2 writes X and
T1 writes X before T2 reads X and
T1 writes X before T2 writes X

$$\begin{tikzcd} T1 \arrow[rr, bend left] & & T2 \end{tikzcd}$$

T1	T2
`Read(X)`	
`Write(X)`	
	`Read(X)`
	`Write(X)`

Do you remember the Detect & Recover strategy in OS? – YES, similar idea later

▢ Hierarchical relationship between serializability classes

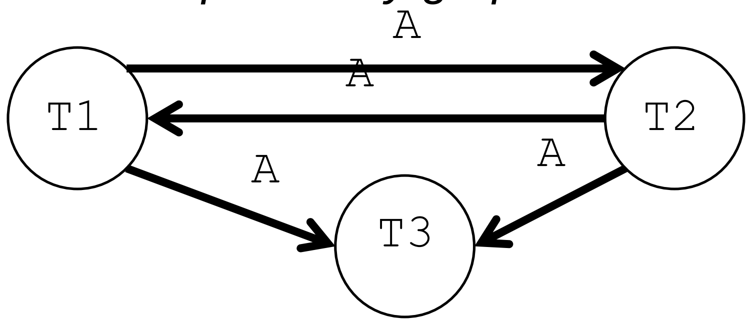


Srializability ≠ Conflict Serializability

- Following schedule is not conflict serializable

```
T1:R(A),     W(A),
T2:     W(A),
T3:                WA
```

Dependency graph



- However, the schedule is serializable since its output is equivalent with the following serial schedule

```
T1:R(A),W(A),
T2:          W(A),
T3:               WA
```

- Note: deciding whether a schedule is serializable (not conflict-serializable) is NP-complete

Concurrency Control

- **Now the challenge is**
  - How to ensure Serializability while not locking the tables
- **The coordination of the simultaneous execution of transactions in a multiprocessing database is known as *concurrency control***
  - The objective of concurrency control is to ensure the serializability of transactions in a multiuser database environment
  - Important → simultaneous execution of transactions over a shared database can create several data integrity and consistency problems
- **The three main problems are:**
  - lost updates, uncommitted data, inconsistent retrievals

Concurrency v.s Data Inconsistency

- **Large databases are used by many people**
  - Many transactions to be run on the database
  - It is desirable to let them run at the same time as each other
  - Need to preserve isolation

- **If we don't allow for concurre sequentially**
  - Have a queue of transactions
  - Long transactions (e.g. backups) will make others wait for long periods

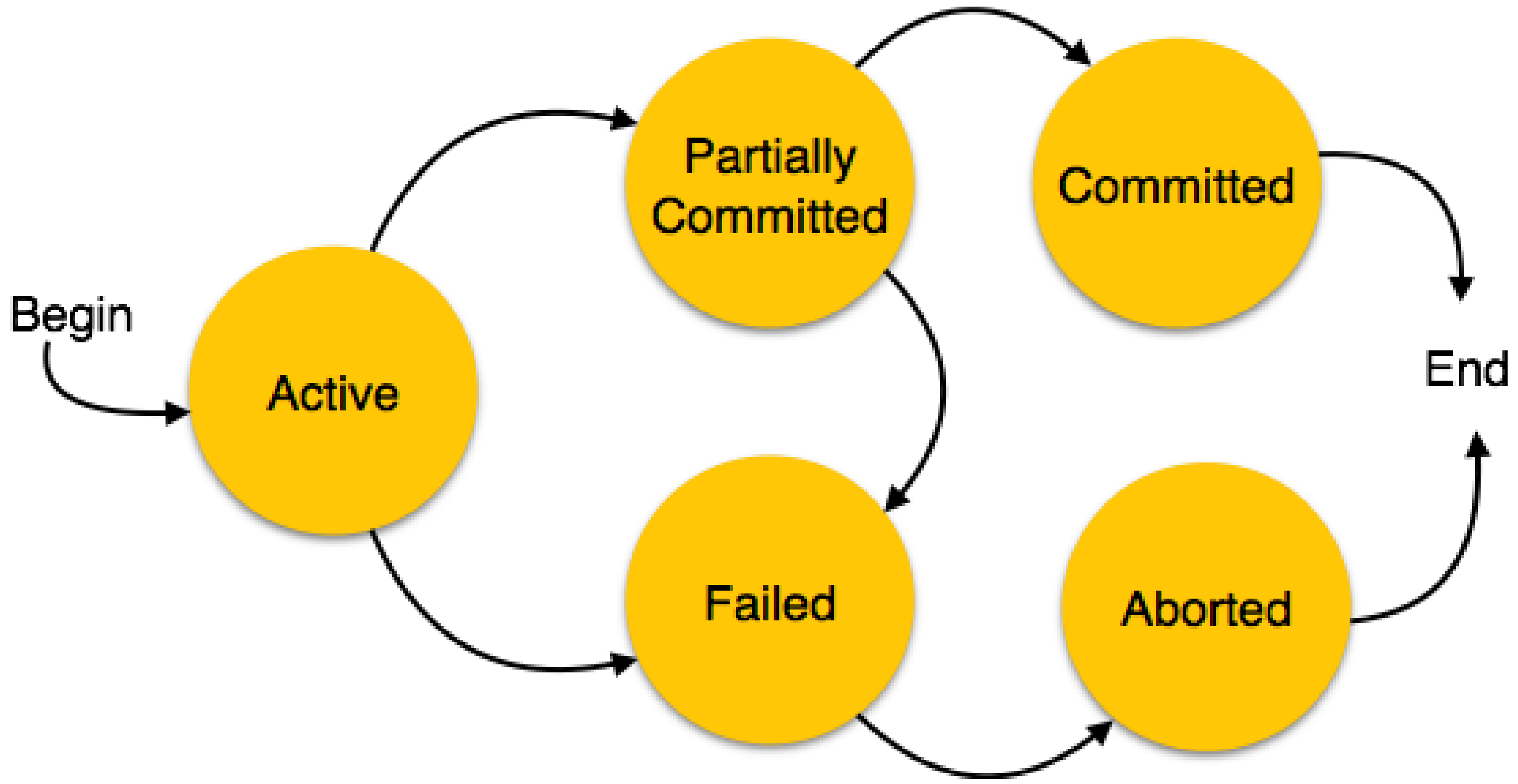
We need Concurrency Control/Transaction Control

ACID Properties

- **Transactions should be atomic**
  - Either all or none of a transaction’s actions are carried out
- **A transaction must preserve DB consistency**
  - The responsibility of the DB designers and the users
- **A transaction should make sense in isolation**
  - Transactions should stand on their own and
  - Should be protected from the effects of other concurrently executing transactions
- **Transactions should be durable**
  - Once a transaction is successfully completed, its effects should persist, even in the event of a system crash

States of Transactions

- A transaction in a database can be in one of the following states



https://www.tutorialspoint.com/dbms/dbms_transaction.htm

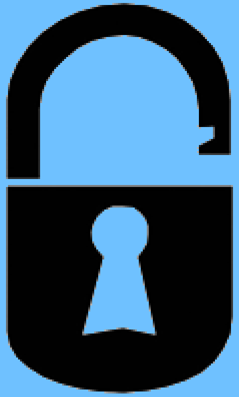
Chapter 4: Transaction Control/Management

- **My understanding about**
  - Transactions and Scheduler
    - Serializable schedule
  - Concurrency control
    - How to ensure the serializability?
      - ✓ Lock, Lock table, Scheduler, …
      - ✓ Two Phased Locking protocol/strategy
- **3rd project**
- **Advanced project in reading PostgreSQL**
  - Transaction Management in PostgreSQL

LOCK (to ensure Mutual Exclusion) learned in OS is still helpful

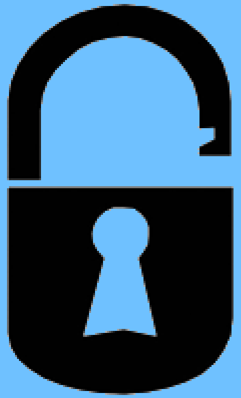
- The general layout is of lock solution is:

```
do {
        acquire lock
          critical section
        release lock
        remainder section
} while (TRUE);
```



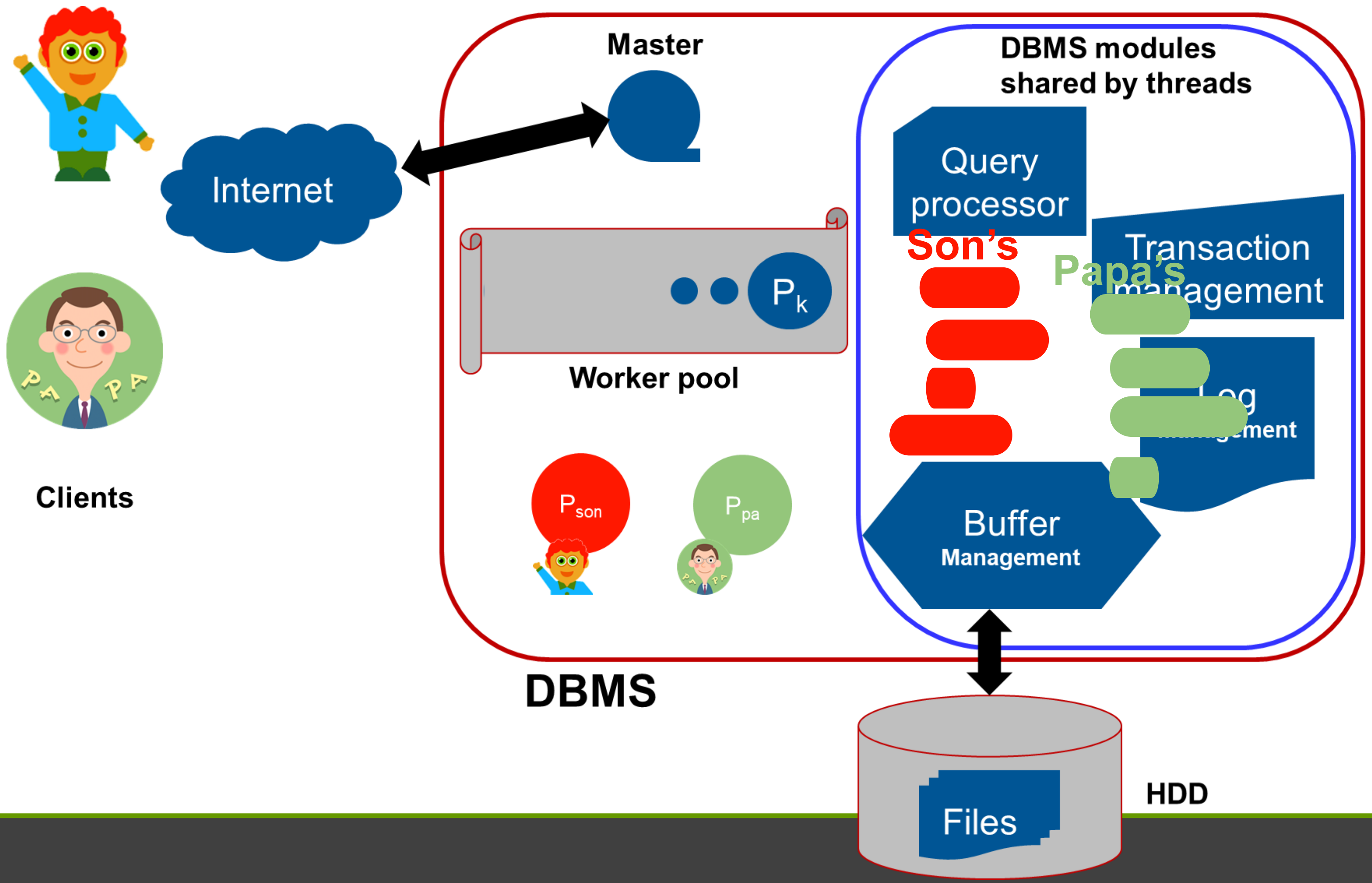
LOCK learned in OS is still helpful

- **Software solutions –**
  - algorithms who's correctness does not rely on any other assumptions.
- **Hardware solutions –**
  - rely on some special machine instructions.
- **Operating System solutions – Semaphore**
  - provide some functions and data structures to the programmer through system/library calls.
- **Programming Language solutions –**
  - Linguistic constructs provided as part of a language.



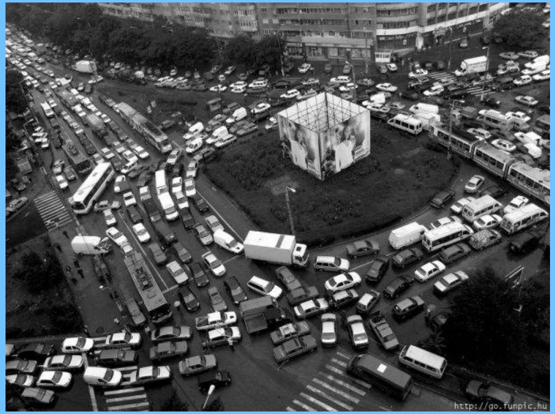
Locking the involved tables is Not effective for DBMS Transactions

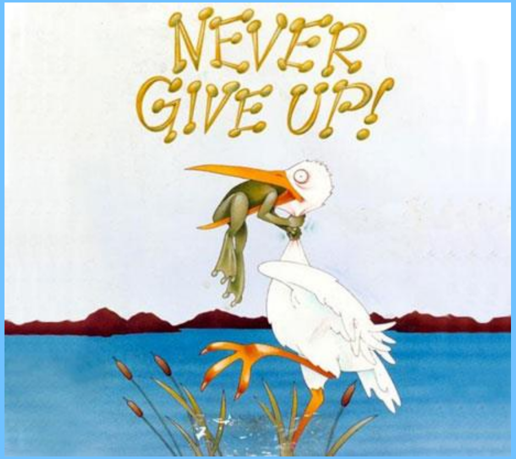
- The access to the data elements in DBMS by concurrent transactions is dynamic



Deadlock risk should also be considered

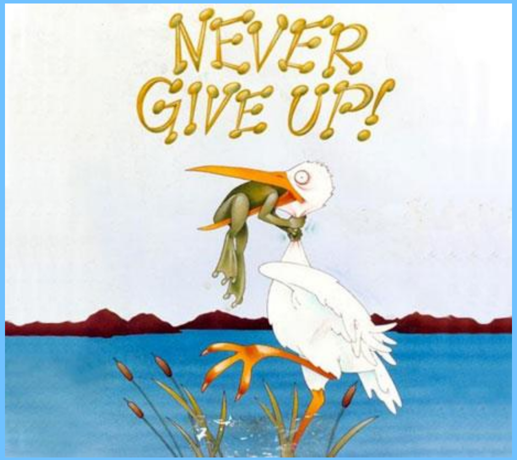
- A deadlock is a situation wherein **two or more competing actions are waiting for the other to finish, and thus neither ever does**.
- **A set of blocked processes each holding some resources and waiting to acquire a resource held by another process in the set.**
- **None of the processes can proceed or back-off (release resources it owns)**





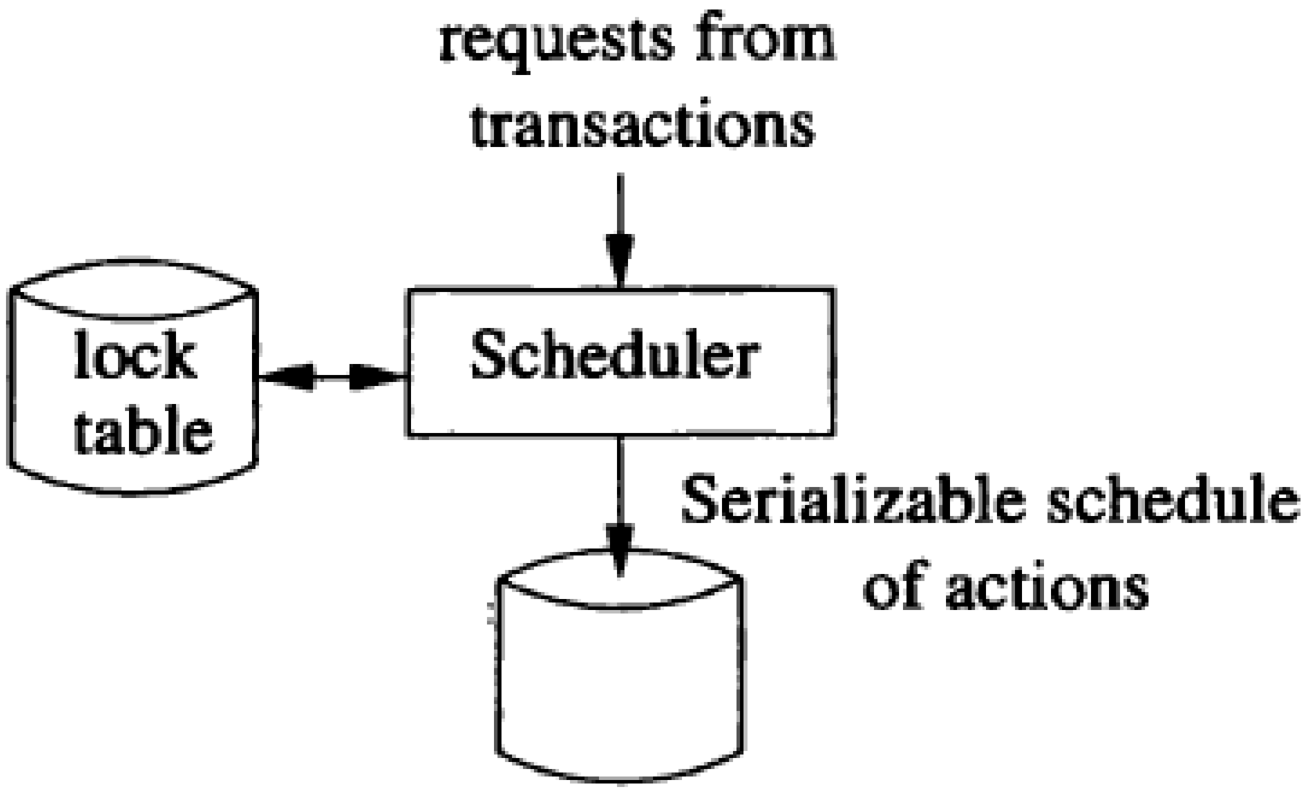
Deadlock risk should also be considered

- Methods for Handling Deadlocks
  - Providing **enough resources**
  - Staying Safe
    - **Preventing** Deadlocks
    - **Avoiding** Deadlocks
  - Living Dangerously
    - Let the deadlock happen, then **detect** it and **recover** from it.
    - **Ignore** the risks



Elements to carry out Transaction Control

- Lock table, Scheduler



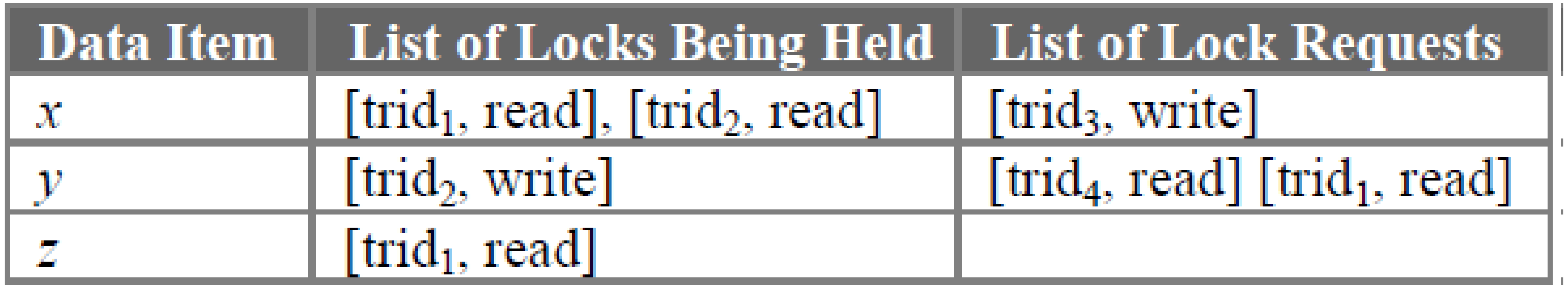
Phil Bernstein at Microsoft Research

Lock Manager in DBMS

- **A lock manager is a component that provides the operations**
  1. Lock(transaction-id, data-item, lock-mode) - Set a lock with mode lock-mode on behalf of transaction transaction-id on data-item.
  2. Unlock(transaction-id, data-item) - Release transaction transaction-id's lock on data-item.
  3. Unlock(transaction-id) - Release all of transaction transaction-id's locks.
- **It implements these operations by storing locks in a lock table.**
  - This is a **low-level data structure** in main memory, much like a control table in an operating system

https://courses.cs.washington.edu/courses/csep545/01wi/lectures/Chapt6V3.pdf

https://courses.cs.washington.edu/courses/csep545/01wi/lectures/Chapt6V3.pdf



Data Item	List of Locks Being Held	List of Lock Requests
x	[trid$_1$, read], [trid$_2$, read]	[trid$_3$, write]
y	[trid$_2$, write]	[trid$_4$, read] [trid$_1$, read]
z	[trid$_1$, read]	

Figure 6.2 A Lock Table Each entry in a list of locks held or requested is of the form [transaction-id, lock- mode]

To execute a Lock operation, the lock manager sets the lock if no conflicting lock is held by another transaction. For example, in Figure 6.2, the lock manager would grant a request by T$_2$ for a read lock on *z*, and would therefore add [trid$_2$, read] to the list of locks being held on *z*.

If the lock manager receives a lock request for which a conflicting lock *is* being held, the lock manager adds a request for that lock, which it will grant after conflicting locks are released. In this case, the transaction that requires the lock is blocked until its lock request is granted. For example, a request by T$_2$ for a write lock on *z* would cause [trid$_2$, write] to be added to *z*'s list of lock requests and T$_2$ to be blocked.

https://courses.cs.washington.edu/courses/csep545/01wi/lectures/Chapt6V3.pdf

- Data item identifiers are usually required to be a fixed length, say 32 bytes.
  - It is up to the caller of the lock manager to compress the name of the object to be locked (e.g., a table, page, or row) into a data item identifier of the length supported by the lock manager.
- Any data item in a database can be locked, but only a small fraction of them are locked at any one time, because only a small fraction of them are accessed at any one time by a transaction that's actively executing.
  - Therefore, instead of allocating a row in the lock table for every possible data item identifier value, **the lock table is implemented as a hash table**, whose size is somewhat larger than the maximal number of locks that are held by active transactions.

https://courses.cs.washington.edu/courses/csep545/01wi/lectures/Chapt6V3.pdf

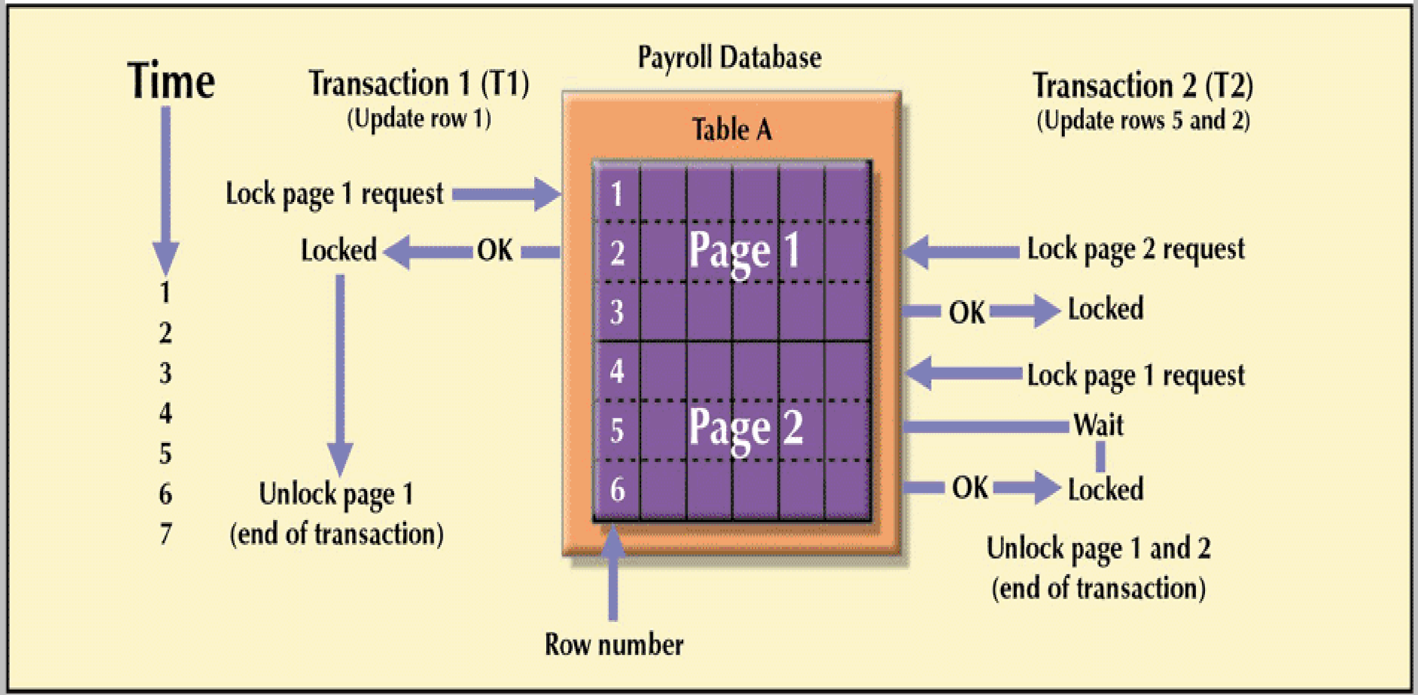
- **Lock operations on each data item must be atomic relative to each other. Otherwise, two conflicting lock requests might incorrectly be granted at the same time.**
  - For example, if two requests to set a write lock on v execute concurrently, they might both detect that v is unlocked before either of them set the lock.
  - To avoid this bad behavior, the lock manager executes each lock or unlock operation on a data item completely before starting the next one on that data item.
  - That is, it executes lock and unlock operations on each data item atomically with respect to each other.
    - Note that lock operations on different data items can safely execute concurrently.

Lock Granularity

- **Indicates the level of lock use**
- **Locking can take place at the following levels:**
  - Database-level lock
    - Entire database is locked
  - Table-level lock
    - Entire table is locked
  - Page-level lock
    - Entire diskpage is locked
  - Row-level lock
    - Allows concurrent transactions to access different rows of the same table, even if the rows are located on the same page
  - Field-level lock
    - Allows concurrent transactions to access the same row, as long as they require the use of different fields (attributes) within that row

Page-Level Lock

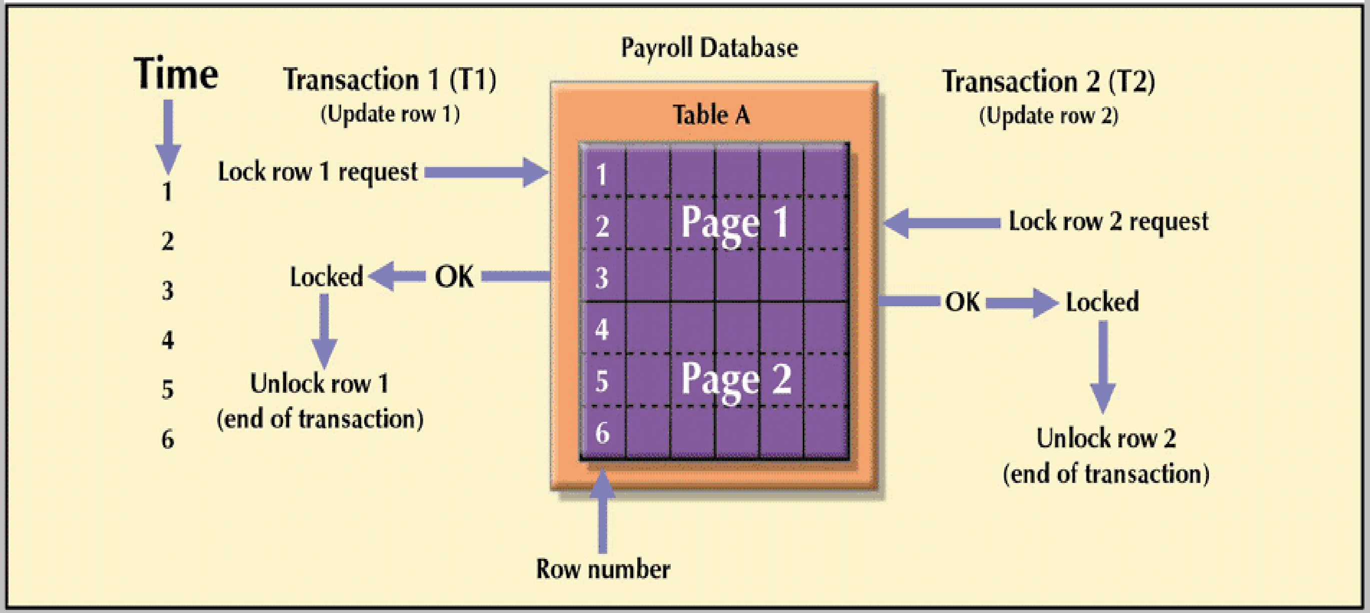
FIGURE 9.5 AN EXAMPLE OF A PAGE-LEVEL LOCK



- An entire disk page is locked (a table can span several pages and each page can contain several rows of one or more tables)
- Most frequently used multi-user DBMS locking method

Row-Level Lock

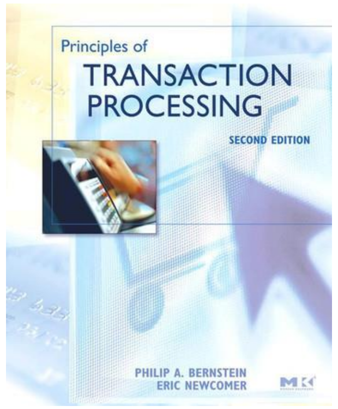
FIGURE 9.6 AN EXAMPLE OF A ROW-LEVEL LOCK

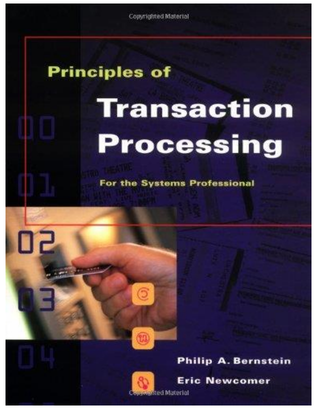


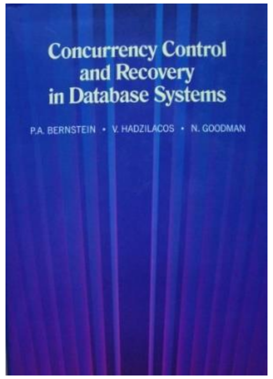
- Concurrent transactions can access different rows of the same table even if the rows are located on the same page
- Improves data availability but with high overhead (each row has a lock that must be read and written to)

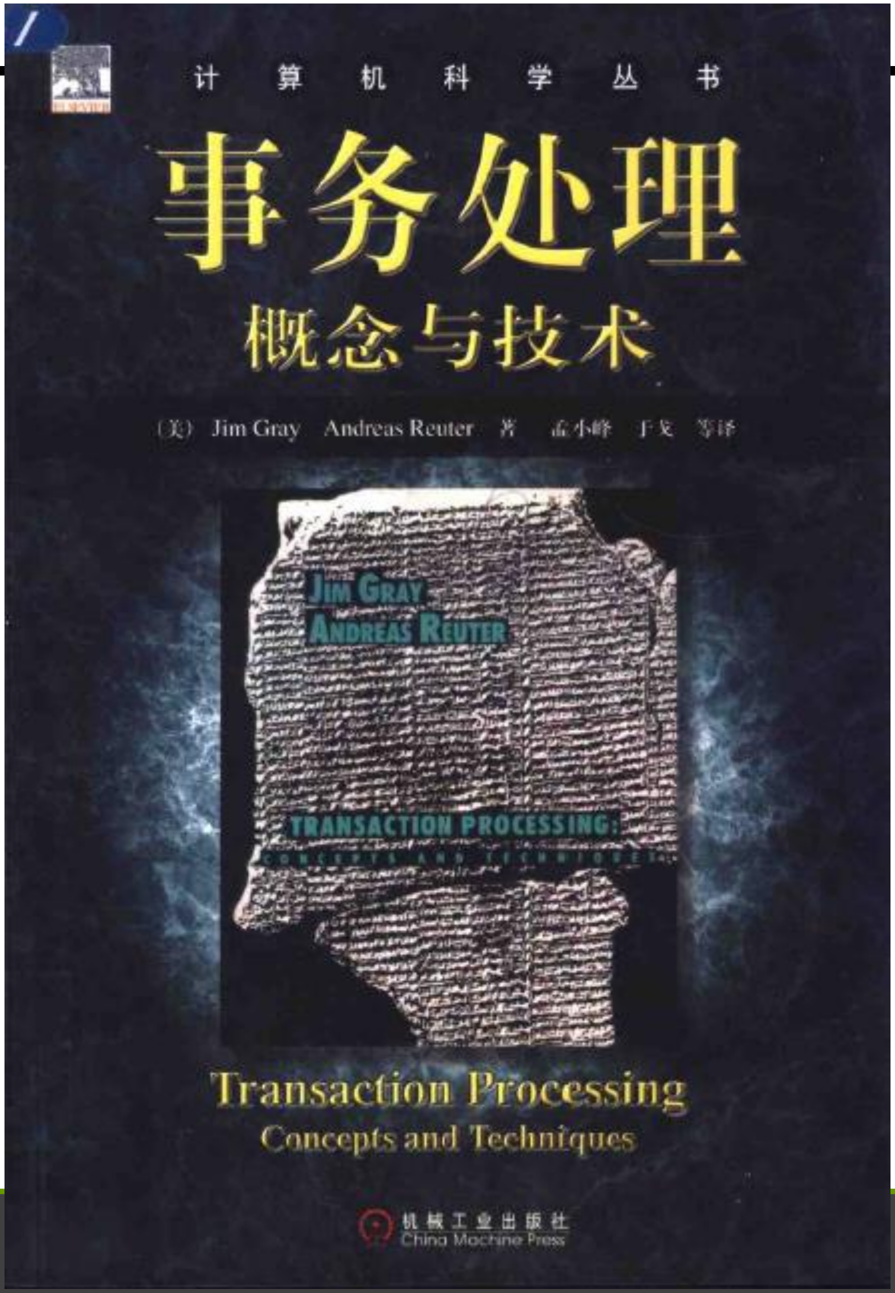
Phil Bernstein
at Microsoft Research

- 事务处理原理(第2版)
- Principles of Transaction Processing, Sceond Edition
- 2010年12月1日
- 伯恩斯坦(Philip A.Bernstein) (作者), 纽克默(Eric Newcomer) (作者),
- 战晓苏 (译者), 马严 (译者)

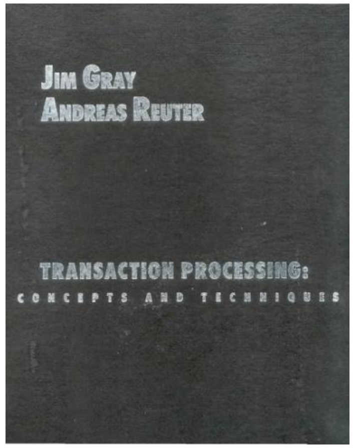


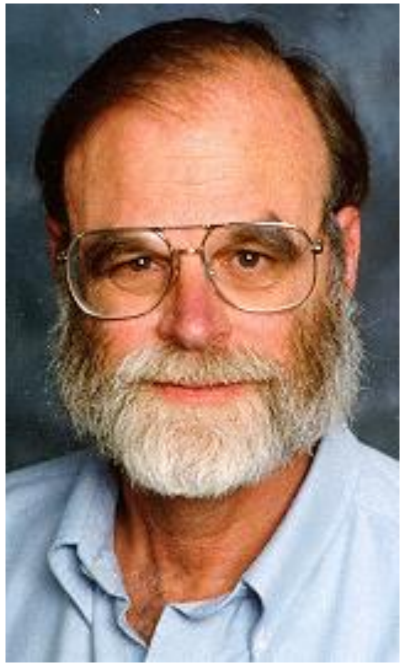






- Transaction Processing
- Jim Gray, Andreas Reuter





Chapter 4: Transaction Control/Management

- **My understanding about**
  - Transactions and Scheduler
    - Serializable schedule
  - Concurrency control
    - How to ensure the serializability?
      - ✓ Lock, Lock table, Scheduler, …
      - ✓ Two Phased Locking protocol/strategy
- **3rd project**
- **Advanced project in reading PostgreSQL**
  - Transaction Management in PostgreSQL

How to ensure Conflict Serializability?

- **Are Locks Enough?**
- ☹ **Sadly… no.**
  - Locks alone do not guarantee serializability.
    - ➢Imagine the interleaved locks
- ☺ **Happily… there is a simple protocol (2PL) which uses locks to guarantee serializability.**
  - Easy to explain
  - Easy to implement
  - Easy to enforce

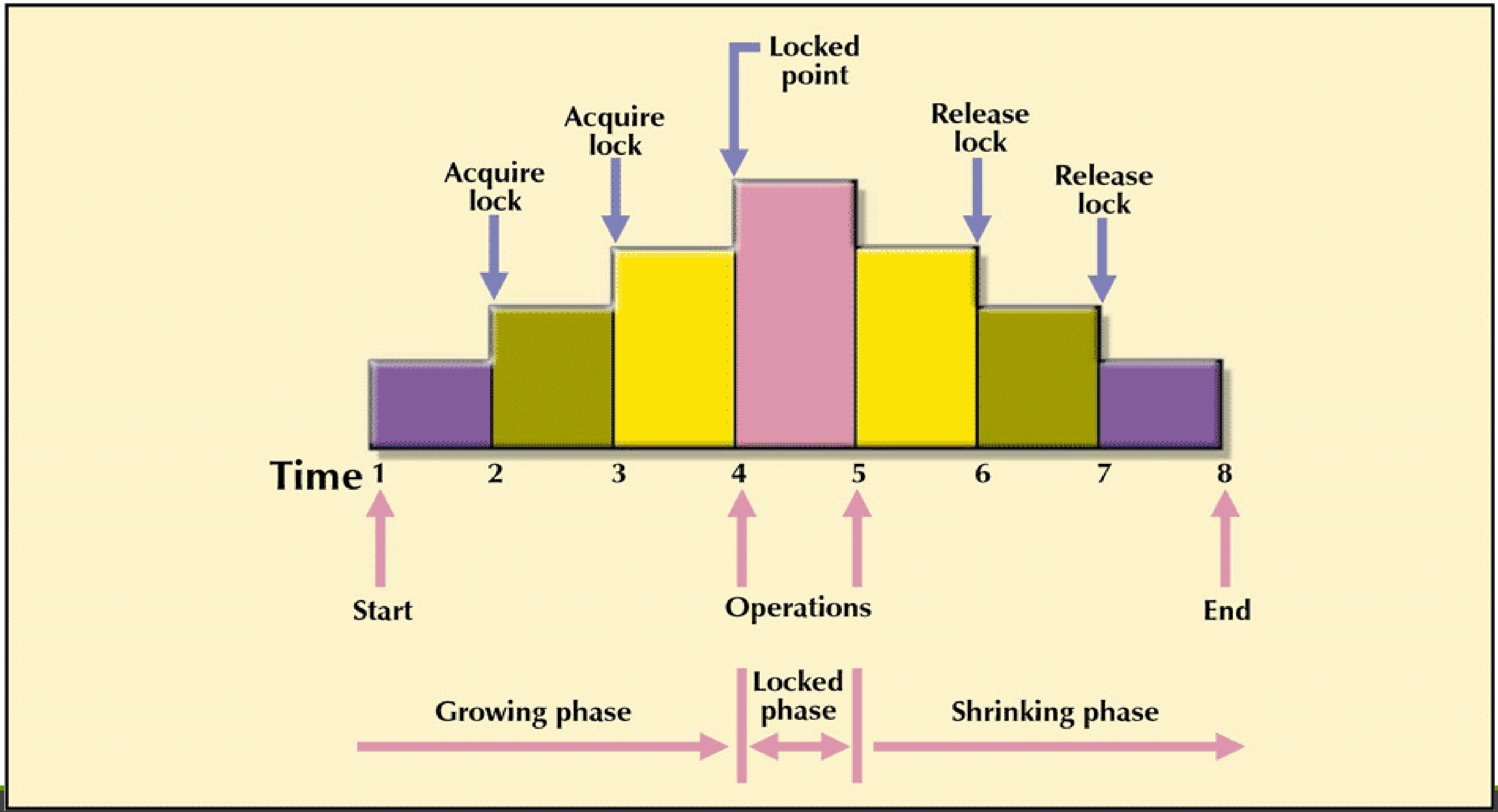
https://courses.cs.washington.edu/courses/cse444/97au/slides/n-concurrency.ppt

2 Phase Lock is fundamental

- Derived by the fact/law/rule
  - For each transaction, “No UNLOCK before any LOCK” could ensure Conflict Serializability with other transactions
- 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*.

FIGURE 9.7 Two-Phase Locking Protocol



https://courses.cs.washington.edu/courses/csep545/01wi/lectures/Chapt6V3.pdf

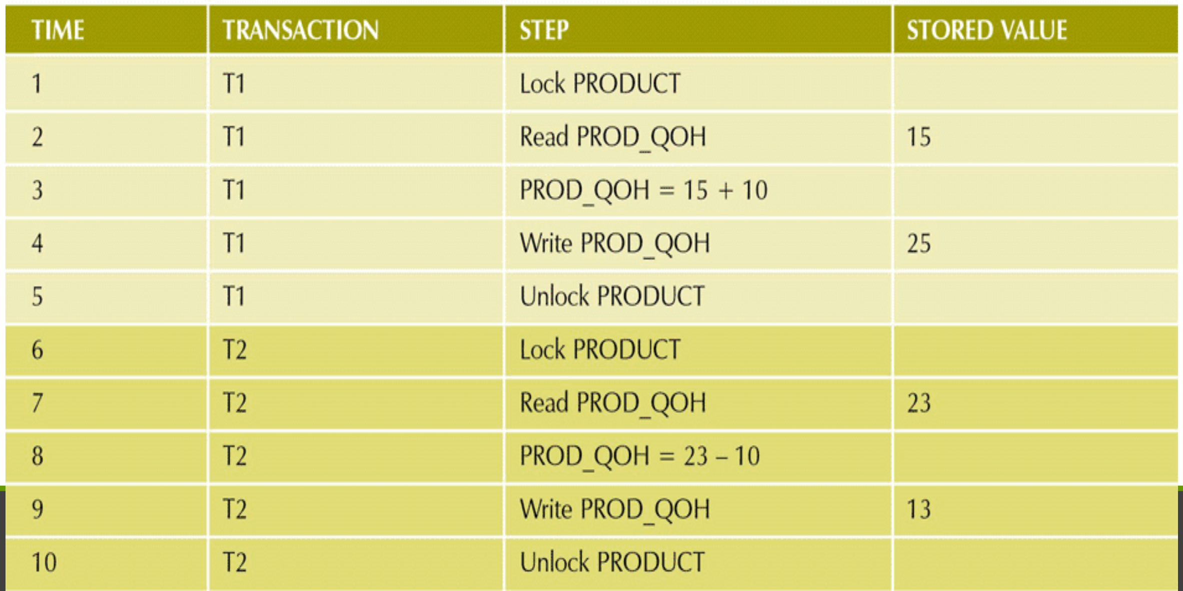
- **Two-Phase Locking**
  - Theorem
    - **If all transactions in an execution are two-phase locked, then the execution is serializable.**

- **Despite the simple intuition behind locking, there are no simple proofs of the Two-Phase Locking Theorem.**
  - The original proof by Eswaran et al. appeared in 1976 and was several pages long. The simplest proof we know of is by **Ullman**.

Protocols for locking

- Naïve – **Binary Lock**
  - Has only two states: locked (1) or unlocked (0)
- Eliminates “Lost Update” problem – the lock is not released until the write statement is completed
  - Can not use PROD_QOH until it has been properly updated
- Considered too restrictive to yield optimal concurrency conditions as it locks even for two READs when no update is being done

TABLE 9.10 An Example of a Binary Lock



TIME	TRANSACTION	STEP	STORED VALUE
1	T1	Lock PRODUCT	
2	T1	Read PROD_QOH	15
3	T1	PROD_QOH = 15 + 10	
4	T1	Write PROD_QOH	25
5	T1	Unlock PRODUCT	
6	T2	Lock PRODUCT	
7	T2	Read PROD_QOH	23
8	T2	PROD_QOH = 23 – 10	
9	T2	Write PROD_QOH	13
10	T2	Unlock PRODUCT	

Protocols for locking

- **Shared/Exclusive Locks – flexible but with risks**
  - Exclusive lock
    - Access is specifically reserved for the transaction that locked the object
      - ✓ Must be used when the potential for conflict exists – when a transaction wants to update a data item and no locks are currently held on that data item by another transaction
    - *Granted if and only if no other locks are held on the data item*
  - Shared lock
    - Concurrent transactions are granted Read access on the basis of a common lock
    - Issued when a transaction wants to read data and no exclusive lock is held on that data item
      - ✓ Multiple transactions can each have a shared lock on the same data item if they are all just reading it
  - Mutual Exclusive Rule
    - Only one transaction at a time can own an exclusive lock in the same object

Example

- **T1 follows 2PL protocol**
  - All of its locks are acquired before it releases any of them
- **T2 does not**
  - It releases its lock on X and then goes on to later acquire a lock on Y

T1	T2
read-lock(X)	read-lock(X)
Read(X)	Read(X)
write-lock(Y)	unlock(X)
unlock(X)	write-lock(Y)
Read(Y)	Read(Y)
Y = Y + X	Y = Y + X
Write(Y)	Write(Y)
unlock(Y)	unlock(Y)

- **Serializability Theorem**
  - Any schedule of two-phased transactions is conflict serializable
- **Lost Update can't happen with 2PL**

	T1	T2	
read-lock(X)	Read(X)		
	X = X - 5		
		Read(X)	read-lock(X)
		X = X + 5	
cannot acquire write-lock(X): T2 has read-lock(X)	Write(X)		
		Write(X)	cannot acquire write-lock(X): T1 has read-lock(X)
	COMMIT		
		COMMIT	

Uncommitted Update cannot happen with 2PL

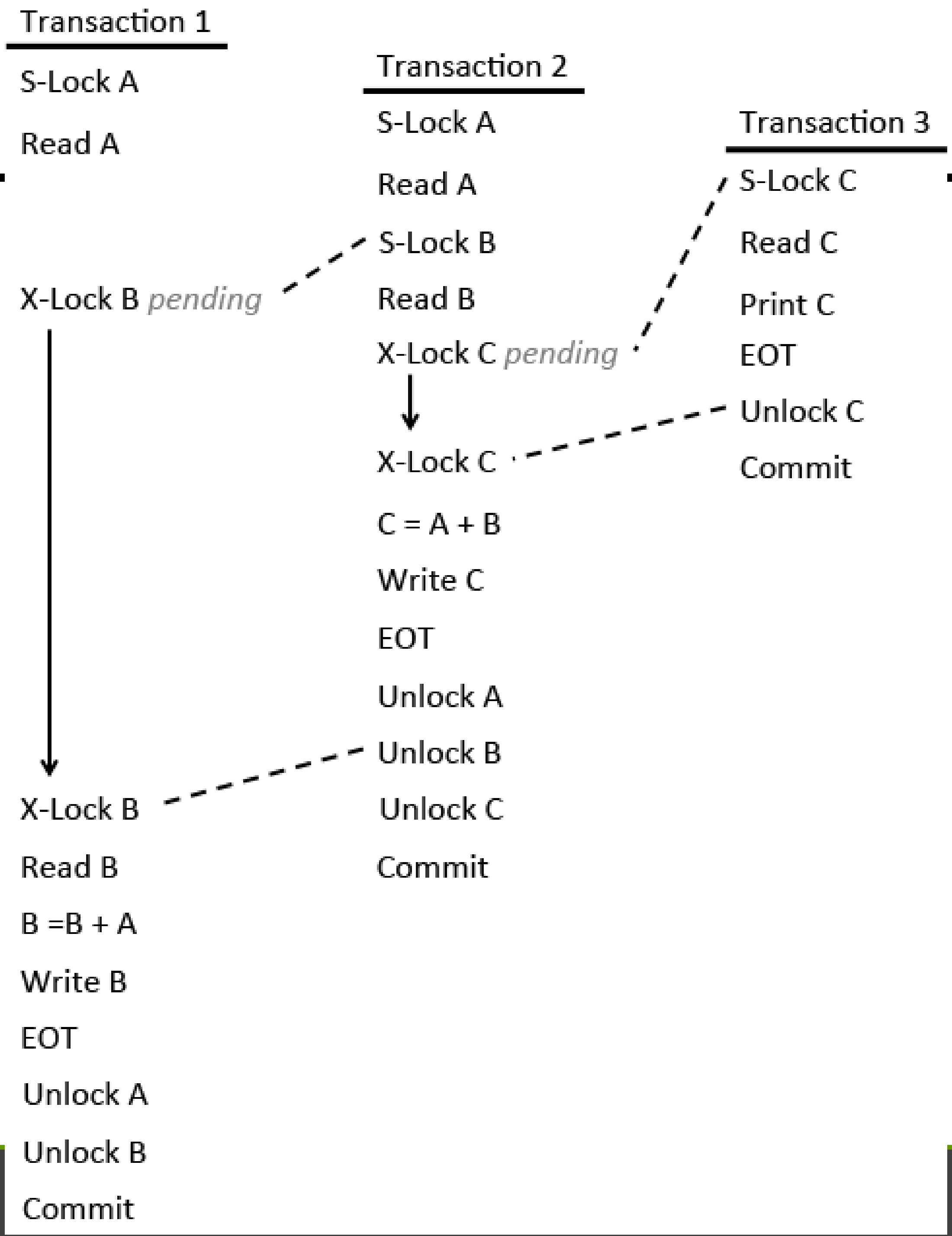
	T1	T2	
read-lock(X)	`Read(X)`		
	`X = X - 5`		
write-lock(X)	`Write(X)`		
		`Read(X)`	Waits till T1 releases write-lock(X)
		`X = X + 5`	
		`Write(X)`	
Locks released	`ROLLBACK`		
		`COMMIT`	

Inconsistent analysis cannot happen with 2PL

	T1	T2	
read-lock(X)	`Read(X)`		
	`X = X - 5`		
write-lock(X)	`Write(X)`		
		`Read(X)`	Waits till T1 releases write-locks on X and Y
		`Read(Y)`	
		`Sum = X+Y`	
read-lock(Y)	`Read(Y)`		
	`Y = Y + 5`		
write-lock(Y)	`Write(Y)`		

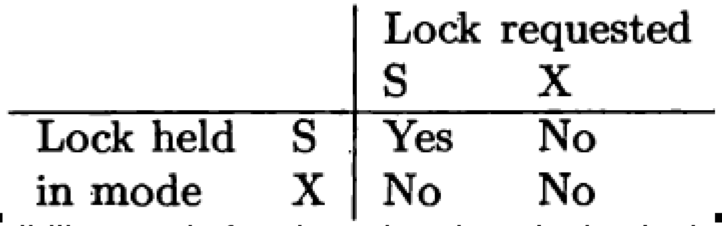
Concurrency Control by using 2PL

- **Another demonstration for 2PL**



2PL is Good Stuff

- **2PL is widely used**
  - Can be adapted to variety of situations, such as distributed processing
- **Variations:**
  - Basic 2PL (as described)
  - **Conservative 2PL**: T locks all items before execution
  - **Strict 2PL**: T doesn't unlock any items until after COMMIT or ABORT.
- **But**
  - 2PL reduces concurrency ☹
    - ➢ Conservative and strict variants reduce it even more, because they hold locks longer.
  - **Basic and strict 2PL are both subject to deadlock ☹**



		Lock requested	
		S	X
Lock held in mode	S	Yes	No
	X	No	No

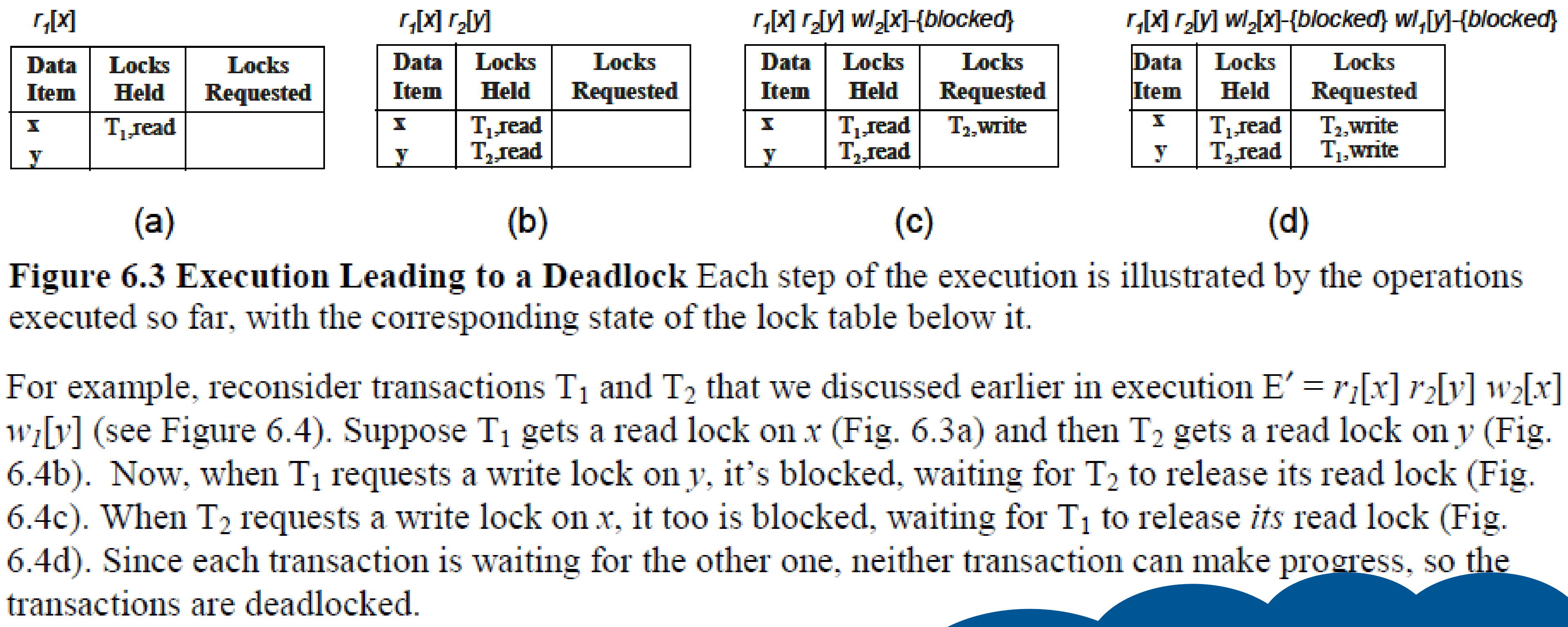
Compatibility matrix for shared and Exclusive locks

- **Increased overhead**
  - The type of lock held must be known before a lock can be granted
  - Three lock operations exist:
    - READ_LOCK to check the type of lock
    - WRITE_LOCK to issue the lock
    - UNLOCK to release the lock
  - A lock can be upgraded from share to exclusive and downgraded from exclusive to share

- **Two possible major problems may occur**
  - The resulting transaction schedule may not be serializable
  - The schedule may create **deadlocks**

Deadlock

https://courses.cs.washington.edu/courses/csep545/01wi/lectures/Chapt6V3.pdf



$r_1[x]$

Data Item	Locks Held	Locks Requested
x	T_1,read	
y		

(a)

$r_1[x]\ r_2[y]$

Data Item	Locks Held	Locks Requested
x	T_1,read	
y	T_2,read	

(b)

$r_1[x]\ r_2[y]\ wl_2[x]$-{blocked}

Data Item	Locks Held	Locks Requested
x	T_1,read	T_2,write
y	T_2,read	

(c)

$r_1[x]\ r_2[y]\ wl_2[x]$-{blocked} $wl_1[y]$-{blocked}

Data Item	Locks Held	Locks Requested
x	T_1,read	T_2,write
y	T_2,read	T_1,write

(d)

Figure 6.3 Execution Leading to a Deadlock Each step of the execution is illustrated by the operations executed so far, with the corresponding state of the lock table below it.

For example, reconsider transactions T_1 and T_2 that we discussed earlier in execution $E' = r_1[x]\ r_2[y]\ w_2[x]\ w_1[y]$ (see Figure 6.4). Suppose T_1 gets a read lock on x (Fig. 6.3a) and then T_2 gets a read lock on y (Fig. 6.4b). Now, when T_1 requests a write lock on y, it's blocked, waiting for T_2 to release its read lock (Fig. 6.4c). When T_2 requests a write lock on x, it too is blocked, waiting for T_1 to release *its* read lock (Fig. 6.4d). Since each transaction is waiting for the other one, neither transaction can make progress, so the transactions are deadlocked.

We need protocols/strategies to use Lock Table

- ***Deadlock* occurs when two transactions are each waiting for the other**
  - e.g., each waiting for a lock that the other holds
  - can also be deadlocks in larger circles of T's

- **2PL is not deadlock-free**

- **Two possible approaches:**
  - Deadlock *prevention* (don’t let it happen)
  - Deadlock *detection* (notice when it's happened and take recovery action)

https://courses.cs.washington.edu/courses/cse444/97au/slides/n-concurrency.ppt

Deadlock Prevention Using Timestamps

- ***Transaction timestamps*: unique numerical identifiers, same order as the starting order of the T's.**
  - Notation: TS(T_i)
  - Not necessarily a system clock value
  - Could be simply integer++ for each T

- **Interesting factoid: global timestamp techniques can be used in distributed systems, even when each system has its own clock.**

factoid 英 [ˈfæktɔɪd]
n. 仿真陈述（仅因出现在出版物上而被信以为真）
adj. 虚构的；似是而非的
- factoid questions 事实性问题
- factoid QA 事实型问答系统
- Factoid Question Answering 事实类问题回答

Two TS Schemes

- **Suppose T_j has locked X, and now T_i wants to lock X.**
  - *Wait-die*:

    if $TS(T_i)<TS(T_j)$ then T_i is allowed to wait

    else T_i dies and is restarted with its same TS.
  - *Wound-wait*:

    if $TS(T_i)<TS(T_j)$ then abort T_j and restart with its same TS

    else Ti is allowed to wait.
- **In both schemes, the older T has a certain preferential treatment.**
  - An aborted and restarted T always keeps it original TS.
  - Thus it gets older with respect to the other Ts, so eventually it gets the preference.

Deadlock Prevention Without Timestamps

- ***No waiting*: If T needs X and X is already locked, immediately abort T, restart later.**
    - May cause lots of restarting

- ***Cautious waiting*: Suppose T_i needs X, and X is already locked by T_j.**

  If T_j is waiting for anything, then abort T_j

  else abort T_i.

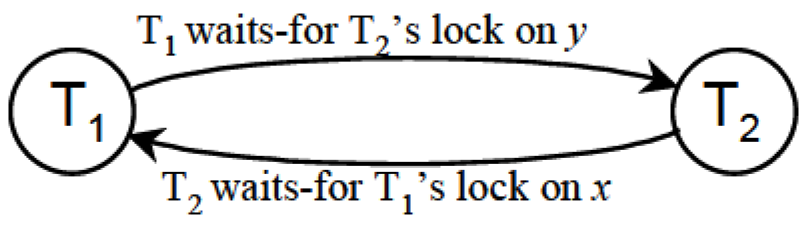
- ***Timeout*: If T waits longer than some fixed time, abort and restart T.**

Deadlock Detection

https://courses.cs.washington.edu/courses/csep545/01wi/lectures/Chapt6V3.pdf

- **There are two techniques that are commonly used to detect deadlocks: timeout-based detection and graph-based detection.**
  - Timeout-based detection guesses that a deadlock has occurred whenever a transaction has been blocked for too long.
    - It uses a timeout period that is much larger than most transactions' execution time (e.g., 15 seconds) and aborts any transaction that is blocked longer than this amount of time.
  - The alternative approach, called graph-based detection, explicitly tracks waiting situations and periodically checks them for deadlock.
    - This is done by building a waits-for graph, whose nodes model transactions and whose edges model waiting situations.

https://courses.cs.washington.edu/courses/csep545/01wi/lectures/Chapt6V3.pdf


T_1 waits-for T_2's lock on y
T_1
T_2
T_2 waits-for T_1's lock on x

$r_1[x]\ r_2[y]\ wl_2[x]\text{-}\{blocked\}\ wl_1[y]\text{-}\{blocked\}$

Figure 6.5 A Waits-For Graph The graph on the left represents the waiting situations in the execution on the right (see also Figure 6.4). Since there is a cycle involving T_1 and T_2, they are deadlocked.

- **Any newly added edge in the waits-for graph could cause a cycle. So it would seem that the data manager should check for cycles (deadlocks) whenever it adds an edge.**
- **While this is certainly correct, it is also possible to check for deadlocks less frequently, such as every few seconds.**
  - A deadlock won’t disappear **spontaneously （本能的, 自动的）**, so there is no harm in checking only periodically; the deadlock will still be there whenever the deadlock detector gets around to look for it.

Victim Selection

https://courses.cs.washington.edu/courses/csep545/01wi/lectures/Chapt6V3.pdf

- **After a deadlock has been detected using graph-based detection, one of the transactions in the cycle must be aborted. This is called the victim.**
- **Like all transactions in the deadlock cycle, the victim is blocked.**
  - It finds out it is the victim by receiving an error return code from the operation that was blocked, which says "you have been aborted."
  - It's now up to the application that issued the operation to decide what to do next.
- **Usually, it just restarts, possibly after a short artificial delay to give the other transactions in the cycle time to finish, so they don't all deadlock again.**

Concurrency Control by Timestamp Ordering

Main idea: insure that conflicting operations occur in timestamp order

- **Each item X must have a read TS and a write TS**
  - = TS's of the T which last read or wrote X
  - If T wants to use X, then X's TSs are checked. For a conflicting operation, TS(T) must be later. If not, T is aborted and restarted with a <u>new</u> (later) TS.
- **Properties**
  - Guarantees serializability
    - in fact, guarantees conflict serializability
  - Deadlock free
  - Not starvation free ☹
    - You *hope* that eventually T becomes "late enough" to slide through, but that might never happen.

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.**

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.

Chapter 4: Transaction Control/Management

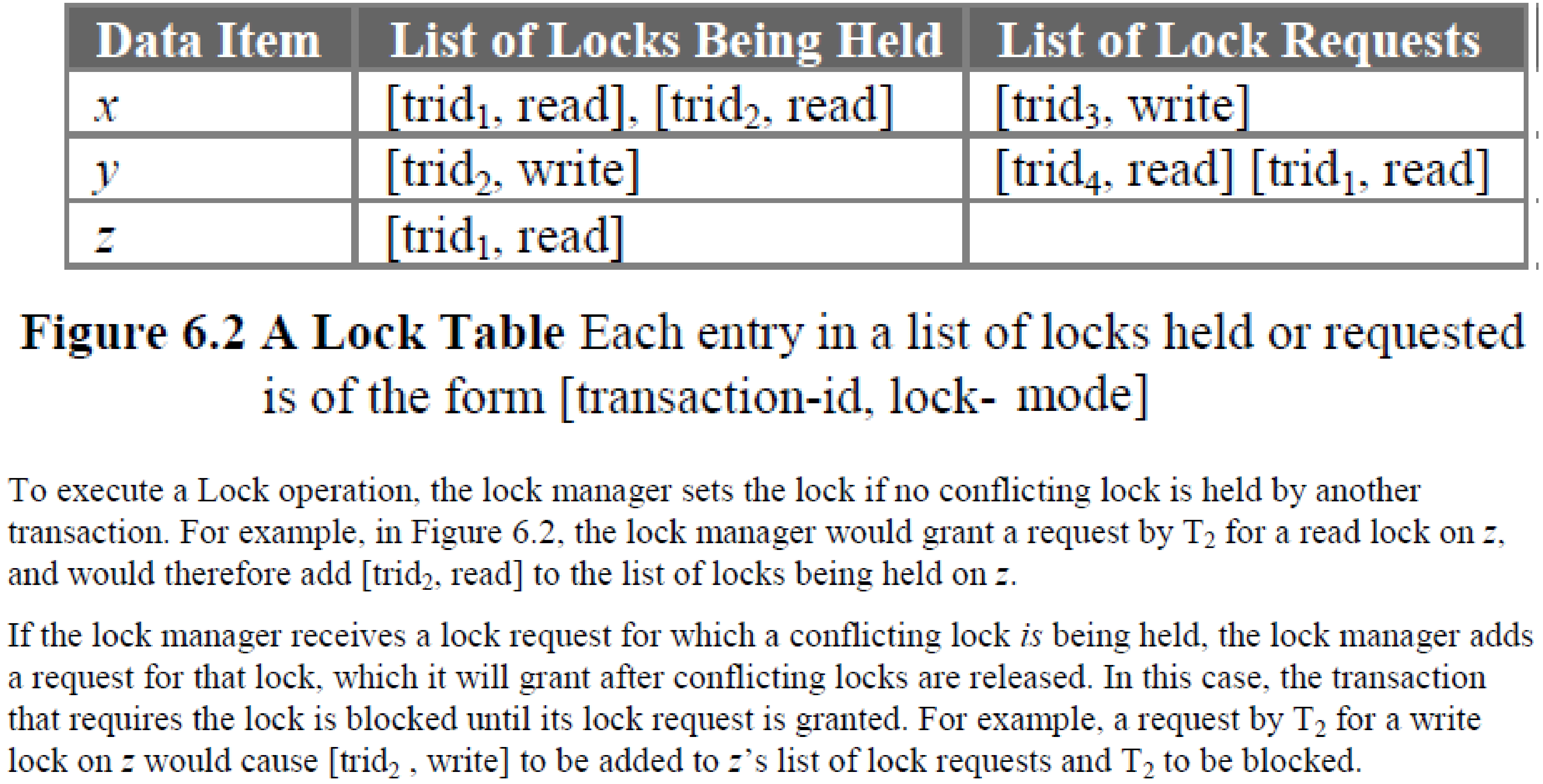
- **My understanding about**
  - Transactions and Scheduler
    - Serializable schedule
  - Concurrency control
    - How to ensure the serializability?
      - ✓ Lock …
- **4th project**
- **Advanced project in reading PostgreSQL**
  - Transaction Management in PostgreSQL

4th project

- **In Project 2, the server also maintains a lock table, which is used to simulate 2PL.**
  - Client:
    - Random loopnum
    - While (loopnum)
      - ✓ Access (**Read** or **Write**) a random character from A to Z
      - ✓ Sleep for a random time
      - ✓ loopnum--
  - Server with different strategy [Shared L, 2PL]
    - Maintain a lock table to control the concurrent access
      - ✓ 2PL, and Deadlock detection (Wait-for graph)
- **Requirement**
  - How to simulate? How to check if Deadlock or not?

- **Distill the workflow of SQL processing with an example**
  - Major classes/functions
  - + Data Structure
  - + Relationships of function calls
  - + your understanding about the mechanism

https://courses.cs.washington.edu/courses/csep545/01wi/lectures/Chapt6V3.pdf



Data Item	List of Locks Being Held	List of Lock Requests
x	[trid$_1$, read], [trid$_2$, read]	[trid$_3$, write]
y	[trid$_2$, write]	[trid$_4$, read] [trid$_1$, read]
z	[trid$_1$, read]	

Figure 6.2 A Lock Table Each entry in a list of locks held or requested is of the form [transaction-id, lock- mode]

To execute a Lock operation, the lock manager sets the lock if no conflicting lock is held by another transaction. For example, in Figure 6.2, the lock manager would grant a request by T$_2$ for a read lock on z, and would therefore add [trid$_2$, read] to the list of locks being held on z.

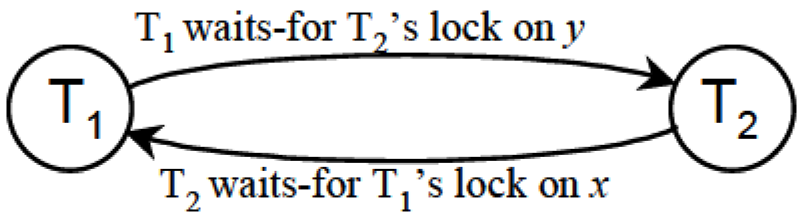
If the lock manager receives a lock request for which a conflicting lock *is* being held, the lock manager adds a request for that lock, which it will grant after conflicting locks are released. In this case, the transaction that requires the lock is blocked until its lock request is granted. For example, a request by T$_2$ for a write lock on z would cause [trid$_2$, write] to be added to z's list of lock requests and T$_2$ to be blocked.

Deadlock Detection

https://courses.cs.washington.edu/courses/csep545/01wi/lectures/Chapt6V3.pdf

- **There are two techniques that are commonly used to detect deadlocks: timeout-based detection and graph-based detection.**
  - Timeout-based detection guesses that a deadlock has occurred whenever a transaction has been blocked for too long.
    - It uses a timeout period that is much larger than most transactions' execution time (e.g., 15 seconds) and aborts any transaction that is blocked longer than this amount of time.
  - The alternative approach, called graph-based detection, explicitly tracks waiting situations and periodically checks them for deadlock.
    - This is done by building a waits-for graph, whose nodes model transactions and whose edges model waiting situations.

https://courses.cs.washington.edu/courses/csep545/01wi/lectures/Chapt6V3.pdf



$r_1[x]\ r_2[y]\ wl_2[x]$-{*blocked*} $wl_1[y]$-{*blocked*}

Figure 6.5 A Waits-For Graph The graph on the left represents the waiting situations in the execution on the right (see also Figure 6.4). Since there is a cycle involving T_1 and T_2, they are deadlocked.

- **Any newly added edge in the waits-for graph could cause a cycle. So it would seem that the data manager should check for cycles (deadlocks) whenever it adds an edge.**
- **While this is certainly correct, it is also possible to check for deadlocks less frequently, such as every few seconds.**
  - A deadlock won't disappear **spontaneously （本能的, 自动的）**, so there is no harm in checking only periodically; the deadlock will still be there whenever the deadlock detector gets around to look for it.

“On Optimistic Methods for Concurrency Control”

H. T. Kung, John T. Robinson 1981

•

•

SigOps HoF citation (2015):

This paper introduced the notion of optimistic concurrency control, proceeding with a transaction without locking the data items accessed, in the expectation that the transaction’s accesses will not conflict with those of other transactions. This idea, originally introduced in the context of conventional databases, has proven very powerful when transactions are applied to general-purpose systems.

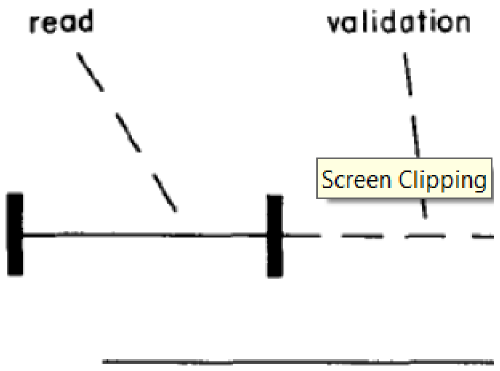
– “An interesting problem is one where it is not known in advance how (or even if) the problem can be solved…I love working on interesting problems.”

What's Wrong with Locks?

- Locks are overhead vs. sequential case
  - Even for read-only transactions; Deadlock detection
- No general-purpose deadlock-free locking protocols that always provide high concurrency
- Paging leads to long lock hold times
- Locks cannot be released until end of transaction (to allow for transaction abort)
- Locking may be necessary only in the worst case

- Priority inversion
- Lock-based programs do not compose: correct fragments may fail when combined

- Three Phases of



During read phase: Any wr

Write phase occurs if

Re

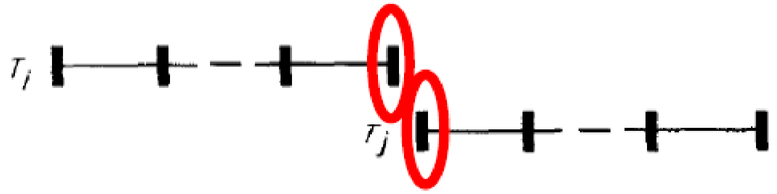
create	create a
delete(*n*)	delete o
read(*n*, *i*)	read ite
write (*n*, *i*, *v*)	write *v*
copy(*n*)	create a r *n* and ret
exchange(*n*1, *n*2)	exchange

```
tcreate = (
    n := create;
    create set :=
    return n)

twrite(n, i, v) =
    if n ∈ create
        then wr
    else if n ∈ w
        then wr
    else (
        m := cop
        copies[n
        write set
        write(copies[n], i, v)))
```

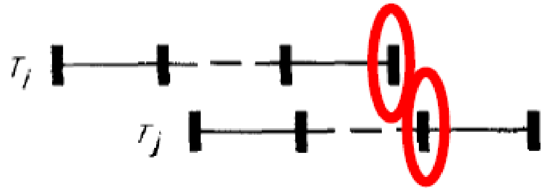
Validation Phase

- Assign transaction number at the end of the re
  - Optimization: at end of a successful write pha
- Serial equivalence: T_i before T_j

1)

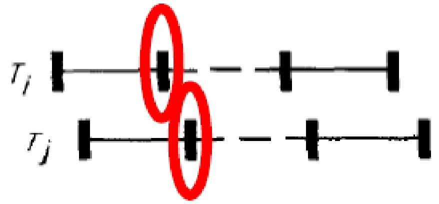


2)



$+ WriteSet_i$ doesn't

3)



$+ WriteSet_i$ doesn't in

"Concurrency Control and Recovery"

Michael J. Franklin 1997

- Mike Franklin (U. Maryland, UC Berkeley, U. Chicago)
  - ACM Fellow
  - Sigmod Test-of-Time award 2004 & 2013

Concurrency Control

- **Two-phase Locking: Each transaction acquires all its locks before releasing any**
  - Guarantees serializability
  - OCC not used because consumes more resources than locking

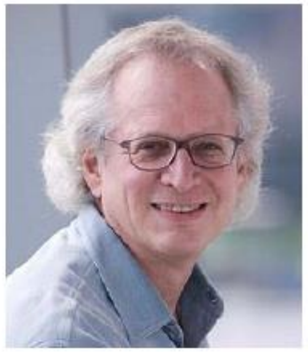
- **Deadlock avoidance/detection**
  - Avoidance: Impose order on locks
  - Detection: Timeouts or cycles in waits-for graphs

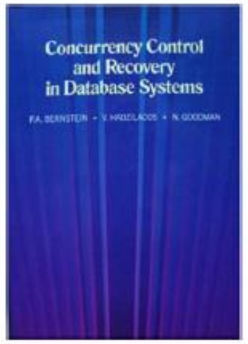
- **Isolation levels: Write locks held to commit/abort (strict locking)**
  - READ UNCOMMITTED: no read locks (can read roll-backed updates)
  - READ COMMITTED: short-duration read locks (non-repeatable reads)
  - REPEATABLE READ: strict read locks (phantom tuples)
  - SERIALIZABLE: strict read locks on predicates (harder to achieve)

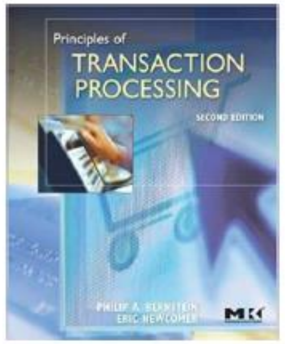
“Multiversion Concurrency Control: Theory and Algorithms”

Philip Bernstein, Nathan Goodman 1983

- **Phil Bernstein (Harvard, Microsoft Research)**
  - NAE, ACM Fellow
  - SIGMOD Edgar F. Codd Innovations Award




Concurrency Control and Recovery in Database Systems
P.A. BERNSTEIN • V. HADZILACOS • N. GOODMAN



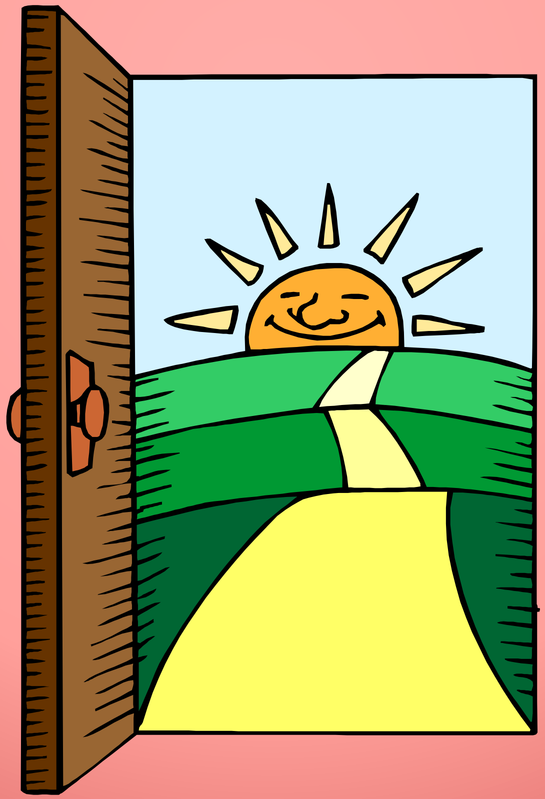
- **Nathan Goodman (Harvard, bioinformatics industry)**
  - Founded non-profit organization providing education & support for people with Huntington’s disease

Multiversion Concurrency Control

- **Theory for analyzing the correctness of MVCC algorithms**
  - Need to map reads/writes to versioned reads/writes
  - Transactions ordered by “write before its reads”
  - One-copy serializability (1-SR): equivalent to serial schedule on single-version DB

- **Proves correctness of 3 MVCC algorithms**
  - Timestamp-based [Reed78]
  - Locking-based [generalization of Bayer80, Stearns81]
  - Mixed Method using Lamport clocks for consistent timestamps [generalization of Dubourdieu82]

Appendix A: Deadlocks



Deadlocks

- **A deadlock is an impasse that may result when two or more transactions are waiting for locks to be released which are held by each other.**
  - For example: T1 has a lock on X and is waiting for a lock on Y, and T2 has a lock on Y and is waiting for a lock on X.

- **Given a schedule, we can detect deadlocks which will happen in this schedule using a *wait-for graph* (WFG).**

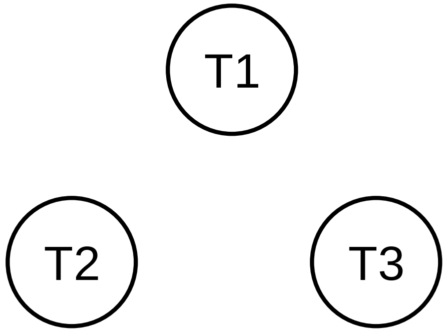
Precedence/Wait-For Graphs

- **Precedence graph**
  - Each transaction is a vertex
  - Arcs from T1 to T2 if
    - T1 reads X before T2 writes X
    - T1 writes X before T2 reads X
    - T1 writes X before T2 writes X

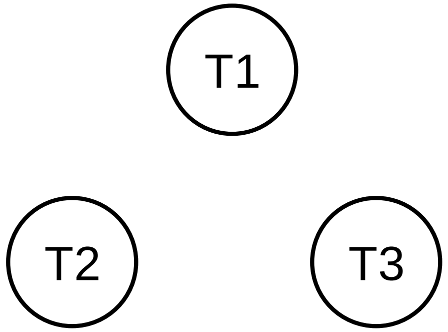
- **Wait-for Graph**
  - Each transaction is a vertex
  - Arcs from T2 to T1 if
    - T1 read-locks X then T2 tries to write-lock it
    - T1 write-locks X then T2 tries to read-lock it
    - T1 write-locks X then T2 tries to write-lock it

Example

T1 Read(X)
T2 Read(Y)
T1 Write(X)
T2 Read(X)
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)

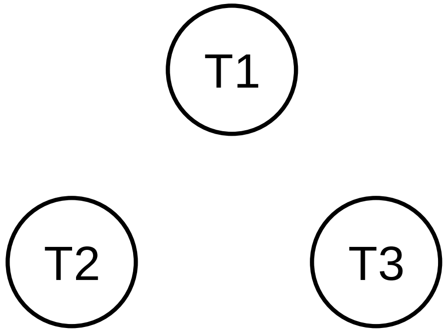


Wait for graph

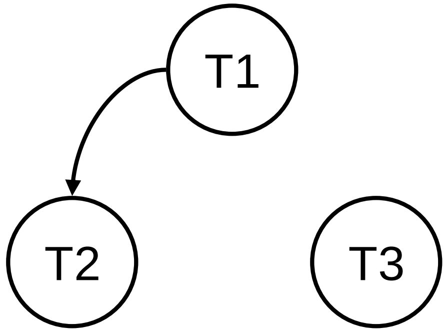


Precedence graph

T1 Read(X)
T2 Read(Y)
T1 Write(X)
T2 Read(X)
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)

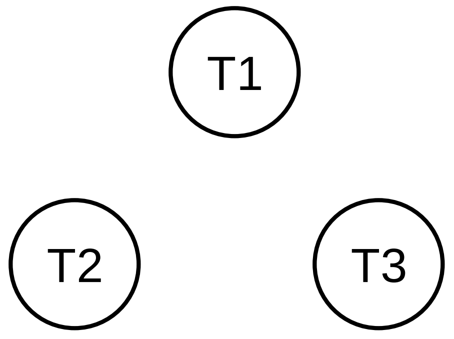


Wait for graph

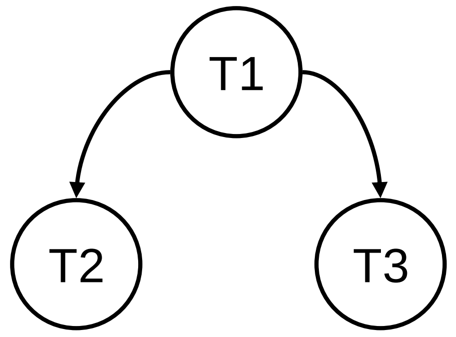


Precedence graph

T1 Read(X)
T2 Read(Y)
T1 Write(X)
T2 Read(X)
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)



Wait for graph



Precedence graph

T1 Read(X)

T2 Read(Y)

T1 Write(X)

T2 Read(X)

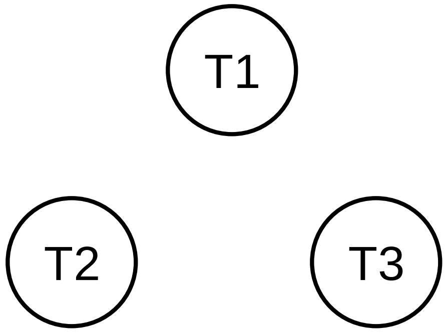
T3 Read(Z)

T3 Write(Z)

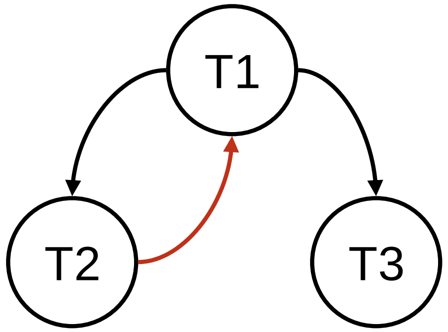
T1 Read(Y)

T3 Read(X)

T1 Write(Y)



Wait for graph



Precedence graph

T1 Read(X) read-locks(X)

T2 Read(Y) read-locks(Y)

T1 Write(X) **write-lock(X)**

T2 Read(X) **tries read-lock(X)**

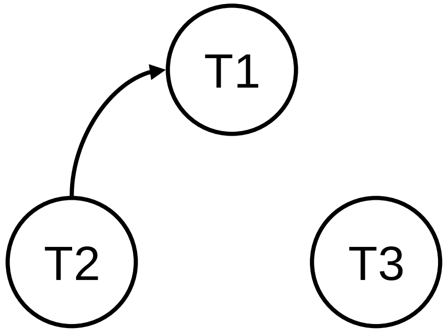
T3 Read(Z)

T3 Write(Z)

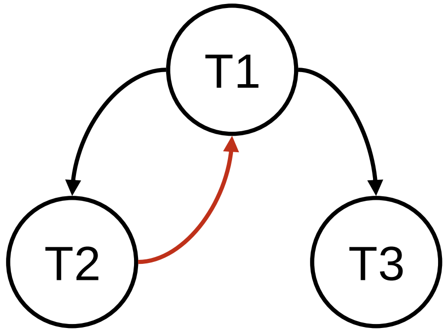
T1 Read(Y)

T3 Read(X)

T1 Write(Y)



Wait for graph



Precedence graph

T1 Read(X) read-locks(X)

T2 Read(Y) read-locks(Y)

T1 Write(X) **write-lock(X)**

T2 Read(X) tries read-lock(X)

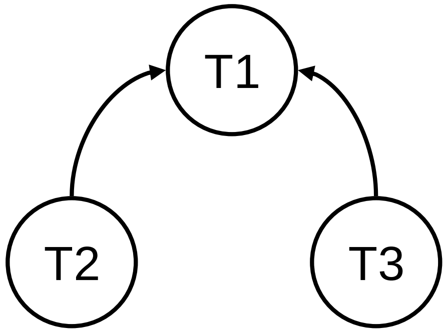
T3 Read(Z) read-lock(Z)

T3 Write(Z) write-lock(Z)

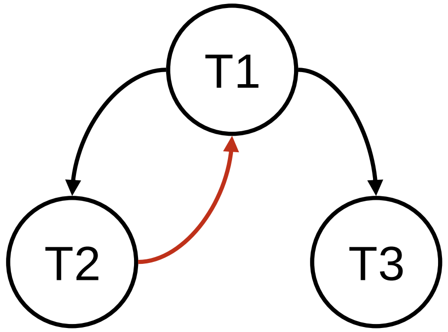
T1 Read(Y) read-lock(Y)

T3 Read(X) **tries read-lock(X)**

T1 Write(Y)



Wait for graph



Precedence graph

T1 Read(X) read-locks(X)

T2 Read(Y) **read-locks(Y)**

T1 Write(X) write-lock(X)

T2 Read(X) tries read-lock(X)

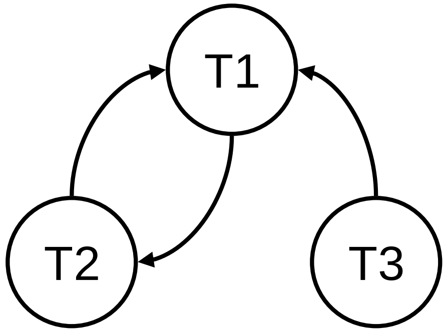
T3 Read(Z) read-lock(Z)

T3 Write(Z) write-lock(Z)

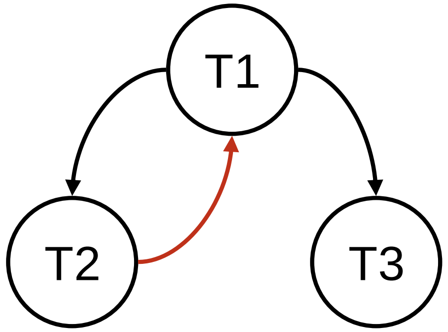
T1 Read(Y) read-lock(Y)

T3 Read(X) tries read-lock(X)

T1 Write(Y) **tries write-lock(Y)**



Wait for graph



Precedence graph

Deadlock Prevention

Familiar here? – OS's prevention strategy for deadlock!

- **Deadlocks can arise with 2PL**
  - Deadlock is less of a problem than an inconsistent DB
  - We can detect and recover from deadlock
  - It would be nice to avoid it altogether

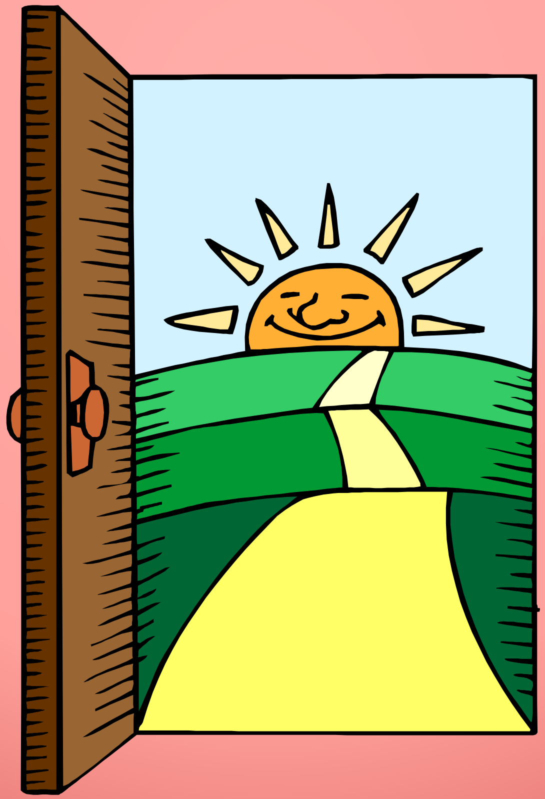
1. **Conservative 2PL**
   - **All locks must be acquired before the transaction starts**
   - Hard to predict what locks are needed
   - Low 'lock utilisation' - transactions can hold on to locks for a long time, but not use them much
2. **We impose an ordering on the resources [ORA – Ordered Resource Allocation]**
   - Transactions must acquire locks in this order
   - Transactions can be ordered on the last resource they locked
   - This prevents deadlock
     - If T1 is waiting for a resource from T2 then that resource must come after all of T1's current locks
     - All the arcs in the wait-for graph point 'forwards' - no cycles

Example of resource ordering

- Suppose resource order is: X < Y
- This means, if you need locks on X and Y, you first acquire a lock on X and only after that a lock on Y
  - (even if you want to write to Y before doing anything to X)

- It is impossible to end up in a situation when T1 is waiting for a lock on X held by T2, and T2 is waiting for a lock on Y held by T1.

Appendix B: Concurrency Control With No Locking



Optimistic Concurrency Control

- **Locking protocols are *pessimistic* as they aim to abort or block conflicts**
  - This requires overhead when there is little contention
  - In optimistic concurrency control the premise is that conflicts are rare
- **We will look at two versions of optimistic CC**
  - ***Timestamps*** – maintain timestamps of transactions and reads and writes of database objects
  - ***Validation*** – similar to the timestamp system except that data is recorded about the actions of transactions
    - Rather than data about database objects

Timestamping

- Each transaction has a timestamp, TS, and if T1 starts before T2 then TS(T1) < TS(T2)
  - use the system clock or an incrementing counter to generate timestamps
- Each resource has two timestamps
  - R(X), the largest timestamp of any transaction that has read X
  - W(X), the largest timestamp of any transaction that has written X

Timestamp Protocol – W/R only valid when "≥"

- If T tries to **read** X
  - If TS(T) < W(X), T is rolled back and restarted with a later timestamp
  - If TS(T) ≥ W(X), then the read succeeds and we set R(X) to be max(R(X), TS(T))
- T tries to **write** X
  - If TS(T) < W(X) or TS(T) < R(X) then T is rolled back and restarted with a later timestamp
  - Otherwise the write succeeds and we set W(X) to TS(T)

Timestamps

- **Each transaction is given a timestamp**
  - Issued in ascending order when the transaction begins
- **Timestamps can be generated**
  - By using the system clock
  - By maintaining a counter within the scheduler
- **The scheduler maintains a table of active transactions and their timestamps**

Timestamp Example

T1	T2
Read(X)	Read(X)
Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R			
W			

	T1	T2
TS		

	T1	T2
→	Read(X)	Read(X)
	Read(Y)	Read(Y)
	Y = Y + X	Z = Y - X
	Write(Y)	Write(Z)

	X	Y	Z
R	1		
W			

	T1	T2
TS	1	

T1	T2
→ Read(X)	→ Read(X)
Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R	2		
W			

	T1	T2
TS	1	2

T1	T2
Read(X)	→ Read(X)
→ Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R	2	1	
W			

	T1	T2
TS	1	2

T1	T2
Read(X)	Read(X)
→ Read(Y)	→ Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R	2	2	
W			

	T1	T2
TS	1	2

T1	T2
Read(X)	Read(X)
Read(Y)	→ Read(Y)
→ Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R	2	2	
W			

	T1	T2
TS	1	2

T1	T2
Read(X)	Read(X)
Read(Y)	Read(Y)
→ Y = Y + X	→ Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R	2	2	
W			

	T1	T2
TS	1	2

T1	T2
Read(X)	Read(X)
Read(Y)	Read(Y)
Y = Y + X	→ Z = Y - X
→ Write(Y)	Write(Z)

	X	Y	Z
R	2	2	
W			

	T1	T2
TS	1	2

T1	T2
Read(X)	Read(X)
Read(Y)	Read(Y)
Y = Y + X	→ Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R	2	2	
W			

	T1	T2
TS	3	2

→ T1	T2
Read(X)	Read(X)
Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	→ Write(Z)

	X	Y	Z
R	2	2	
W			2

	T1	T2
TS	3	2

T1	T2
→ Read(X)	Read(X)
Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R	3	2	
W			2

	T1	T2
TS	3	2

T1	T2
Read(X)	Read(X)
→ Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R	3	3	
W			2

	T1	T2
TS	3	2

T1	T2
Read(X)	Read(X)
Read(Y)	Read(Y)
→ Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R	3	3	
W			2

	T1	T2
TS	3	2

T1	T2
Read(X)	Read(X)
Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
→ Write(Y)	Write(Z)

	X	Y	Z
R	3	3	
W		3	2

	T1	T2
TS	3	2

Timestamps Versus Locks

- **Using timestamps is superior where most transactions are read-only**
  - Or when it is rare for concurrent transactions to read and write the same element
- **Locking performs better when there are many conflicts**
  - Locking delays transactions
  - But rollbacks will be more frequent, leading to even more delay

- **Timestamping**

- **The protocol means that transactions with higher times take precedence**
  - Equivalent to running transactions in order of their final time values
  - Transactions don’t wait - no deadlock

- **Problems**
  - Long transactions might keep getting restarted by new transactions - starvation
  - Rolls back old transactions, which may have done a lot of work

MVCC – Multi-version concurrency control

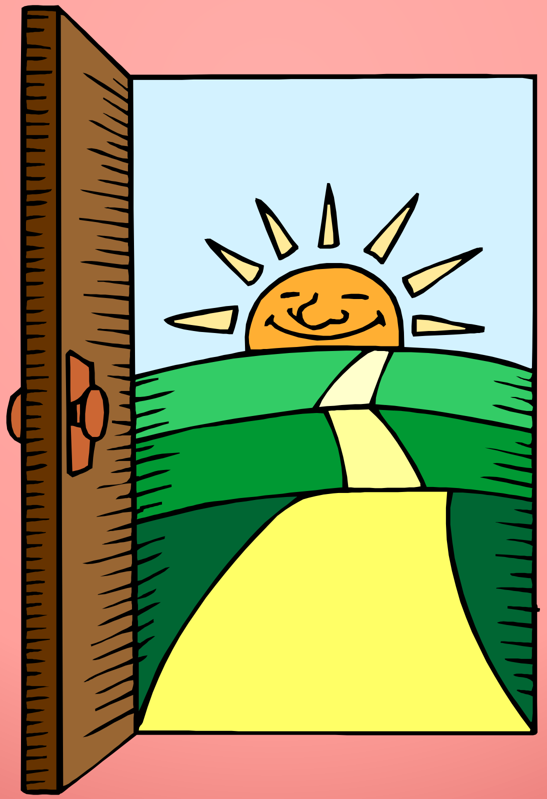
- For example, the **T1 transaction** changes the **A record** from *1* to *2* and then changes the **B record**, the **T2 transaction** can simultaneously change the **A record**, too. Let's assume that the **T2 transaction** changes the **A record** from *2* to *4* by adding **+2**.
  - If two transactions are successfully terminated, there is no issue. But it is important that all transactions can be rolled back.
    - If the T1 transaction is rolled back, the value of the A record should be returned to *1*, i.e. the value *before* the T1 transaction was executed. This is to satisfy the ACID property of the database.
    - However, the T2 transaction has already changed the A record value to *3*. So, it is impossible to return the A record to *1* regardless of the situation.

- MVCC allows each of them, **T1** and **T2 transactions**, to have their own changed versions.
  - Even when the T1 transaction has changed the A record from *1* to *2*, the T1 transaction leaves the original value *1* as it is and writes that the *T1* ***transaction version*** of the A record is 2.
  - Then, the following T2 transaction changes the A record from *1* to *3*, not from *2* to *4*, and writes that the *T2* ***transaction version*** of the A record is 3.

  - When the T1 transaction is rolled back, it does not matter if the 2, *the T1 transaction version*, is not applied to the A record. After that, if the T2 transaction is committed, the 3, *the T2 transaction version*, will be applied to the A record.
  - If the T1 transaction is committed prior to the T2 transaction, the A record is changed to 2, and then to 3 at the time of committing the T2 transaction. The final database status is identical to the status of executing each transaction independently, without any impact on other transactions.
- Therefore, it satisfies the ACID property. This method is called **Multi-version concurrency control (MVCC)**.

Appendix C:

Database Recovery Management



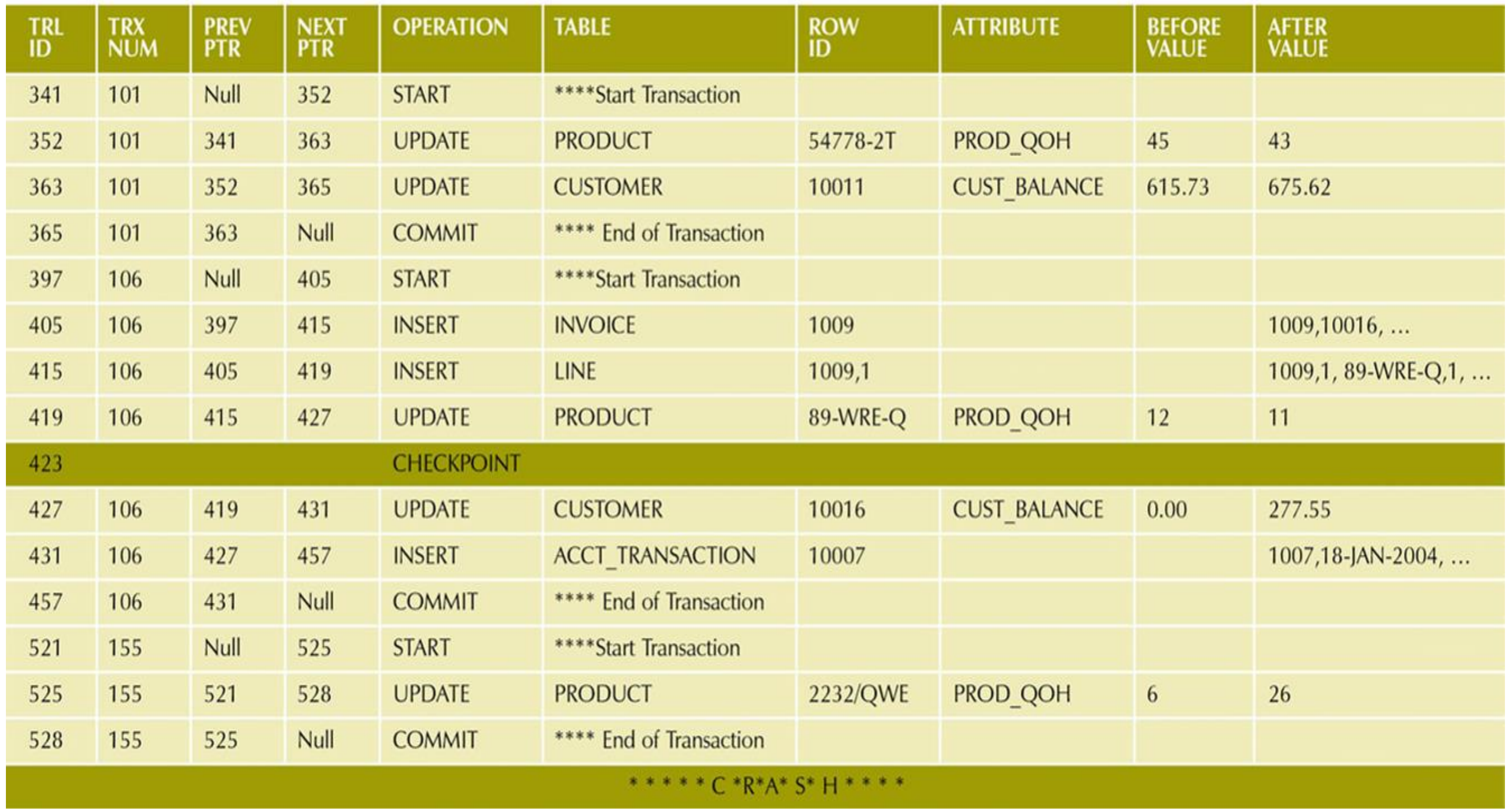
The Transaction Log

- **The transaction log records the details of all transactions**
  - Any changes the transaction makes to the database
  - How to undo these changes
  - When transactions complete and how
- **The log is stored on disk, not in memory**
  - If the system crashes it is preserved
- **Write ahead log rule**
  - The entry in the log must be made before **COMMIT** processing can complete

http://www.cs.nott.ac.uk/~psznza/G51DBS08/lecture14.ppt

A Transaction Log for Transaction Recovery Examples

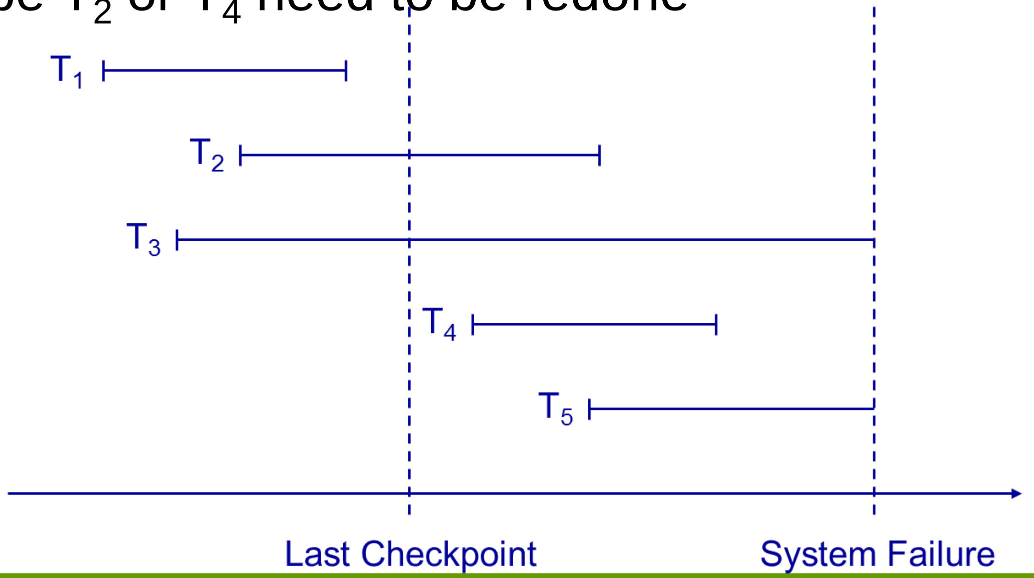
TABLE 9.13 A TRANSACTION LOG FOR TRANSACTION RECOVERY EXAMPLES



TRL ID	TRX NUM	PREV PTR	NEXT PTR	OPERATION	TABLE	ROW ID	ATTRIBUTE	BEFORE VALUE	AFTER VALUE
341	101	Null	352	START	****Start Transaction				
352	101	341	363	UPDATE	PRODUCT	54778-2T	PROD_QOH	45	43
363	101	352	365	UPDATE	CUSTOMER	10011	CUST_BALANCE	615.73	675.62
365	101	363	Null	COMMIT	**** End of Transaction				
397	106	Null	405	START	****Start Transaction				
405	106	397	415	INSERT	INVOICE	1009			1009,10016, …
415	106	405	419	INSERT	LINE	1009,1			1009,1, 89-WRE-Q,1, …
419	106	415	427	UPDATE	PRODUCT	89-WRE-Q	PROD_QOH	12	11
423				CHECKPOINT					
427	106	419	431	UPDATE	CUSTOMER	10016	CUST_BALANCE	0.00	277.55
431	106	427	457	INSERT	ACCT_TRANSACTION	10007			1007,18-JAN-2004, …
457	106	431	Null	COMMIT	**** End of Transaction				
521	155	Null	525	START	****Start Transaction				
525	155	521	528	UPDATE	PRODUCT	2232/QWE	PROD_QOH	6	26
528	155	525	Null	COMMIT	**** End of Transaction				
					* * * * * C *R*A* S* H * * * *				

System Recovery

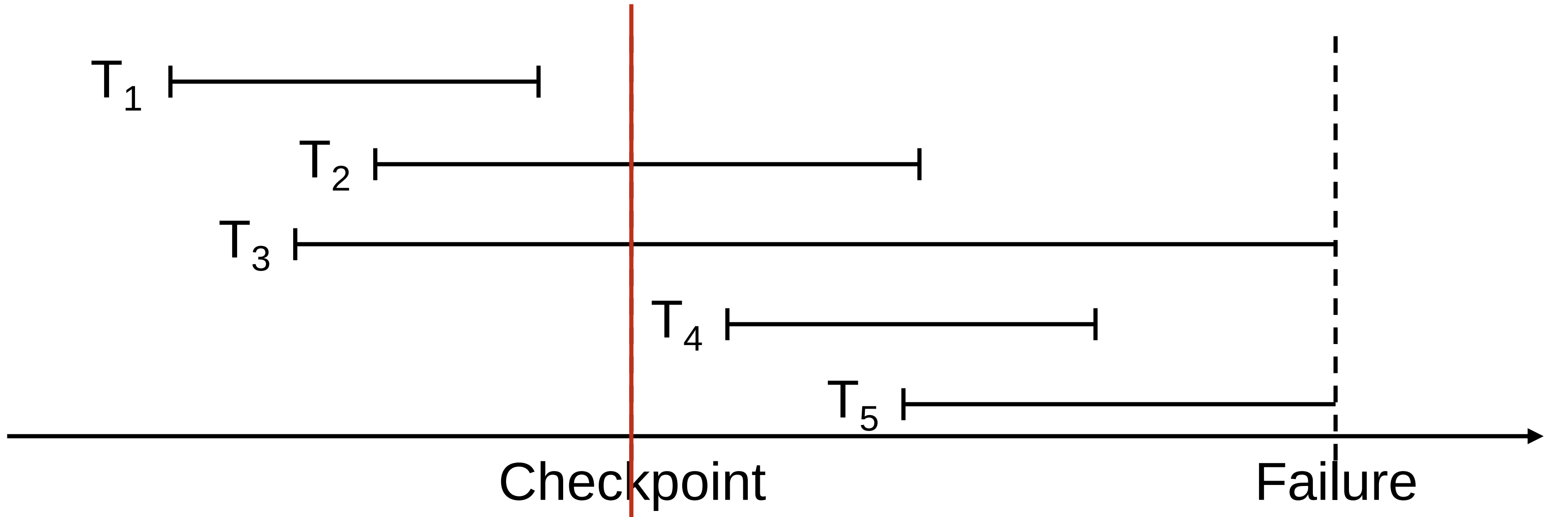
- **Any transaction that was running at the time of failure needs to be undone and restarted**
- **Any transactions that committed since the last checkpoint need to be redone**
  - Transactions of type T_1 need no recovery
  - Transactions of type T_3 or T_5 need to be undone and restarted
  - Transactions of type T_2 or T_4 need to be redone



Transaction Recovery

- **UNDO and REDO: lists of transactions**
  - UNDO = all transactions running at the last checkpoint
  - REDO = empty

- **For each entry in the log, starting at the last checkpoint**
  - If a BEGIN TRANSACTION entry is found for T
    - Add T to UNDO
  - If a COMMIT entry is found for T
    - Move T from UNDO to REDO

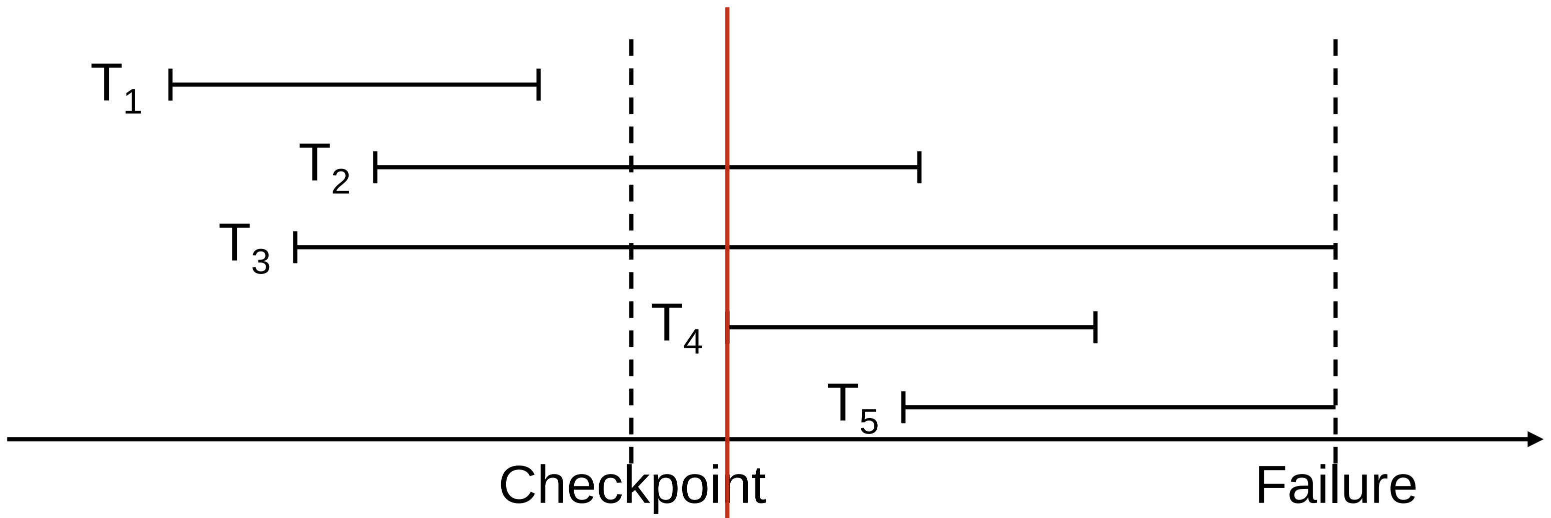


UNDO: T_2, T_3

REDO:

Last Checkpoint

Active transactions: T_2, T_3

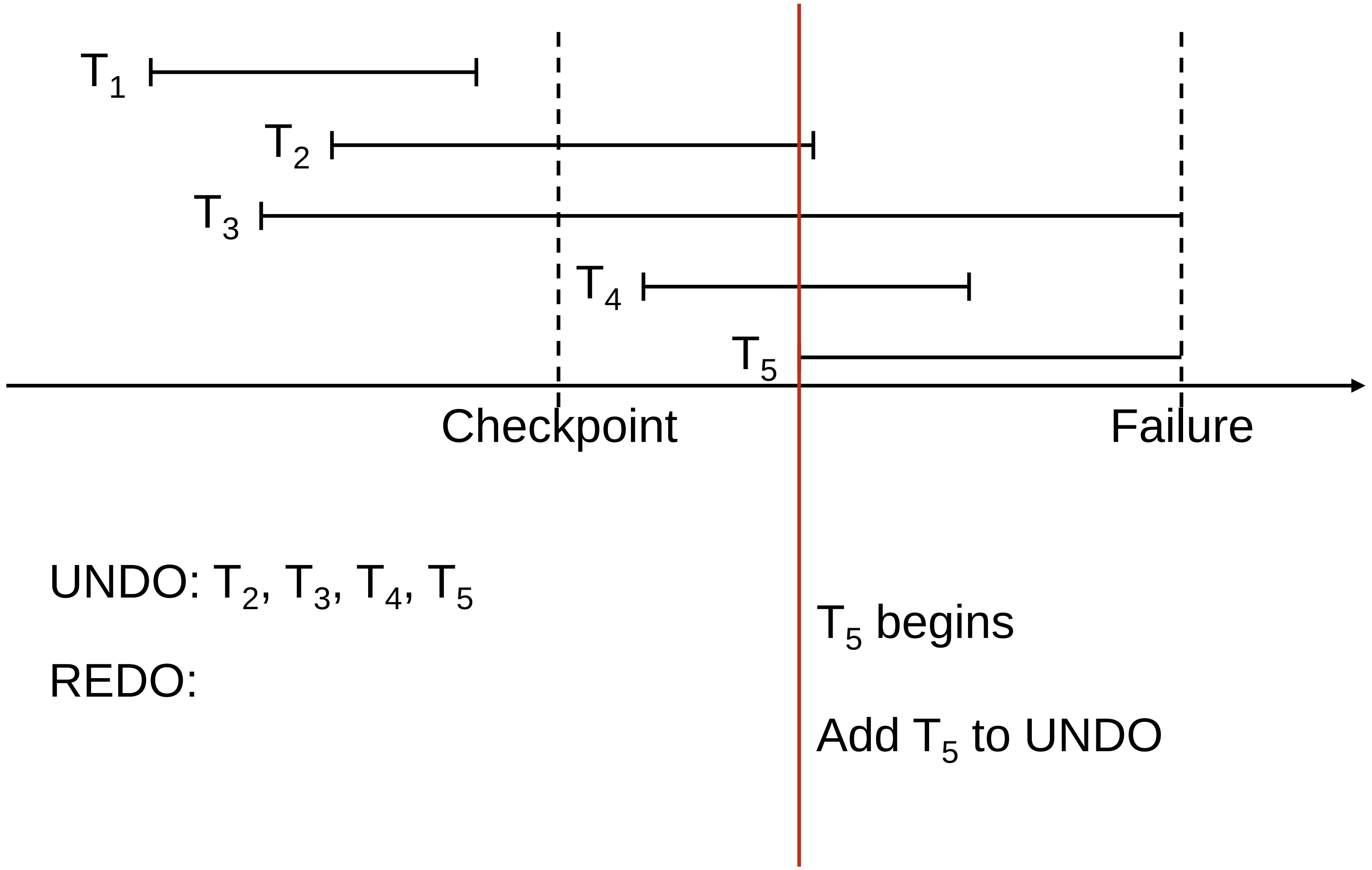


UNDO: T_2, T_3, T_4

REDO:

T4 Begins

Add T_4 to UNDO

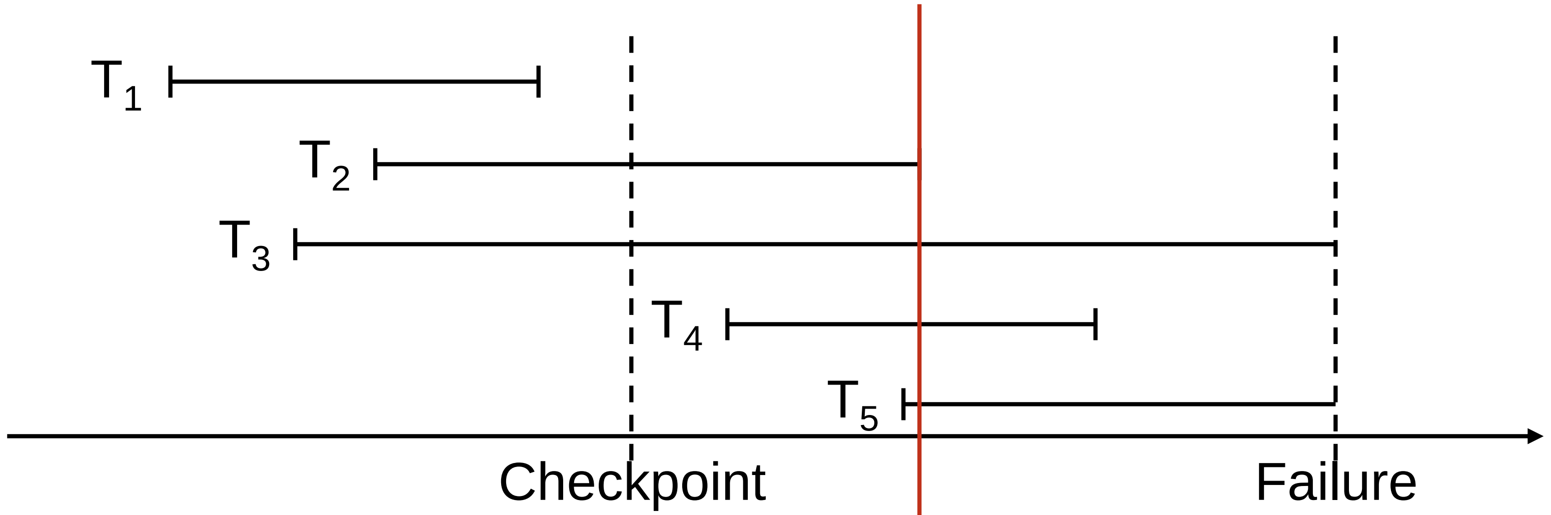


UNDO: T_2, T_3, T_4, T_5

REDO:

T_5 begins

Add T_5 to UNDO

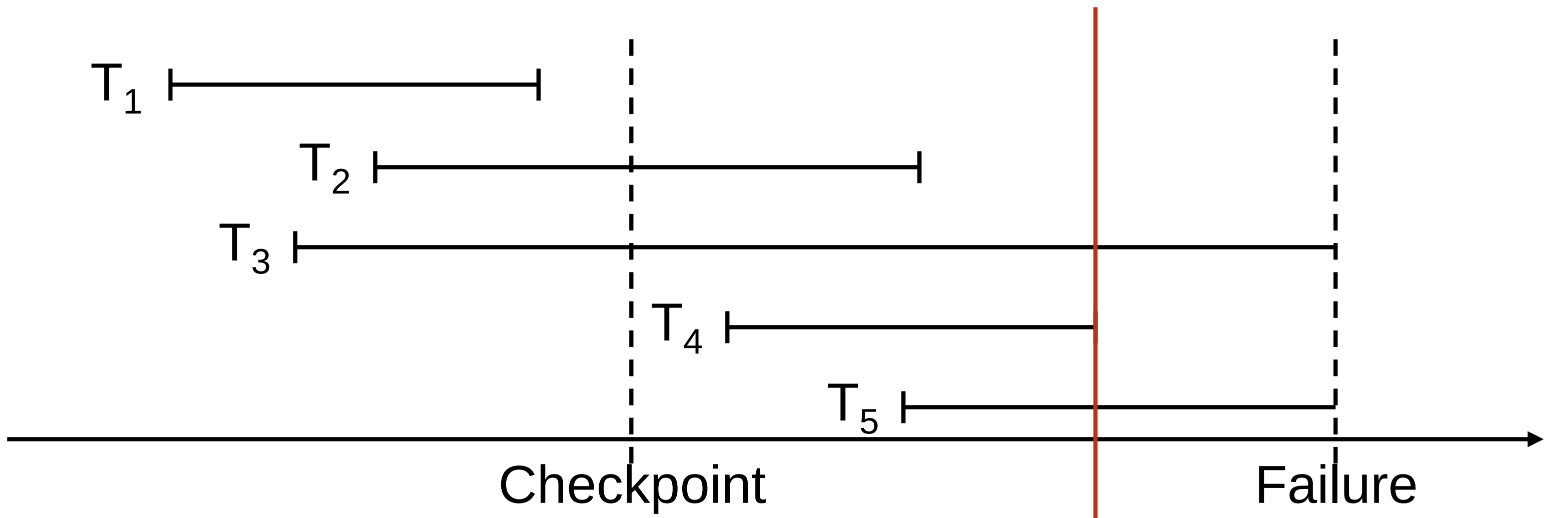


UNDO: T_3, T_4, T_5

REDO: T_2

T_2 Commits

Move T_2 to REDO



UNDO: T_3, T_5

REDO: T_2, T_4

T_4 Commits

Move T_4 to REDO

Forwards and Backwards

- **Backwards recovery**
  - We need to undo some transactions
  - Working backwards through the log we undo any operation by a transaction on the UNDO list
  - This returns the database to a consistent state

- **Forwards recovery**
  - Some transactions need to be redone
  - Working forwards through the log we redo any operation by a transaction on the REDO list
  - This brings the database up to date

Media Failures & Backups

- **Media Failures**
  - System failures are not too severe. Only information since the last checkpoint is affected. This can be recovered from the transaction log
  - Media failures (disk crashes etc.) are more serious
    - The data stored to disk is damaged
    - The transaction log itself may be damaged
- **Backups**
  - Backups are needed to recover from media failure
    - The transaction log and entire contents of the database is written to secondary storage (often tape)
    - Time consuming, and often requires down time
  - Backups frequency
    - Frequent enough that little information is lost
    - Not so frequent as to cause problems
    - Every day (night) is common

Recovery from Media Failure

- Restore the database from the last backup
- Use the transaction log to redo any changes made since the last backup
- If the transaction log is damaged you can't do step 2
  - Store the log on a separate physical device to the database
  - The risk of losing both is then reduced