COMP90050 Advanced Database Systems
Final Exam Notes

Frederick Zhang Hill Feng

08/06/2017

Contents

1 Introduction (INSIGNIFICANT)

1.1 Database Technologies

1. Simple file systems like UNIX file system
   - Usually very fast
   - Can be less reliable
   - Application dependent optimisation
   - Hard to maintain (concurrency issues)
   - Lack of features – many of the features exist in RDB need to be incorporated – unnecessary code development and potential increase in unreliability
2. Relational database systems (RDB)
   - Proven technology (over 40 years)
   - Can be slow for some applications
   - Very reliable
   - Application independent optimisation
   - Well suited applications related to commerce
   - Now viable in many applications thanks to better implementations and optimisations
   - Some RDBs support OO model, e.g. Oracle, DB2
   - Widely used
3. Object oriented database systems
   - Can be slow in some applications
   - Reliable
   - Limited application independent optimisation although some newer developments make them very suitable
   - Well suited for applications requiring complex data
   - XML based Database Systems are becoming dominant
   - Many applications now use XML documents
   - Some RDBs also support limited OO models and XML handling
   - Seldom used nowadays
4. Deductive database systems
   - Generalisation of RDBs, e.g. allow recursion
   - No commercially available systems like OODBs
   - Many applications do not require the expressive power of these systems (e.g. commerce applications)

- Many RDBs provide some of the functionality of deductive database systems, e.g. supporting transitive closure operation (SQL2)

5. Key-value pair based database systems
   - Very fast, high parallel processing of large data
   - MapReduce and Hadoop are examples
   - Many applications do not require the expressive functionality of transaction processing (e.g. web search, big data analysis)
   - Atomic update at key-value pair level only
6. NoSQL
   - Mechanism other than tabular relations
   - A result of Web 2.0, Facebook, Amazon needs
   - Simple design, high scalability
   - Less consistency, more availability, partition tolerance and speed
   - Allows replication
   - Eventual consistency

1.2 Database Architectures

1. Centralised database systems
   - Data is stored in one location
   - System may contain several processors
   - System administration is simple
   - Optimisation process is generally very effective
   - Well proven database technology
   - May not be suitable for applications that require data distribution
   - Most dominant technology even now
   - Cloud computing/data farms/data centres are examples of this
2. Client-server
   - Processing is shared between client processes and server processes
   - Client and server may be in different locations
   - Client generally provides user interfaces for input and output
   - Server provides all the necessary database functionality
   - System administration is relatively simple
   - System recovery is similar to centralised database systems
3. Distributed database systems
   - Data is distributed across several nodes

- Nodes are connected by some communication network
- Users do not need to know the location of data
- System provides necessary concurrency, recovery and transaction processing
- Hard to process queries efficiently
- Transaction processing can be very inefficient
- System administration is very hard
- System recovery after a crash is complicated
- Potential reliability with replication but it may introduce more problems

4. World Wide Web

   - Data is stored in many locations
   - Data consistency is not guaranteed
   - Several owners of data and therefor no certainty of availability and consistency
   - Optimisation process is generally very ineffective
   - Evolving database technology – no standards except for XML/HTTP
   - Security could be a potential problem
   - Multiple source for accessing data (topic of source trusting)
   - Very convenient to use
   - Notions of transactions is much more difficult to enforce
   - RDB models are slowly appearing in WWW systems

5. Grid databases

   - Similar to distributed database systems except that data may be stored in standard file system and dedicated applications directly access the data
   - Data and processing are shared among a group of computer systems which may be geographically separated
   - Designed for particular purpose, e.g. scientific application
   - Administration are done locally by each of the owners of the system
   - No general purpose systems
   - Reliability and security are not well studied or developed
   - Small user community
   - Kind of outdated

6. P2P databases

   - Data and processing are shared among a group of computer systems which may be geographically separated
   - Nodes can join and leave at will
   - Duplication for reliability
   - Designed for particular usage, e.g. scientific application

- Administration of such system is done by the owners of the data
- Unlike in Grid Database systems data of an owner can be distributed among many nodes
- No general purpose systems
- Reliability and security of such systems can be worse than Grid Databases and are not well studied
- Finer level of granularity and hence can allow much higher parallelism

1.3 Basic Hardware

1.3.1 Moore's Law

Memory chip capacity grows at a rate (doubles every 18 moths since 1970)

$$= 2^{\frac{(year-1970)\times 2}{3}} \ Kb/chip$$

1.3.2 Joy's Law

Processor performance grows at a rate (doubles every two years since 1984)

$$= 2^{\frac{year-1984}{2}} \ mips$$

1.3.3 Hit ratio

$$Hit\ ratio = \frac{references\ sastisfied\ by\ cache}{total\ references}$$

Effective memory access time:

$$EA = H \times C + (1 - H) \times S$$

where H = hit ratio, C = cache access time, S = memory access time

1.3.4 Disk access time

$$Disk\ access\ time = seek\ time + rotational\ time + \frac{transfer\ length}{bandwidth}$$

1.3.5 RAID

RAID 0 Block level striping, MTTF reduces by a factor of 2

RAID 1 Mirroring, MTTF quadratic improvement

RAID 2 Bit level striping, MTTF decrease by half as in RAID 0

RAID 3 Byte level striping, MTTF $\frac{1}{3}$ of RAID 1

RAID 4 Block level level striping, dedicated disk for parity blocks, MTTF same as RAID 3

RAID 5 Block level level striping, no dedicated disk for parity, MTTF same as RAID 3

RAID 6 Block level level striping, similar to RAID 5 except two parity blocks used, MTTF is of the order of $MTTF^3/10$

1.3.6 Transmit time

$$transmit\ time = \frac{distance}{Cm} + \frac{message_bits}{bandwidth}$$

where Cm = speed of light in the medium (200 million meters/sec)

This means we can never reduce latency and the message length should be large to have an efficient transmission.

1.4 Transaction Processing

1.4.1 Definition

A transaction is collection of operations that need to be performed on physical and abstract application state

1.5 ACID (IMPORTANT)

Atomicity A transaction's changes to the state are atomic (all or none)

Consistency A transaction is a correct transformation of the state. Actions taken as a whole do not violate the integrity of the application state assuming transactions are correct programs

Isolation Even when several transactions are executed simultaneously, it appears to each transaction T that others executed either happen before T or after T but not at the same time

Durability State changes committed by a transaction survive failures

1.6 Provided functionality

1. Resource managers
   Responsible for providing ACID operations on objects they implement, e.g. data base systems, persistent programming languages, spoolers, reliable communication mangers, etc

2. Durable state
   The application designer represents the application state as durable data stored by the resource manager

3. Transaction programming language
   Allows expression of group of actions as an atomic operation, e.g. SQL

1.7 Transaction processing system features

- Application development features
  - Provides facilities for building applications automatically by means of a generic application
  - In most cases 90% of the applications can be developed using the application builder

  - Domain specific part of the application is written in a conventional host language such as C, Java, Cobol, Fortran, SQL, etc.

- Repository features
  - These features are similar to build tools and version control systems
  - It keeps track of dependencies and changes

- TP monitor
  - Responsible for authentication
  - Provides execution environment for processing the requests

- Data communication features provide reliable communication by means of
  - Recording every message sent or received on stable storage
  - Re-sending messages if acknowledgements are not received in time
  - Ignoring duplicate messages received, however, sending acknowledgements to every received message

- Database features
  - Provides stable repository
  - Location independency (distributed systems)
  - Provides standard interfaces to data definition and manipulation, e.g. SQL
  - Data control

- Security, concurrency

- Database display
  - Provides browsing and report producing tools

- Database operations utilities for administering the database
  - loading databases
  - archiving
  - recovery utilities
  - tuning
  - performance monitoring
  - application development tools

1.8 Software

1.8.1 Definitions

- Process is a virtual processor
- Process address space
- Protection domain – a process can perform any valid operations within a domain and any facilities provided by other domains require explicit transfer of control
- Threads are light-weighted processes
- Some OS's can schedule threads directly and therefore can run in parallel when the system contains multiple processors
- Threads can make wring some software development easier but debugging can be extremely difficult

1.8.2 Message passing

1. Session based
   usually efficient and reliable for an interaction that requires several message transfers
2. Datagrams
   efficient for single message transfers

Note: session-based communication is built on top of datagrams on the Internet

1.9 Scheduling

1.9.1 Goals and Explanations

General goals of scheduling are

- maximise utilisation
- minimise response time

The above goals are conflicting
Given U = utilisation, S = average service time and R = average response time
R is proportional to $S/(1-U)$
Hence scheduler should not push for very high utilisation

2 Reliability (NORMAL)

2.1 Probability bases (IMPORTANT)

Assuming A and B are independent

$$\begin{aligned} P(A \vee B) &= P(A) + P(B) - P(A \wedge B) \\ &= P(A) + P(B) - P(A) * P(B) \\ &\approx P(A) + P(B) \end{aligned}$$

Mean time to event $MT(A) = \frac{1}{P(A)}$

2.2 Module availability (IMPORTANT)

Measure the ratio of service accomplishment to elapsed time

$$= \frac{\textit{Mean time to failure}}{\textit{Mean time to failure} + \textit{Mean time to repair}}$$

2.3 Fault types (INSIGNIFICANT)

Environmental Such as cooling, power, weather, data communication lines, fires, earthquakes, tsunami, wars, sabotage

Operational System administration, system configuration and system operation procedures

Maintenance Procedures for regular maintenance, replacement of hardware on regular basis

Hardware Devices, cooling

Software Programs

Process Strikes, management decision for shutdowns

Civil Wars

2.4 Schemes

Failvote Use two or more modules and compare their output. Stops if there are no majority outputs agreeing. It fails twice as often with duplication but gives clean failure semantics

Failfast (Voting) Similar to failvoting except the system senses which modules are available and then uses the majority of the available modules

2.5 N-plex repair (IMPORTANT)

In this configuration the faulty equipment is repaired with an average time of MTTR as soon as fault is detected

Usually $MTTF >> MTTR$, hence probability of a particular module is not available:

$$\begin{aligned} &= MTTR/(MTTF + MTTR) \\ &\approx MTTR/MTTF \end{aligned}$$

Probability of $(n-1)$ modules unavailable

$$P_{n-1} = (\frac{MTTR}{MTTF})^{n-1}$$

Probability of a module N fails

$$P_f = 1/MTTF$$

Probability that N-plex system fails

$$= (\frac{n}{MTTF})(\frac{MTTR}{MTTF})^{n-1}$$

MTTF of N-plex system

$$= (\frac{MTTF}{n})(\frac{MTTF}{MTTR})^{n-1}$$

2.6 Old master - new master technique

- record all updates (transactions) to be performed in a separate file (stable store)
- at night (usually) produce a separate new (next day) master using the old (previous day's) master and the batched updates (transactions)

Fault tolerance because we can always produce new master file for the next day as long as we have old master file and the transactions to be performed

The problem is not online

2.7 Software reliability and hardware reliability

- Hardware reliability requires tolerating components failures
- Software reliability requires tolerating design and coding faults
- The distinction between Hardware and Software is becoming less at most hardware units have substantial amount of software components. These systems are generally called embedded systems.

2.8 N-version programming

- use n programs which are run in parallel, taking majority vote for each answer
- the advantage is that the diversity of design and coding can mask many failures

2.9 How to improve software reliability

2.9.1 Process pairing

1. Periodic transfer of data
   One process called Primary does all the work until it fails. The second process called Backup takes over the primary and continues the computation. In order to do this, Primary need to tell on a regular basis that it is alive and also transmit its state to the secondary

2. Checkpoint-restart
   The primary records its state on a duplexed storage module. At takeover the secondary starts reading the state of the primary from the duplexed storage and resumes the application

3. Checkpoint messages
   The primary sends its state changes as messages to the backup. At takeover the backup gets its current state from the most recent checkpoint message

4. Persistent
   Backup restarts in the null state and lets Transaction mechanism to clean up all uncommitted transactions. This is the approach taken by the most database systems.

2.9.2 Others

- Highly available storage
  - write to several storage modules
  - have some kind of checksum to make sure that the data read is correct with a very high probability
  - disk monitoring is an example of this
  - shadowing is another mirroring technique which allows atomic write operations
- Highly available processes
  - process pairing
  - transactions bases restart
  - checkpoint restart

3 Transaction Models (IMPORTANT)

3.1 Atomic Disk writes

Either entire block is written correctly on disk or the contents of the block is unchanged. It requires **duplex write**:

- a block of data is written two disk blocks sequentially
- determine whether the contents of a disk block has an error or not by checking its CRC.
- each block is associated with a version number
- the block with the latest version number contains the most recent data
- if one of the writes fail, system can issue another write to the disk block that failed
- it always guarantees at least one block has consistent data

3.2 Logged write

Similar to duplex write except one of the writes goes to a log.

3.3 Type of Actions

Unprotected actions No ACID property

Protected actions these actions are not externalised before they are completely done. These actions are controlled and can be rolled back if required. These have ACID property.

Real actions These are real physical actions once performed cannot be undone. In many situations, **atomicity is not possible with real actions**.

3.4 Flat Transaction

Flat transactions do not model many real applications.

```
BEGIN WORK
  S1: book flight from Melbourne to Singapore
  S2: book flight from Singapore to London
  S3: book flight from London to Dublin
END WORK
```

From Dublin if we cannot reach our final destination instead we wish to fly to Paris from Singapore and then reach our final destination. If we roll back we need to redo the booking from Melbourne to Singapore which is a waste.

3.5 Nested Transaction

Periodically save transaction.

3.5.1 Commit rule

- A subtransaction can either commit or abort, however, commit cannot take place unless the parent itself commits.
- Subtransactions have Atomicity, Consistency, and Isolation properties but not have Durability property unless all its ancestors commit.
- Commit of a sub transaction makes its results available only to its parents, not available to its siblings.

3.5.2 Roll back rules

- If a subtransaction rolls back all its children are forced to roll back

3.5.3 Visibility rules

- Changes made by a sub transaction are visible to the parent only when the sub transaction commits. Whereas all objects of parent are visible to its children. Implication of this is that the parent should not modify objects while children are accessing them. This is not a problem as parent is not run in parallel with its children.

3.6 Transaction Processing monitor (INSIGNIFICANT)

1. Batch processing
2. Time-sharing processing
3. Real-time processing
4. Client-Server processing
5. Transaction-Oriented processing

3.7 Transaction Processing services

3.7.1 Heterogeneity

If the application needs access to different database systems local ACID properties of individual database systems is not sufficient. Local TP monitor needs to interact with other TP monitors to ensure ACID property(additional layer needs to be added). A form of 2 phase commit protocol has to be employed for this purpose

3.7.2 Control communication

If the application communicates with other remote processes, the local TP monitor should maintain the communication status among the processes for it be able to recover from a crash.

3.7.3 Terminal management

Since many terminals run client software the TP monitor should provide appropriate ACID property between the client and the server processes.

3.7.4 Presentation service

This is similar to terminal management in the sense it has to deal with different presentation (user interface) software.

3.7.5 Context management

maintaining the sessions etc.

3.7.6 Start/Restart

There is no difference between start and restart in TP based system.

3.8 Transaction process structure

3.8.1 One process per terminal performing all possible requests

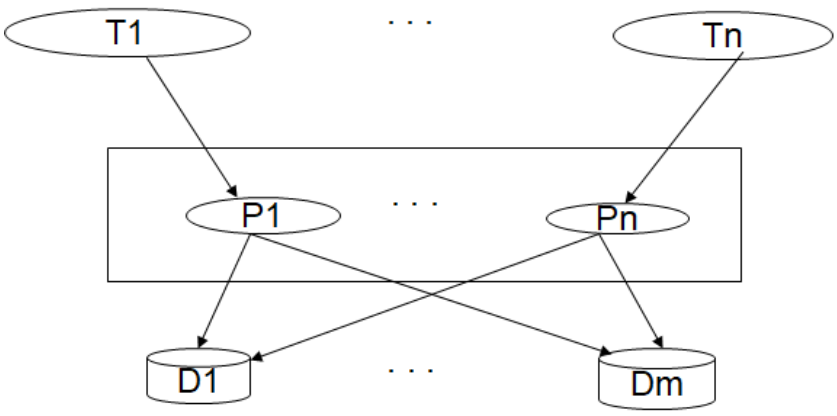

Figure 1

Very expensive if we have many terminals and files.

3.8.2 One process for all terminals performing all possible requests

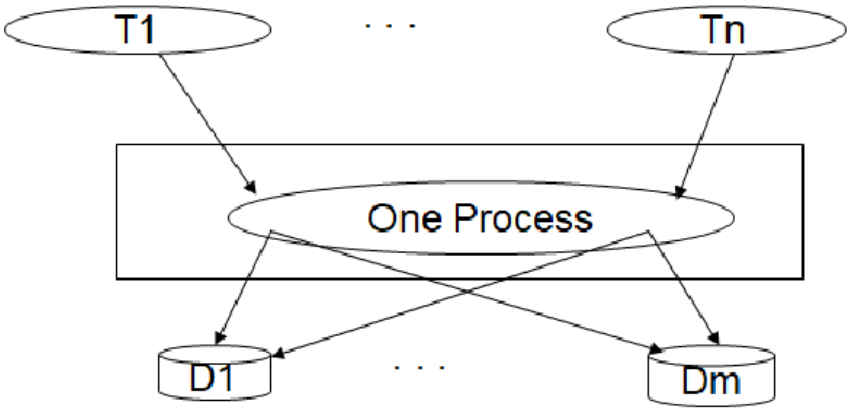

Figure 2

Context switching due to scheduling can cause poor response. Single large program has to deal with all kinds of terminals

3.8.3 Multiple communication processes and servers

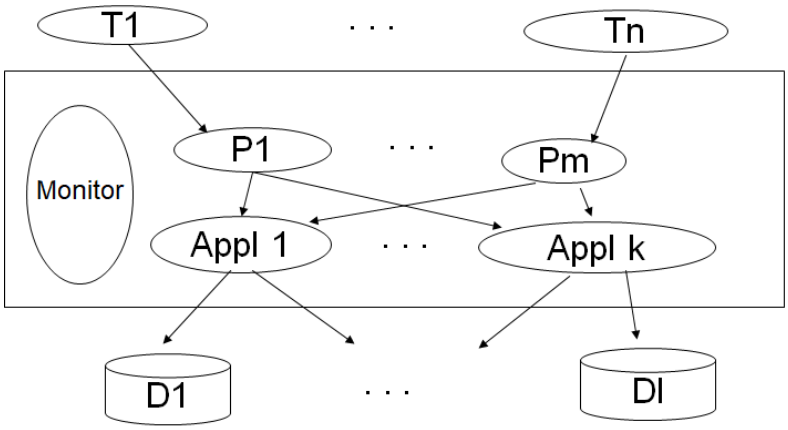

Figure 3

4 Locks and Concurrency (NORMAL)

4.1 Implementation of exclusive access (atomic operations)

1. Dekker's algorithm

   - needs almost no hardware support although it needs atomic reads and writes to main memory

- the code is very complicated to implement if more than two transactions/processes are involved
- harder to understand the algorithm for more than two processes
- takes lot of storage space
- uses busy waiting
- efficient if the lock contention is low

2. OS supported primitive such as lock and unlock

   - through an interrupt call, the request is passed to the operating system to implement desired lock request
   - no special hardware
   - are very expensive in general
   - do not use busy waiting and therefore more effective if lock contention is high
   - solution is independent of number of processes
   - machine independent unlike the previous scheme
   - need to use spin locks to implement OS primitives in the kernel if the system has symmetric multi processors (all modern processors do support some form of spin locks)

4.2 Spin locks

Atomic machine instructions such as test and set or swap instructions

- need hardware support – should be able lock bus for two memory cycles (reading and writing). During this time no other devices' access is allowed to this memory location
- are very simple to implement
- use busy waiting
- algorithm does not depend on number of processes unlike Dekker's algorithm
- are very efficient if the lock contentions are low
- are necessary in symmetric multiprocessor machines

4.3 Deadlocks (IMPORTANT)

4.3.1 Definition

In a deadlock situation, each member of the deadlock processes is waiting for another member to release the resources it wants.

4.3.2 Solutions

- Have enough resources so that no waiting occurs
- Simply do not allow a process to wait simply rollback. This can create live locks which are worse than deadlocks
- Linearly order the resources and request of resources should follow this order. That is a transaction after requesting ith resource can request jth resource. if $j > i$, this type of allocation guarantees no cyclic dependencies among transactions
- Pre-declare all the resources that are needed and allocate all the resources in a single request
- Allow a transaction to wait for a certain maximum time on a lock and force it to rollback (timeout). Many successful systems (IBM, Tandem) have chosen this approach
- Periodically check the resource dependency graph for cycles. If a cycle exists rollback (terminate) one or more transactions to eliminate cycles (deadlocks). The chosen transactions should be cheap.
- Many distributed database systems maintain only local dependency graphs and use timeouts for global deadlocks

4.3.3 Phantom deadlocks

Phantom deadlocks can happen if we cannot build consistent dependency graphs across many independent database systems.

4.4 Probability

We assume

- *number transactions* $= n + 1 \approx n$ when n is large
- each transaction access r locks exclusively – that is it takes an exclusive lock sequentially
- total number of records in the database $= R$
- on average each transaction is holder $\frac{r}{2}$ locks approximately (minimum zero and maximum r)
- transaction just commenced holds none and transaction about to finish holds r of them
- average number of locks taken by the other n transactions $= n \times \frac{r}{2} = \frac{nr}{2}$

$$\text{The probability that a particular transaction waits for a lock} = \text{Requesting an lock on one of the } nr/2 \text{ locks held by other } n \text{ transactions} = nr/2 \text{ out of potential } R \text{ records} = \frac{nr}{2R}$$

The probability that a particular transaction waits for a lock $= \frac{nr}{2R}$

The probability that a transaction did not wait,

$$\sim pw(T) = (1 - \frac{nr}{2R})^r$$
$$\approx 1 - \frac{nr^2}{2R}$$

Note: $(1 + \varepsilon)^r = (1 + r \times \varepsilon)$ when ε is small

The probability that a transaction waits in its lifetime,

$$pw(T) = 1 - (1 - \frac{nr}{2R})^r = 1 - (1 - \frac{nr^2}{2R}) = \frac{nr^2}{2R}$$

Probability that a particular transaction waits for some transaction T1 and T1 waits for T is

$$pw(T) \times (pw(T1)/n) = \frac{nr^4}{4R^2}$$

Probability of any two transaction causing deadlock is

$$n \times nr^4/4R^2 = \frac{n^2r^4}{4R^2}$$

Probability of deadlock happening increases with $O(r^4)$ with respect to the number of locks taken and $O(n^2)$ with the number of concurrent transactions and inversely proportional to $O(R^2)$ with the size of the database

5 Isolation Concepts

Isolation guarantees consistency provided each transaction itself is consistent. Kinds of consistency can be stated as invariants: (INSIGNIFICANT)

- Checksum field/CRC of each page must be correct
- For each doubly linked list satisfy the property *prev(next(x))= x*
- Accounts balances must be positive
- If a record is inserted in EMPLOYEE relation a corresponding record must exist in ADDRESS relation
- A change in account balance should also be recorded in the ledger relation
- The reactor rods cannot be moved faster than a X-centimetres/sec

5.1 How to achieve isolation

5.1.1 Sequentially processing

Sequentially (batch processing) processing each transaction but this is generally not efficient and provides poor response times.

5.1.2 Concurrently processing

Goals:

- Concurrent execution should not cause application programs (transactions) to malfunction
- Concurrent execution should not have lower throughput or bad response times than serial execution.

In order to achieve isolation we need the concept of locking and/or restricting type of concurrent operations on an object.

Isolation properties can be studied by means of dependency graphs. I_i set of inputs (objects that are read) of a transaction T_i and O_i (objects that are modified) are its outputs.

$$O_i \cap (I_j \cup O_j) = \textit{empty for all } i \neq j$$

This approach cannot be planed ahead as it is not generally practical as in many situation inputs and outs may be state dependent.

5.2 Violation of Isolation (IMPORTANT)

Lost update a transaction write is ignored by another transaction.

```
E.g.   T2 read  < o,1 >
  T1 write < o,2 >
  T2 write < o,3 >     % T2 did not see the update of T1
```

Dirty read a transaction reads an object written by another transaction which again modifies the object afterwards

```
E.g.  T2 write < o,2 >
  T1 read  < o,2 >    %T1 is unaware of T2 changing
  T2 write < o,3 >
```

Non repeatable read a transaction reads an object twice and gets different values

```
E.g.  T1 read < o,1 >
  T2 write <o,2 >
  T1 read < o,2 >     % 2nd  read can result different value
```

5.3 Mode of Lock

	Mode of Lock		
Current Mode	Free	Shared	Exlusive
Shared request (SLOCK)	Compatible Request granted immediately Changes Mode from Free to Shared	Compatible Request granted immediately Mode Stays Shared	Conflict Request delayed until the state becomes compatible Mode Stays Exclusive
Exclusive request (XLOCK)	Compatible Request granted immediately Changes Mode from Free to Exclusive	Conflict Request delayed until the state becomes compatible Mode Stays Shared	Conflict Request delayed until the state becomes compatible Mode Stays Exclusive

5.4 Actions in Transactions

A history is some merge of the actions of a set of transactions

- READ
- WRITE
- XLOCK
- SLOCK
- UNLOCK
- BEGIN
- COMMIT can be replaced by UNLOCK A if SLOCK A or XLOCK A appears in T for any object A
- ROLLBACK can be replaced WRITE(UNDO) A if WRITE A appears in T for any object A and UNLOCK A if SLOCK A or XLOCK A appears in T for any object A

Some definitions:

Well-formed transactions A transaction is well formed if all READ, WRITE and UNLOCK operations are covered earlier by appropriate LOCK operations

Two phase transactions A transaction is two phased if all LOCK operations precede all its UNLOCK operations

5.5 Dependency relation

In a history sequence H, consisting of tuples of the form (T, action, object). If T1 and T2 are transactions, O is an object, and there exists indexes i and j such that i < j,

H[i] involves action a1 on O by T1 (i.e. H[i] = (T1, a1, O))

H[j] involves action a2 on O by T2 (i.e. H[j] = (T2, a2, O))

and there are no H[k] = (T',WRITE,O) for i < k < j, the dependency of T1 on T2 can be written as:

(T1, O, T2) if a1 = WRITE & a2 = WRITE';

or a1 = WRITE & a2 = READ;

or a1 = READ & a2 = WRITE.

This captures WRITE –> WRITE, WRITE –> READ, READ –> WRITE dependencies respectively.

Dependency relations:

$$DEP(H) = \{(Ti, O, Tj) | Tj \ depends \ on \ Ti\}$$

Dependency graph: Transactions are nodes, and object are the edges from the node Ti to Tj if (Ti, O, Tj) is in DEP(H).

5.5.1 Equivalent History

Two histories H1 and H2 are equivalent if DEP(H1) = DEP(H2). This implies that a given database will end up in exactly the same final state when the database is subjected to the sequence of operation by H1 or H2. A sample dependency graph is shown in Figure 4.

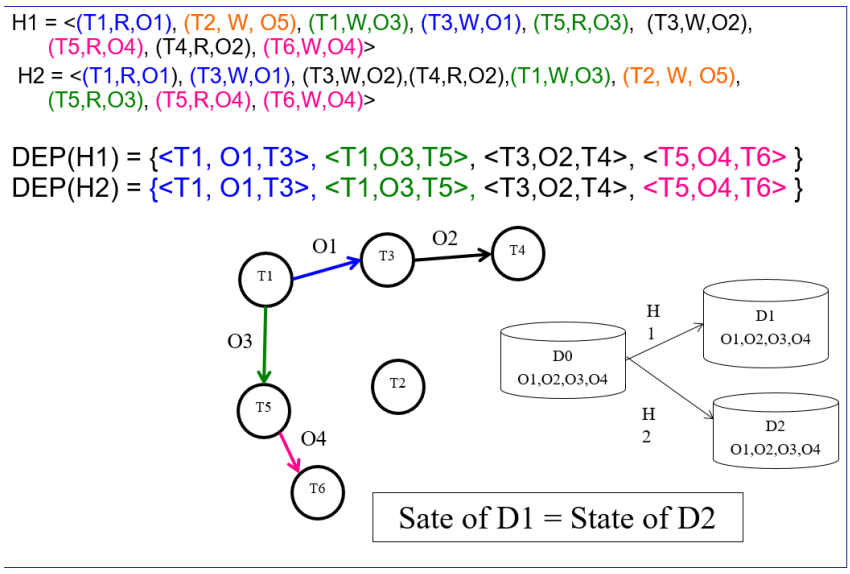

Figure 4: Dependency Sample

5.5.2 Isolated History

A serial history is history that is resulted as a consequence of running transactions sequentially one at a time. N transactions can result in a maximum of N! serial histories.

A serial history is an isolated history.

5.5.3 Wormhole

T1 precedes T2 is written as T1 << T2.
Before(T) = T' | T' << T
After(T) = T'| T << T'
T' is a wormhole transaction if

$$T' \in Before\,(T) \cap After\,(T)$$

That is T << T' << T. This implies there is a cycle in the dependency graph of the history. Presence of a wormhole transaction implies it is not isolated.

5.6 Theorems (IMPORTANT)

Wormhole theorem A history is isolated if and only if it has no wormholes.

Locking theorem If all transactions are well-formed (READ, WRITE and UNLOCK operation is covered earlier by a corresponding lock operation) and two-phased (all LOCK operations precede all its UNLOCK operations), then any legal (does not grant conflicting grants) history will be isolated.

Locking theorem (Converse) If a transaction is not well formed or is not two-phase, then it is possible to write another transaction such that it is a wormhole.

Rollback theorem An update transaction that does an UNLOCK and then does a ROLLBACK is not two phase.

5.7 Degrees of Isolation (VERY VERY IMPORTANT)

5.7.1 Degree 3

A Three degree isolated Transaction has no lost updates, and has repeatable reads. This is "true" isolation. Example: Bank, Stock
Lock protocol is two phase and well formed.
It is sensitive to the following conflicts: write->write; write ->read; read->write

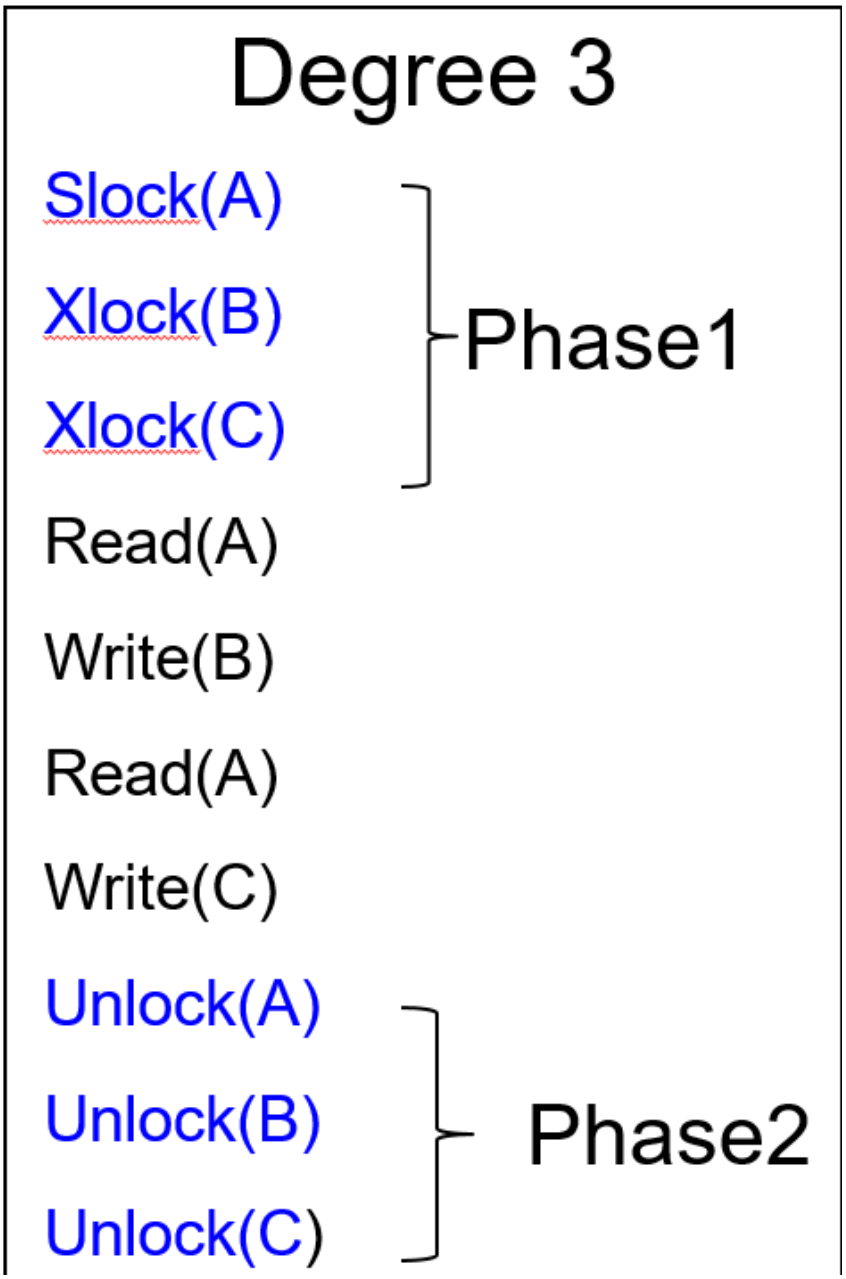

Figure 5: Degree 3

5.7.2 Degree 2

A Two degree isolated transaction has no lost updates and no dirty reads. Example: Ticket Sale System

Lock protocol is two phase with respect to exclusive locks and well formed with respect to Reads and writes. (Non repeatable reads.)

It is sensitive to the following conflicts: write->write; write ->read;

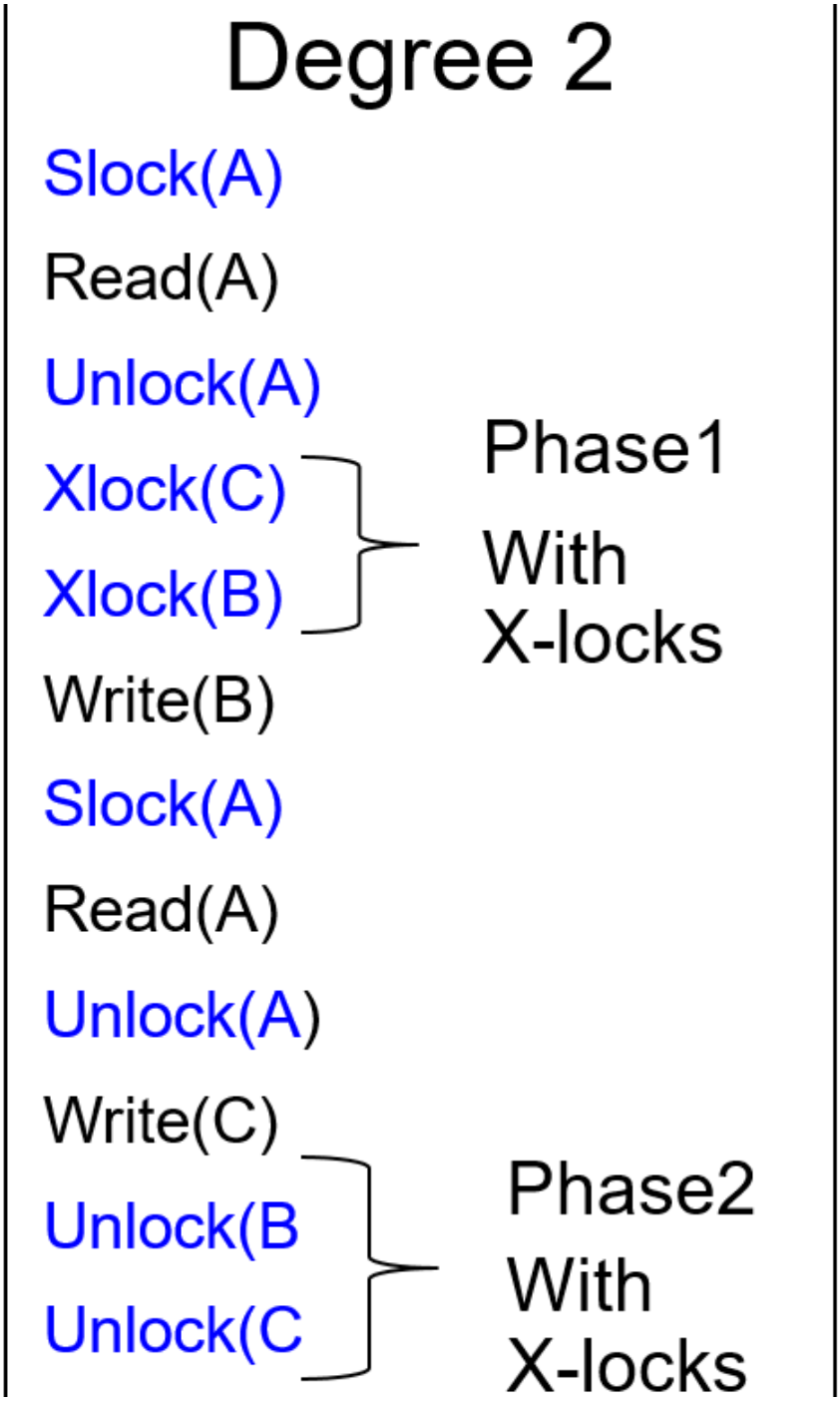

Figure 6: Degree 2

5.7.3 Degree 1

A One degree isolation has no lost updates. Example: Library

Lock protocol is two phase with respect to exclusive locks and well formed with respect to writes.

It is sensitive the following conflicts: write->write;

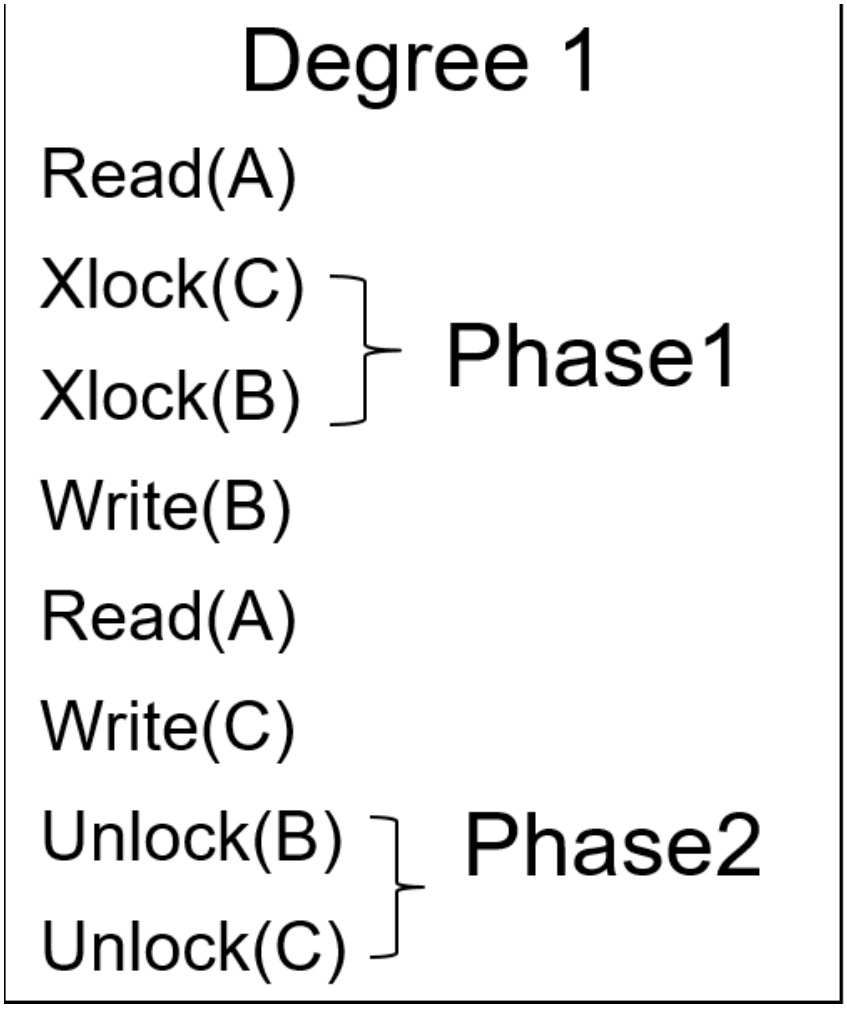

Figure 7: Degree 1

5.7.4 Degree 0

A Zero degree transaction does not overwrite another transactions dirty data if the other transaction is at least One degree. Example: Google

Lock protocol is well-formed with respect to writes.

It ignores all conflicts.

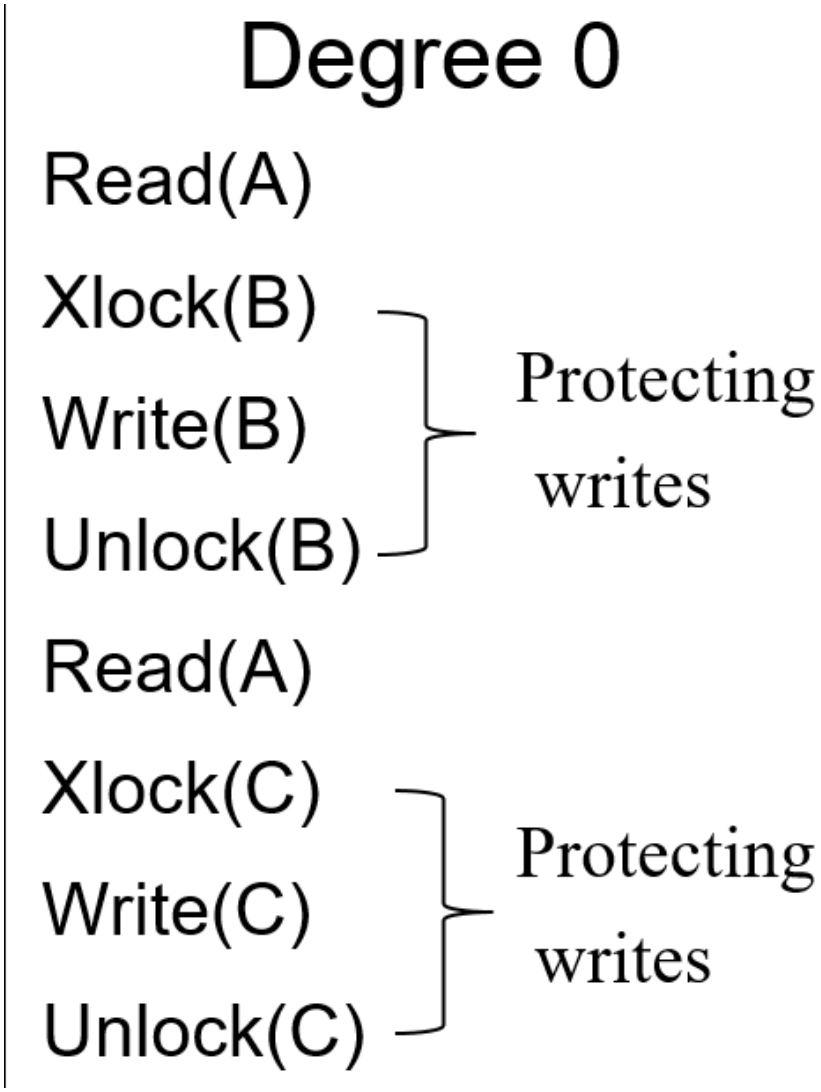

Figure 8: Degree 0

5.7.5 Comparisons (VERY VERY IMPORTANT)

Issue	Degree 0	Degree 1	Degree 2	Degree 3
Common	Chaos	Browse	Cursor stability	Isolated/ Serialisable/ repeatable reads
Protection	Let others run at higher level	Same as Degree 0 – no lost updates	No lost updates or dirty reads	No lost updates, no dirty reads, repeatable reads
Committed data	Writes visible immediately	Visible at the end of transaction	Same as Degree 0 and 1 plus do not read dirty data	Same as Degree 0,1 and 2 plus no dirty data reads (repeatable read)
Lock protocol	Set short Xlocks on data modified	Set 2 phase Locking on data modified	As in 1 degree plus short Slocks on data read	2 phase locking on both Slocks and Xlocks
Dirty data	No overwrites on dirty data (someone else modifying)	Others do not overwrite the data that is being modified	Same as degree 0 and 1 plus do not read dirty data	Same as degree 0, 1 and 2 plus no one modifies the data that is being read
Transaction structure	well formed writes	well formed writes / 2 phase writes	well formed reads and writes / 2 phase writes	well formed reads and writes / 2 phase reads and writes
Concur-rency	maximum	high	medium	low
Overhead	least	small	medium	medium
Rollback	Not possible	yes	yes	yes
Dependen-cies	none	write->write	write->write / write->read	write->write / write->read / read->write
Recovery	no	yes	yes	yes

5.8 Phantoms and Predicate locks (NORMAL)

If locks are taken at finest possible granularity then we may be performing updates which otherwise should be delayed.

Read and write sets can be defined by predicates (e.g. Where clauses in SQL statements) and associate a lock with such a set

When a transaction accesses a set for the first time,

1. Automatically capture the predicate and associate appropriate lock with such a predicate
2. Do set intersection with predicates of other transactions
3. Delay this transaction if it conflicts with others that is if there is an overlap which leads to a lock conflict.

Problems with predicate locks:

1. Set intersection = predicate satisfiability is NP complete (slow to impossible). Note: this is not a normal set intersection it is a predicate intersection.
2. Hard to capture predicates
3. Pessimistic: T1 locks all pages containing eye = blue T2 locks all pages containing hair= red Predicate says conflict, but DB may not have blue eyed and red haired person.

5.9 Granularity Locks (VERY VERY IMPORTANT)

The ideas is:

- Pick a fixed set of predicates
- They form a lattice or a tree
- Lock the nodes in this graph/lattice/tree

5.9.1 Tree locking and Intent Lock Modes

None no lock is taken all requests are granted

IS allows IS and S mode locks at finer granularity and prevents others from holding X on this node.

IX allows to set IS, IX, S, SIX, U and X mode locks at finer granularity and prevents others holding S, SIX, X, U on this node.

S allows read authority to the node and its descendants at a finer granularity and prevents others holding IX, X, SIX on this node

SIX allows reads to the node and its descendants as in IS and prevents others holding X, U, IX, SIX, S on this node or its descendants but allows the holder IX, U, and X mode locks at finer granularity. SIX = S + IX

U allows read to the node and its descendants and prevents others holding X, U, SIX, IX and IS locks on this node or its descendants.

X allows writes to the node and prevents others holding X, U, S, SIX, IX locks on this node and all its descendants.

Compatibility Mode of Granular Locks							
Current	None	IS	IX	S	SIX	U	X
Request	+|- (Next mode) + granted / - delayed						
IS	+(IS)	+(IS)	+(IX)	+(S)	+(SIX)	-(U)	-(X)
IX	+(IX)	+(IX)	+(IX)	-(S)	-(SIX)	-(U)	-(X)
S	+(S)	+(S)	-(IX)	+(S)	-(SIX)	-(U)	-(X)
SIX	+(SIX)	+(SIX)	-(IX)	-(S)	-(SIX)	-(U)	-(X)
U	+(U)	+(U)	-(IX)	+(U)	-(SIX)	-(U)	-(X)
X	+(X)	-(IS)	-(IX)	-(S)	-(SIX)	-(U)	-(X)

5.10 Dead Locks (IMPORTANT)

A dead lock sample:

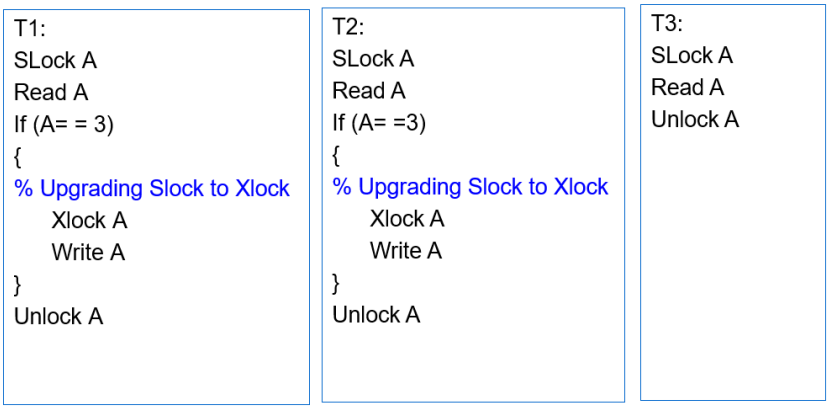

Figure 9: Dead lock sample

A solution:

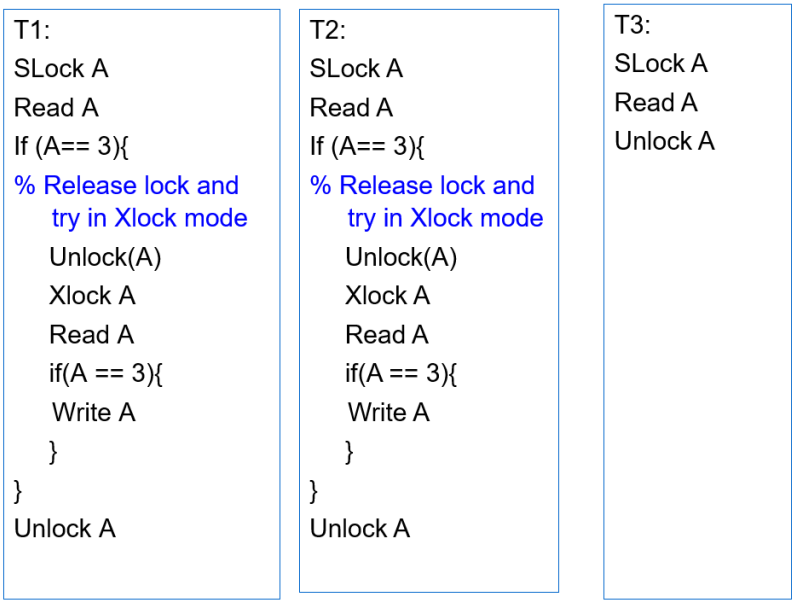

Figure 10: Solution 1

Problems: too many locks, read too many times

Update mode lock: An **Update lock** indicates the it will make some read and then make some write to the data item:

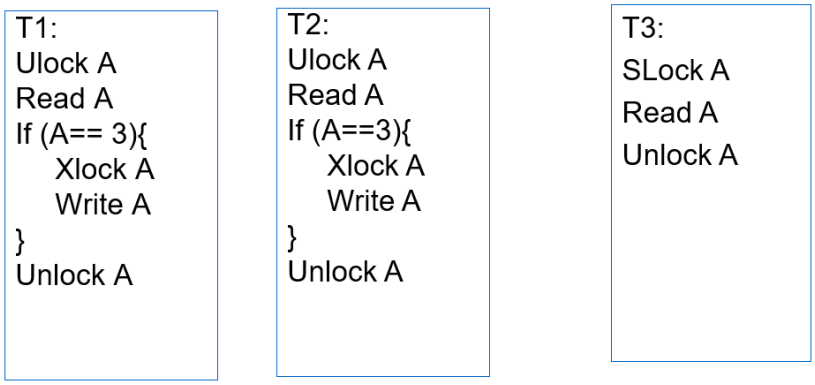

Figure 11

5.11 Convoy phenomenon (NORMAL)

FIFO scheduling can cause the problem of long queues called convoys.

When this happens the system can be extremely slow reducing throughput by a factor of 2 to 10. The problem is worse with multi processors as more concurrent processing is possible. This is another kind of priority inversion problem.

5.11.1 Example of convoy

Assume there is a single processor.

- Let P1, P2, P3, ... are processes running on the system
- P1 is a low-priority process acquires a log lock (storage for recording updates for recovery purpose) and execution is pre-emptied by a higher priority process P2
- P2 soon after requests log lock and gets suspended.
- P3, P4, ... all similarly get suspended as they request the same log lock
- Eventually P1 is scheduled. It completes writing to log, gives lock on log to the next process in the queue (FIFO) P2.
- Since P2 has a higher priority than P1 it pre-empties P1 and uses the log and then it gives the lock to next waiting process P3.
- Assuming P2 is not pre-empted by P3 it requests the log lock to perform second update and joins the queue at the end
- Now P3 is scheduled and uses the log and gives the lock to the next waiting process P4. Once again assuming P3 is not pre-empted it requests the log lock and joins the queue at the end
- These convoys are stable as long as all the processes request and release the locks on the log.

- Suspending and activating processes frequently(core problem) can slowdown the whole system which happens due waiting for locks.

5.11.2 Solution

- Do not schedule hotspot locks in FIFO basis. Wake up all the waiting process on release and let the scheduler choose the next process
- In a multiprocessor, spin on a lock for few hundred instructions rather than wait.
- Do not pre-empt processing having hotspot locks

5.12 Optimistic locking (IMPORTANT)

- Read the current quantity on hand into the application (take Slocks briefly read the contents and release the Slocks)
- Process the values read and determine new values to be written if any.
- At commit time take appropriate locks (Slocks on items to be tested and Xlocks on tuples to be modified) check the values read are not modified.
- If modified abort the transaction (release all locks) and restart the transaction
- If not make changes.

5.12.1 Example

```
Slock A % this shared lock can be ignored
Read A into A1
Slock B % this shared lock can be ignored
Read B into B1
Unlock A and B
% No locks are held
Loop: Compute new values based on A1 and B1
  % Start taking locks
  Slock A
  Read A into A2
  Slock B
  Read B into B2
  if (A1 == A2 & B1 == B2)
    Xlock C
    Write C
    commit
    Unlock A, B and C
  else   % read data is changed
    A1 = A2
    B1 = B2
    unlock A,  B
    goto Loop
  end
```

Once the condition is true – it is effectively 2 phase locking but duration of locking is very short, but it can force many repeated attempts due to failure of the condition.

Check the data source, which are used to compute new values, before XLock and Write. (Repeatable Read).

5.13 Snapshot Isolation (IMPORTANT)

5.13.1 Example

```
Read C into C1
Read D into D1
Loop:
  Read A into A1
  Read B into B1
  Compute new values for C and D based on A1 and B1
  % Start taking locks on records that need modification.
  Let new value for C is C3 and for D is D3
  Xlock C
  Xlock D
  Read C into C2
  Read D into D2
  if (C1 = = C2 & D1 = = D2)
    %    first writer commits
    write C3 to C
    write D3 to D
    commit
    unlock(C and D)
  else   % not first modifier
    C1 = C2
    D1 = D2
    unlock(C and D)
    goto Loop
end
```

Snapshot Isolation method is used in Oracle but it will not guarantee Serialisability. (Integrity constraint A+B >= 0; A = 100; B = 100; two transactions each deducts 100 from A or B, only one transaction can commit) However, its transaction throughput is very high compared to two phase locking scheme.

Check the data which are to be updated before write.(No lost update)

5.14 Time stamping

These are a special case of optimistic concurrency control. At commit, time stamps are examined. If time stamp is more recent than the transaction read time the transaction is aborted.

At the commit time, the system validates all the transaction's updates and writes updates to durable media. This model of computation unifies concurrency, recovery and time domain addressing

5.14.1 Example

T1: select average (salary) from employee

T2: update employee set salary = salary*1.1 where salary < 40000

If transaction T1 commences first and holds a read lock on a employee record with salary < 40000, T2 will be delayed until T1 finishes. But with time stamps T2 does not have to wait for T1 to finish!

6 Aries – Logging and Recovery (IMPORTANT)

6.1 Motivation

Atomicity Transaction may abort

Durability DMBS could be stopped

Recovery manager guarantees atomicity and durability

6.2 Buffer Caches (pool)

6.2.1 Assumptions

1. Data is stored on disks
2. Reading a data item requires reading the whole page from disk to memory
3. Modifying a data item requires reading the whole page from disk, modifying in memory and write the page back
4. Step 2 & 3 can be expensive and we can minimize disk IO by buffer cache
5. When buffer cache is full we need to evict some pages in it
6. Eviction needs to make sure that no one else is using the page and any modified pages should be copied to the disk
7. Latches are used to handle transaction concurrency IO to pages

6.2.2 Fix and unfix operations

1. `fix(pageid)`
   - reads pages from disk into buffer cache if it is not already in the cache
   - fixed pages cannot be dropped from the buffer as they are being used
2. `unfix(pageid)`
   - the page is not used by the transaction and an be evicted as far as the calling transaction is concerned

6.2.3 Force, No force and Steal (IMPORTANT)

Force Write to disk at commit

- Poor response time
- Provides durability

No Force Leaves pages as long as possible in memory

- Faster response, higher efficiency
- Durability problems

Steal In No Force, allowing writing pages to disk even if they still have locks from transactions (because buffer size limitation, this is needed when other transactions want to read something else from disk)

6.3 Logging

6.3.1 Definition

Log: An ordered list of minimal info about REDO/UNDO actions
Example: `<XID, pageID, offset, length, old, new>`

6.3.2 Write-Ahead Logging (WAL)

- Must force the log record for an update before the corresponding data page gets to disk (stolen)
  - guarantees Atomicity
- Must write all log records to disk (force) for a transaction before commit
  - guarantees Durability

6.3.3 Logs in WAL

- Each log record has a unique incremental Log Sequence Number (LSN)
- Each data page contains a pageLSN (the most recent LSN related to the page)
- System keeps track of flushedLSN
- Before a page is written to disk make sure $pageLSN \leq flushedLSN$

Example: `<prevLSN, XID, pageID, offset, length, old, new>`

6.3.4 Transaction Table

- One entry per active transaction
- Contains XID, status (running/committed/aborted) and lastLSN

6.3.5 Dirty Page Table

- One entry per dirty page in buffer pool
- Contains recLSN (recent LSN) – the LSN of the log record which first caused the page to be dirty since loaded into the buffer cache from disk

6.3.6 Normal Execution of a Transaction (Example of Technique Combination)

- Series of reads & writes, followed by commit or abort
- Strict 2PL (2 phase locking)
- STEAL, NO-FORCE buffer management, with WAL

6.4 Checkpointing and Logging

6.4.1 Definition

Periodically, the DBMS creates a checkpoint, in order to minimize the time taken to recover in the event of a system crash.

Components of a checkpoint:

- Begin checkpoint record: indicates when checkpoint began
- End checkpoint record
  - transaction table
  - dirty page table
- Store LSN of checkpoint record in a safe place

6.5 Normal transactions (IMPORTANT)

6.5.1 Transaction abort

- Before restoring old value of a page, write a CLR (Compensation Log Record)
  - Continue logging while UNDO
  - CLR has one extra field undonenextLSN (points to the next LSN to undo)
  - CLRs never undone (but might be redon when repeating history: guarantees Atomicity)
  - At the end of UNDO, write an "end" log record

6.5.2 Transaction commit

- Write commit log record
- All log records up to transaction's lastLSN are flushed
  - Guarantees that $flushedLSN \geq lastLSN$
  - Note that logs flushes are sequential, synchronous, fast
  - Many log records per log page (efficient multiple write)
- `Commit()` returns
- Write end log record

6.6 Crash recovery (IMPORTANT)

6.6.1 Big picture

- Start from a checkpoint in master record
- Three phases

Analysis	Figure out which transactions committed/failed since checkpoint
REDO	Repeat history
UNDO	Undo effects of failed transactions

6.6.2 Phase 1: Analysis

- Reconstruct state at checkpoint
  - via end_checkpoint record
- Scan log forward from checkpoint

End record	Remove transaction from transaction table
Other records	Add transaction to transaction table, set $lastLSN = LSN$, change transaction status on commit
Update record	If P not in Dirty Page Table, add P to D.P.T., set its $recLSN = LSN$

6.6.3 Phase 2: REDO

- Repeat history to reconstruct
  - Reapply all updates (including aborted transactions) and CLRs
- Scan forward from log record containing smallest recLSN in D.P.T. For each CLR or update log record (say, with LSN), redo unless:
  - Affected page is not in D.P.T, or (page had been flushed after the log record, no need to redo)
  - Affected page is in D.P.T. but has $recLSN > LSN$, or (the log record once made the page dirty, but the page had been flushed after that)

- $pageLSN\ in\ DB \geq LSN$ (page had been flushed after log)

- To REDO an action
  - Reapply logged action (Note: this happens in the buffer pool)
  - Set $pageLSN = LSN$, no additional logging (since $pageLSN \leq flushLSN$ based on WAL, the data should be in the disk. even if it crashed while redoing, we could simply restart again from checkpoint)

6.6.4 Phase 3: UNDO

- Construct ToUndo list

  $ToUndo = \{I | I$ is a lastLSN of a loser transaction$\}$ We can construct this list from transaction table

- Repeat until ToUndo is empty:
  - Choose the largest LSN among ToUndo
  - If this LSN is a CLR and $undonenextLSN = NULL$
    - Write an End record for this transaction (crashed after transaction abort and about to finish undoing)
  - If this LSN is a CLR and $undonenextLSN != NULL$
    - Add undonenextLSN to ToUndo (crashed after transaction abort but not yet finished undoing)
  - Else this LSN is an update. Similar as transaction abort
    - Undo the update
    - Write a CLR
    - add prevLSN to ToUndo

6.7 Distributed recovery

6.7.1 Two phase commit

- Phase 1
  - Coordinator sends a prepare message to each subordinate
  - On receiving a prepare message, a subordinate decides to either commit or abort its sub-transaction. It force-writes an abort or prepare log record and sends no or yes to coordinator

- Phase 2
  - If all yes, coordinator force-writes a commit log record and sends commits to all subordinates. If not, force-writes an abort log and sends abort to all subordinates
  - When a subordinate receives an abort, it force-writes an abort log record and sends an acknowledgement to the coordinator. It aborts the sub-transaction

6.7.2 Recovery

- If we have a commit or abort log of T, the status clear. We redo or undo T as in centralised database. The coordinator needs to send messages to subordinates and wait for acknowledgements
- If we have only prepare log for T and the site is a subordinate, we must repeatedly contact the coordinator to respond for commit or abort transaction. After receiving a message the rest of actions are 2PC

6.8 Different LSNs

Table 1: Different LSNs

Name	Logical Location	Physical Location
LSN	Logs	Disk (fast)
pageLSN	Pages	Disk (data area)
lastLSN	Transaction Table	Memory (or checkpoint)
recLSN	Dirty Page Table	Memory (or checkpoint)
flushedLSN	DBMS	Memory
prevLSN	Logs	Disk (fast)
undonenextLSN	Logs (CLR)	Disk (fast)

7 Transaction Processing Council

7.1 Aims

- Create good benchmarks
- Take the role to see the reported benchmarks are valid

7.2 History

1980 TP1 just debit credit– no network or computing cost included

1985 DebitCredit debit credit– with network, computing and maintenance cost included with 95% completions within 1 sec time

1988 TPC-A Debitcredit– with network, computing and maintenance cost included with 90% completions with in 2 secs time

1990 TPC-B debit credit– with no network costs for benchmarking servers

1998 TPC-W Web Commerce benchmark measured OLTP and browsing performance only of the web server, excluding the network and human interaction

8 CAP (INSIGNFICANT)

8.1 Theorem

Consistency All nodes should see the same data at the same time with or without data replication

Availability Node failures do not prevent survivors from continuing to operate implies data replication

Partition-tolerance The system continues to operate despite network partitions implies data replication

8.2 Importance

- The future of databases is distributed
- CAP theorem describes the trade-offs involved in distributed systems
- A proper understanding of CAP theorem is essential for making decisions about the future of distributed database design – what is important – CA, CP or AP

8.3 Types of Consitency

Strong Consistency After the update completes, any subsequent access will return the same updated value.

Weak Consistency It is not guaranteed that subsequent accesses will return the updated value

Eventual Consistency Specific form of weak consistency. It is guaranteed that if no new updates are made to object, eventually all accesses will return the last updated value (e.g., propagate updates to replicas in a lazy fashion)

8.3.1 Eventual Consistency Variations

Causal consitency Processes that have causal relationship will see consistent data

Read-your-write consistency A process always accesses the data item after it's update operation and never sees an older value

Session consistency As long as session exists, system guarantees read-your-write consistency. Guarantees do not overlap sessions

Monotonic read consistency If a process has seen a particular value of data item, any subsequent processes will never return any previous values

Monotonic write consistency The system guarantees to serialize the writes by the same process

8.4 Partitioning Examples

8.4.1 Data Partitioning

Different data may require different consistency and availability

Example:

- Shopping cart: high availability, responsive, can sometimes suffer anomalies
- Product information need to be available, slight variation in inventory is sufferable
- Checkout, billing, shipping records must be consistent

8.4.2 Operational Partitioning

Each operation may require different balance between consistency and availability

Example:

- Reads: high availability; e.g.., "query"
- Writes: high consistency, lock when writing; e.g., "purchase"

8.4.3 Functional Partitioning

System consists of sub-services. Different sub-services provide different balances

Example: A comprehensive distributed system

- Distributed lock service (e.g., Chubby) : Strong consistency
- DNS service: High availability

8.4.4 User Partitioning

Try to keep related data close together to assure better performance

Example: Craglist, Might want to divide its service into several data centers, e.g., east coast and west coast. Users get high performance (e.g., high availability and good consistency) if they query servers closet to them. Poorer performance if a New York user query Craglist in San Francisco

8.4.5 Hierarchical Partitioning

Large global service with local "extensions". Different location in hierarchy may use different consistency.

Example:

- Local servers (better connected) guarantee more consistency and availability
- Global servers has more partition and relax one of the requirement