Contents

Introduction

- Airline reservations, Credit Card Processing, Online Shopping, etc. Are example of very large real-world database systems
  - Can potentially have thousands of users performing operations at the same time
  - Operations can be complex, involved a number of separate actions
    - Create booking
    - Update remaining seats
    - Charge credit card
    - Issue confirmation
    - ...
- *Transactions* and *Concurrency Control* are the ways in which a DB manages complex processes and multi-user access
- Transaction
  - A logical unit of DB processing that must be completed in its entirety to ensure correctness
- Concurrency Control
  - Used when two operations try to access the same data at the same time

Transactions

- A transaction includes one or more DB access operations
  - Insertion
  - Deletion

  - Modification
  - Retrieval

- Transactions where the DB operations retrieve data, but don't update any information
  - Read-Only Transactions
- Transactions which update the DB
  - Read-Write Transactions

```
BEGIN_TRANSACTION
    READ or WRITE operation1
    READ or WRITE operation2
    ...
    READ or WRITE operationN
END_TRANSACTION
COMMIT or ROLLBACK
```

- The end of a transaction is signalled by either
  - commit (successful termination)
  - rollback or abort (unsuccessful termination)
- Commit completes the current transaction, making its changes permanent
- Rollback will undo the operations in the current transaction, cancelling the changes made to the DB

Transaction Failure

- There are a number of reasons why a transaction might fail
  - System crash
    * Hardware, Software or Network Error
  - Transaction Error
    * Divide by zero
    * Incorrect attribute reference
    * Incorrect parameter value
  - Check can be placed in the transaction
    * Insufficient funds to make a withdrawal

Transaction Properties

- All transactions should posses the ACID properties
  - Automicity
    - A transaction is an atomic unit of processing
    - It should either be performed in its entirety or not performed at all
  - Consistency Preservation
    - A transaction should preserve the consistency of the DB
    - It sould take the database from one consistent state to another
  - Isolation
    - A transaction should appear as though it is being executed in isolation
    - The execution of a transaction should not be interfered with by any other transactions executed concurrently
  - Durability (or permanency)
    - The changes applied by a comitted transation must persist in the database
    - These changes must not be lost because of any failure

Concurrency Control

- Consider the following two transactions

Flight Transfer

```
read(reserved_seats_X)
reserved_seats_X = reserved_seats_X-N
write(reserved_seats_X)
read(reserved_seats_Y)
reserved_seats_Y = reserved_seats_Y+N
write(reserved_seats_Y)
```

Flight Reservation

```
read(read(reserved_seats_X)
reserved_seats_X = reserved_seats_X+M
write(reserved_seats_X)
```

- A number of problems can occur if these two simple transactoins are run concurrently

Lost Update

- This occurs when two or more transactions:
  - Access the same data item
  - Are executed concurrently
  - Are interleaved in such a way that results in an incorrect value being written to the database

Temporary Update

- This occurs when two or more transactions:
  - Access the same data item
  - Are executed concurrently
  - Are interleaved
  - One transaction fails and must rollback
- This is also known as the "Dirty Read"

Incorrect Summary

- This occurs when two or more transactions:
  - Access the same data item
  - Are executed concurrently
  - Are interleaved
  - One transaction is calculating an aggregate summary on attributes while another transaction is updating those attributes

Schedules

- When transactions are executing concurrently in an interleaved fashion, the order of execution of operations is called the *schedule*
  - Schedule S of n transactions T_1, T_2, …, T_3 is the ordering of the operations within those transactions
- Operations from different transactions can be interleaved
  - The operations from each transaction T_i must appear in the same order in S, that they do in T_i
- A schedule S is generally written:
  - S_a: $O_i(X)$; $O_i(Y)$; $O_j(X)$; …; $O_n(Y)$;

- Where $O_i(X)$ indicates either a read or write operation executed by a transaction T_i on a data item X
- And $O_n(Y)$ indicates either a read or write operation executed by a transation T_n on a data item Y

- A shorthand notation can be used for each operation type
  - b → BEGIN_TRANSACTION
  - r → Read Item
  - w → Write Item
  - e → END_TRANSACTION
  - c → Commit
  - a → Abort (Rollback)

Schedule Conflicts

- Two operations in a schedule are said to *conflict* if:
  1. They belong to *different transactions*
  2. They access the *same item X*
  3. And *at least one* of the operations is a write(X)
- Intuitively, two operations are conflicting if changing their order can result in a different outcome
  - Or cause one of the concurrency issues we've already discussed

Serial Schedules

- In a *serial schedule* the operations of each transaction are executed consecutively, without any interleaved operations
- Formally, a schedule S is *serial* if, for every transaction T participating in the schedule, all operations of T are executed consecutively
  - Otherwise the schedule is called nonserial
- In a serial schedule, only one transaction is active at a time
  - The commit (or abort) of the active transaction initiates execution of the next transaction
- An assumption that can be made is: *every serial schedule is correct*
  - All transactions should be independent (**I**solation)
  - Each transaction is assumed to be correct if executed on its own (**C**onsistency preservation)
  - Hence, the ordering of transactions in a serial schedule does not matter
    * Once every transaction is executed from beginning to end

Issues with Serial Schedules

- Thay is all great...but:
  - Serial schedules limit concurrency
  - If a transaction is waiting for an operation to complete it is not possible to switch processing to another transaction
  - If a transaction T is very long, all other transactions must wait for T to complete its operations before they can begin
- Hence, serial schedules are considered unacceptable in practice

Serializability

- However, if it can be determined which other schedules are *equivalent* to a serial schedule, those schedules can be allowed to occur
- How is equivalence measured?
  - Result equivalence
  - Conflict Equivalence
- If a nonserial schedule meets these criteria, it is said to be *serializble*

Result Equivalence

- This is the simplest notion of equivalence
- Two schedules are said to be *result equivalent* if they produce the same final state of the DB
- However two schedules could coincidentally produce the same result

Conflict Equivalence

- Two schedules are said to be *conflict equivalent* if the order of any two conflicting operations is the same in both schedules
- Reminder: two operations is a schedule are said to *conflict* if:
  1. They belong to *different transactions*
  2. They access the *same item X*
  3. And *at least one* of the operations is a write(X)
- If two conflicting operations are applied in different orders in two schedules, the effect can be different on the DB
  - Hence, the schedules are not conflict equivalent) If read and write operations occur in the order
  - r_1(X), w_2(X) in schedule S_1

  - $r_2(X)$, $w_2(X)$ in schedule S_2
  - The value read by $r_1(X)$ may be different in the two schedules as it may have been updated by the write
- Similarly, if two write operations occur in the order:
  - w1(X), w2(X) in S1
  - w2(X), w1(X) in S2
  - Then the next r(X) operation in the two schedules will read potentially different values
  - If these are the last two operations in the schedules, the final value of item X in the DB will be different

Serializability

- Being able to say that a schedule is serializable, is the same as saying it is correct
  - All serial schedules are correct
  - This is equivalent to a serial schedule
  - Hence, this is also correct
- Serializable schedules give the benefits of concurrent execution without giving up correctness
- Most DBMS systems will have a set of protocols which:
  - Must be followed by every transaction
  - Are enforced by the concurrency control subsystem
  - Ensure the serializability of all schedules in which the transactions participate
- Concurrency Control Protocols

Concurrency Control Protocols

- Locking Protocols
  - Data items are locked to prevent multiple transactions from accessing them concurrently
- Timestamps
  - A unique identifier is generated for each transaction based upon transaction start time
- Optimistic Protocols
  - Based upon validation of the transaction after it executes its operations

Locking

- A lock is a variable associated with a data item
  - Describes the status of the data item with respect to operations that can be applied to it
  - Data items may be at a variety of granularities, e.g. DB, table, tuple, attribute, etc.
- Locks are used to synchronise access by concurrency transactions
- There are two main types of lock
  - Binary Lock
  - Read/Write Lock

Binary Lock

- A binary lock cab have only two states (or values)
  - Locked and unlocked
- Two operations are used in binary locking
  - lock_item
  - unlock_item
- Each transaction locks the item being using it, and then unlocks it when finished
- A transaction which wants to access a data item requirests to lock the item
  - If the tem is unlocked, the transaction locks it and can use it
  - If the item is already locked, then the transaction must wait until it is unlocked
- Binary locking is rarely used, as it is too restrictive
  - At most, one transaction can access each item at a time
  - Several transactions should be allowed access concurrently if they only need read access

Read/Write Lock

- If multiple transactions want to read an item, then they can access the item concurrently
  - Read operations are not conflicting
- However, if a transaction is to write an item, then it must have exclusive access

- The read/write lock implements this form of locking
  - It is called a multiple mode lock
- There are three locking operations used in read/write locks
  - read_lock
  - write_lock
  - unlock
- A *read_locked* item is also called *share_locked*, as other transactions are allowed to read it
- A *write_locked* item is also called *exclusive_locked* as a single transaction has access to it
- The following rules must be enforced
  1. A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T
  2. A transaction T must issue the operation write_lock(X) before any write_item(X) operation is performed in T
  3. A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T
  4. A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X

Lock Conversion

- Some DBMS allow the conversion of a lock from one state to another
- A transaction T can issue a read_lock(X) and then later *upgrade* the lock by issuing a write_lock(X)
  - If T is the only transaction holding a read lock X at the time is issues the write_lock(X) operation, the lock can be upgraded
  - Otherwise, the transaction must wait
- It is also possible for a transaction T to issue a write_lock(X) and then later to *downgrade* the lock by issuing a read_lock(X) operation

Two-Phase Locking

- Two-Phase Locking is an additional protocol
  - Concerns the positioning of locking and unlocking in every transaction
  - Used to guarantee serializability

- A transaction is said to follow two-pase locking if all lock operations (read_lock, write_lock) precede the first unlock operation in that transaction
- Two-Phase Locking transactions are divided into two phases:
  - Expanding Phase
    * During which new locks on items can be acquired but none can be released
  - Shrinking phase
    * During which existing locks can be released but no new locks can be acquired
- If lock conversion is allowed
  - Upgrading of locks (from read-locked to write-locked) must be done during the expanding phase
  - Downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase
- It has been proven that:
  - If every transaction in a schedule follocks the two-phase locking protocol, the schedule is uaranteed to be *serializable*
- This removes the need to test each schedule produced for serializability
  - However, this comes at a cost

Limitations

- Can limit the concurrency that can occur in a schedule
- A transaction may not be able to release an item X after it is finished using it, if the transaction must lock an additional item Y at a later point
  - Conversely, a transaction may have to lock the additional item Y before it needs it so that it can release X
  - Hence, X must remain locked unit all items and the transaction needs to readور write have been locked, only then can X be released
  - Meanwhile, another transaction seeking to access X may be forced to wait

Problems with Locking

- The use of locking in schedules can cause two additional problems
  - Deadlock
  - Starvation

Deadlock

- Deadlock occurs when
  - Each transaction T_i in a set of two or more transactions is waiting for some item that is locked by some other transaction in the set
  - Hence, each transaction in the set is waiting for one of the other transactions in the set to release the lock on an item
  - But because the other transaction is also waiting, it will never release the lock

Deadlock Prevention

- Ued to decide what to do with a transaction involved in a possible deadlock situation:
  - Should it be blocked and made wait?
  - Should it be aborted?
  - Should the transaction prempt another transaction and cause it to abort?
- No waiting
- Wait-die
- Would-wait
- Cautious Waiting
- Typically use the concept of transaction timestamp
  - Unique identifier assigned to each transaction
- Timestamps are based on the order in which transactions are started
  - Hence, if transactions T1 starts before transactions T2, then TS(T1) < TS(T2)

No Waiting

- In the *no wait* approach, if a transaction is unable to obtain a lock, it is immediately aborted
- It is then restarted after a certain time delay without checking whether a deadlock will actually occur or not
- In this case, no transaction ever waits, so no deadlock will occur
  - However, this scheme can cause transactions to abort and restart endlessly

Wait-die

- Suppose that transaction $T_{!}$ tries to lock an item X but is not able to because X is locked by some other transaction T_b
- The rules followed by *Wait-die* are:
  - If T_a is older than T_b then T_a is allowed to wait
  - Otherwise, if T_a is younger than T_b then abort T_a and restart it later with the same timestamp
  - So T_a either waits or dies

Wound-wait

- Suppose that transaction T_a tries to lock an item X but is not able to because X is locked by some other transaction T_b
- The rules followed by *Wound-wait* are:
  - T_a is older than T_b then abort T_b and restart it later with the same timestamp
  - Otherwise, if T_a is younger than T_b then T_a is allowed to wait
  - So T_a either wounds T_b or waits

Wait-die and Would-wait

- Of the two transactions that may be involved in a deadlock, both these approachs abort the transaction that started later
  - Assumption is that this will waste less processing
- However, both may cause some transactions to be aborted and restarted needlessly
  - Even though those transactions may never actually cause a deadlock

Cautious Waiting

- Propsed to try to reduce the number of needless aborts
- Suppose that transaction T_a tries to lock an item X but is not able to do so because X is locked by some other transaction T_b
- The cautious waiting rules are:
  - If T_b is not blocked (not waiting for some other locked item), then T_a is blocked and allowed to wait
  - Otherwise abort T_a

Deadlock Detection

- Deadlock detection is used to check is a state of deadlock actually exists
  - Attractive is there is little interference among transactions, I.e. If different transactions rarely access the same items at the same time
  - However, if transactions are long and each transaction uses many items, it may be better to use a deadlock prevention procotol
- A simple way to detect deadlock is for the system to construct and waiting a *wait-for graph*
  - One node is created in the wait-for graph for each transaction that is currently executing
  - Whenever a transaction T_a is waiting to lock an item X that is currently locked by a transaction T_b, a directed edge ($T_a \rightarrow T_b$) is created in the wait-for graph
  - When T_b releases the lock(s) on the items that T_a is waiting for, the directed edge is dropped from the wait-for graph
- There is a state of deadlock if and only if the wait-for graph has a loop
- Once detected, deadlock must be resolved by aborting and rolling back one of the transactions
- Choosing which transaction to abort is known as *victim selection*
  - Avoid selecting transactions that have run for a long time or have performed many updates
  - Instead, select transactions that have not made many changes (younger transactions)

Starvation

- Another problem that may occur when using locking is *starvation*
  - When a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally
- Can occur if the waiting scheme for locked items gives priority to some transactions over others
- Can also occur because of victim selection
  - If the algorithm repeatedly selects the same victim transaction, causing it to abort and never finish

Starvation Solution

- First-come-first-served queue

    - Transactions are allowed to lock an item in the order in which they originally requested the lock
- Waiting list
    - Allows some transactions to have priority over others, but increases the priority of a transaction the longer it waits, until it eventually gets the highest priority and proeceeds
- Victim priority
    - Assigns higher priorities to transactions that have been aborted multiple times

Timestamp Ordering

- As already discussed, timestamp values are assigned to transactions in the order they are submitted to the system
    - So a timestamb can be thought of as the transaction start time
- There are concurrency ontrol techniques based pureply on timestamp ordering which do not use locking
    - Hence, deadlocks cannot occur
- For each datat item accessed by conflicting operations in the schedule, the order in which the item is accessed must not violate the timestamp ordering
- This produces serializable schedules which are equivalent to the serial schedule
    - As all conflicting operations are conducted in the same order that they could be in the serial schedule
- In order to enforce this, the DBMS keeps two timestamp values for each data item:
    - read_TS(X) - This is equal to the timestamp of the most recently started (youngest) transaction that has successfully read item X
    - read_TS(X) - This is equal to the timestamp of the most recently started (youngest) transaction that has successfully writen item X
- Whenevr a transaction T issues a write_item(X) operation, the following is checked:
    - If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then abort and rollback T and reject the operation
        * This should be done because some younger transaction with a timestamp greats than TS(T) - and hence after T in the timestamp ordering - has already read or written the value of item X before T has a chance to write X, thus violating the timestamp ordering

  - If tead_TS(X) $\leq$ TS(T) and if write_TS(X) $\leq$ TS(T), then execute the write_item(X) operation of T and set write_TS(X) to TS(T)

- Whenever a transaction T issues a read_item(X) operation, the following is checked
  - If write_TS(X) $>$ TS(T), then abort and rollback T and reject the operation
    * This should be done because some younger transaction with timestamp greater than TS(T) - and hence after T in the timestamp ordering - has already written the value of item X before T has a chance to read X
  - If write_TS(X) $\leq$ TS(T), then execute the read_item(X) operation of T and set read_TS(X) equal to the larger of TS(T) and the current read_TS(X)