An abstract digital graphic on the left side of the slide. It features several blue, three-dimensional cubes or rectangular blocks arranged in a staggered, isometric pattern. The surfaces of these blocks are covered with a fine grid of small, glowing blue dots, resembling binary code or data points. Interspersed among the blocks are several small, bright light sources in blue, green, and red, some of which appear to be emitting thin, glowing lines. The overall color scheme is dominated by deep blues and teals, with the lights providing points of contrast.

Database Management Systems (BCS403) - 2023-24 - Module 5

Dr. Narender M
Department of CS&E
The National Institute of Engineering

Topics

Concurrency Control in Databases

- Two-phase locking techniques for Concurrency control
- Concurrency control based on Timestamp ordering
- Multiversion Concurrency control techniques
- Validation Concurrency control techniques
- Granularity of Data items and Multiple Granularity Locking

NoSQL Databases and Big Data Storage Systems

- Introduction to NOSQL Systems
- The CAP Theorem
- Document-Based NOSQL Systems and MongoDB
- NOSQL Key/Value Stores
- Column-Based or Wide Column NOSQL Systems

Chapter 1: Concurrency Control in Databases



Two-phase locking techniques for Concurrency control

Types of Locks and System Lock Tables

- A lock is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it.
- Types of locks
 - binary locks
 - shared/exclusive locks
 - certify lock


Two-phase locking techniques for Concurrency control

Binary Locks

- A binary lock can have two states or values: locked and unlocked (or 1 and 0, for simplicity).
- A distinct lock is associated with each database item X.
- If the value of the lock on X is 1, item X cannot be accessed by a database operation that requests the item.
- If the value of the lock on X is 0, the item can be accessed when requested, and the lock value is changed to 1.

Two-phase locking techniques for Concurrency control

- Two operations, `lock_item` and `unlock_item`, are used with binary locking.
- A transaction requests access to an item `X` by first issuing a `lock_item(X)` operation.
- If $LOCK(X) = 1$, the transaction is forced to wait. If $LOCK(X) = 0$, it is set to 1 and the transaction is allowed to access item `X`.
- When the transaction is through using the item, it issues an `unlock_item(X)` operation, which sets $LOCK(X)$ back to 0 so that `X` may be accessed by other transactions.



Two-phase locking techniques for Concurrency control

lock_item(X):

```
B:  if LOCK( $X$ ) = 0                (*item is unlocked*)
      then LOCK( $X$ )  $\leftarrow$  1    (*lock the item*)
    else
      begin
        wait (until LOCK( $X$ ) = 0
              and the lock manager wakes up the transaction);
        go to B
      end;
```

unlock_item(X):

```
LOCK( $X$ )  $\leftarrow$  0;                (* unlock the item *)
if any transactions are waiting
  then wakeup one of the waiting transactions;
```

Two-phase locking techniques for Concurrency control

- Each lock can be a record with three fields: <Data_item_name, LOCK, Locking_transaction> plus a queue for transactions that are waiting to access the item.
- The system needs to maintain only these records for the items that are currently locked in a lock table, which could be organized as a hash file on the item name.
- Items not in the lock table are unlocked. The DBMS has a lock manager subsystem to keep track of and control access to locks.

Two-phase locking techniques for Concurrency control

- Every transaction must obey the following rules:
 1. A transaction T must issue the operation `lock_item(X)` before any `read_item(X)` or `write_item(X)` operations are performed in T.
 2. A transaction T must issue the operation `unlock_item(X)` after all `read_item(X)` and `write_item(X)` operations are completed in T.
 3. A transaction T will not issue a `lock_item(X)` operation if it already holds the lock on item X.
 4. A transaction T will not issue an `unlock_item(X)` operation unless it already holds the lock on item X.

Two-phase locking techniques for Concurrency control

Shared/Exclusive (or Read/Write) Locks.

- The preceding binary locking scheme is too restrictive for database items because at most one transaction can hold a lock on a given item.
- We should allow several transactions to access the same item X if they all access X for reading purposes only.
- This is because read operations on the same item by different transactions are not conflicting.
- if a transaction is to write an item X, it must have exclusive access to X. For this purpose, a different type of lock, called a multiple-mode lock, is used.

Two-phase locking techniques for Concurrency control

- There are three locking operations: `read_lock(X)`, `write_lock(X)`, and `unlock(X)`.
- A lock associated with an item `X`, `LOCK(X)`, now has three possible states: read-locked, write-locked, or unlocked.
- A read-locked item is also called share-locked because other transactions are allowed to read the item.
- A write-locked item is called exclusive-locked because a single transaction exclusively holds the lock on the item.

Two-phase locking techniques for Concurrency control

- Each record in the lock table will have four fields:
<Data_item_name, LOCK, No_of_reads, Locking_transaction(s)>.
- If LOCK(X) = write-locked, the value of locking_transaction(s) is a single transaction that holds the exclusive (write) lock on X.
- If LOCK(X)=read-locked, the value of locking transaction(s) is a list of one or more transactions that hold the shared (read) lock on X.

Two-phase locking techniques for Concurrency control

- When we use the shared/exclusive locking scheme, the system must enforce the following rules:
 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.

Two-phase locking techniques for Concurrency control

4. A transaction T will not issue a `read_lock(X)` operation if it already holds a read (shared) lock or a write (exclusive) lock on item X.
5. A transaction T will not issue a `write_lock(X)` operation if it already holds a read (shared) lock or write (exclusive) lock on item X.
6. 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.

Two-phase locking techniques for Concurrency control

Conversion (Upgrading, Downgrading) of Locks

- A transaction that already holds a lock on item X is allowed under certain conditions to convert the lock from one locked state to another.
- A transaction T has a `read_lock(X)` and then upgrades the lock by issuing a `write_lock(X)` operation.
- A transaction T has a `write_lock(X)` and then downgrades the lock by issuing a `read_lock(X)` operation.
- Using binary locks or read/write locks in transactions, as described earlier, does not guarantee serializability of schedules on its own.
- An additional protocol concerning the positioning of locking and unlocking operations in every transaction.

Two-phase locking techniques for Concurrency control

Guaranteeing Serializability by Two-Phase Locking

- A transaction is said to follow the two-phase locking protocol if all locking operations (read_lock, write_lock) precede the first unlock operation in the transaction.
- A transaction can be divided into two phases:
 - an expanding or growing (first) phase, during which new locks on items can be acquired but none can be released
 - a shrinking (second) phase, during which existing locks can be released but no new locks can be acquired.

Two-phase locking techniques for Concurrency control

- If lock conversion is allowed, then upgrading of locks must be done during the expanding phase, and downgrading of locks must be done in the shrinking phase.

(a)

| T_1 | T_2 |
|--|--|
| <pre>read_lock(Y); read_item(Y); unlock(Y); write_lock(X); read_item(X); X := X + Y; write_item(X); unlock(X);</pre> | <pre>read_lock(X); read_item(X); unlock(X); write_lock(Y); read_item(Y); Y := X + Y; write_item(Y); unlock(Y);</pre> |

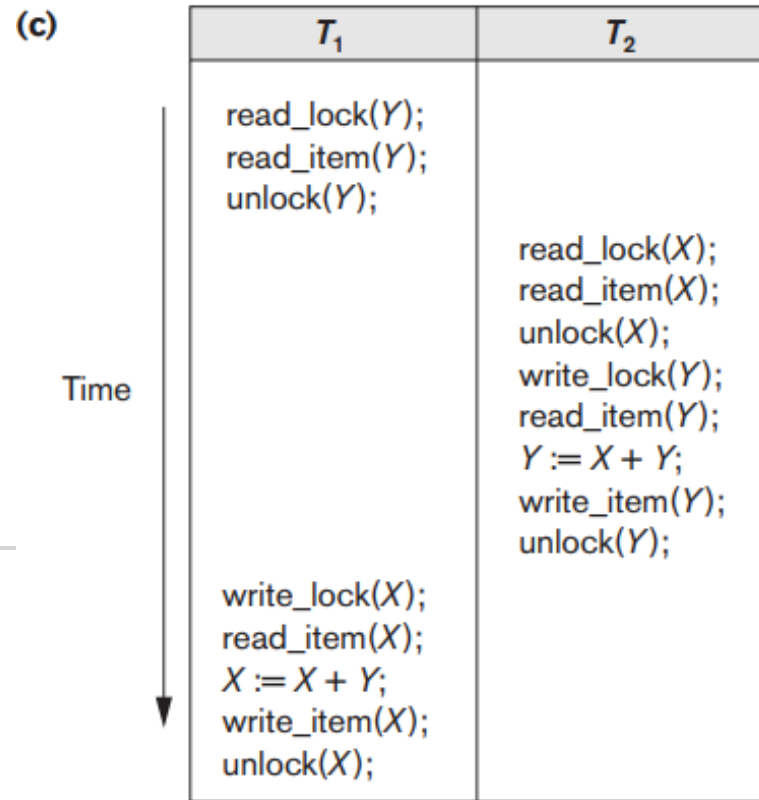
(b)

Initial values: $X=20$, $Y=30$

Result serial schedule T_1
followed by T_2 : $X=50$, $Y=80$

Result of serial schedule T_2
followed by T_1 : $X=70$, $Y=50$

Two-phase locking techniques for Concurrency control



Result of schedule S:
 $X=50, Y=50$
(nonserializable)

Figure 21.3

Transactions that do not obey two-phase locking. (a) Two transactions T_1 and T_2 . (b) Results of possible serial schedules of T_1 and T_2 . (c) A nonserializable schedule S that uses locks.

Two-phase locking techniques for Concurrency control

- If every transaction in a schedule follows the two-phase locking protocol, the schedule is guaranteed to be serializable.

Basic, Conservative, Strict, and Rigorous Two-Phase Locking

- There are a number of variations of two-phase locking (2PL). The technique just described is known as basic 2PL.
- A variation known as conservative 2PL (or static 2PL) requires a transaction to lock all the items it accesses before the transaction begins execution, by predeclaring its read-set and write-set.

Two-phase locking techniques for Concurrency control

- If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking.
- Conservative 2PL is a deadlock-free protocol.
- It is difficult to use in practice because of the need to predeclare the read-set and write-set, which is not possible in some situations.
- The most popular variation of 2PL is strict 2PL, which guarantees strict schedules.
- A transaction T does not release any of its exclusive (write) locks until after it commits or aborts.

Two-phase locking techniques for Concurrency control

- No other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability.
- Strict 2PL is not deadlock-free.
- A more restrictive variation of strict 2PL is rigorous 2PL, which also guarantees strict schedules.
- In this variation, a transaction T does not release any of its locks (exclusive or shared) until after it commits or aborts.

Two-phase locking techniques for Concurrency control

- The difference between strict and rigorous 2PL: the former holds write-locks until it commits, whereas the latter holds all locks (read and write).
- Also, the difference between conservative and rigorous 2PL is that the former must lock all its items before it starts, so once the transaction starts it is in its shrinking phase; the latter does not unlock any of its items until after it terminates (by committing or aborting), so the transaction is in its expanding phase until it ends.
- Usually, the concurrency control subsystem itself is responsible for generating the read_lock and write_lock requests.

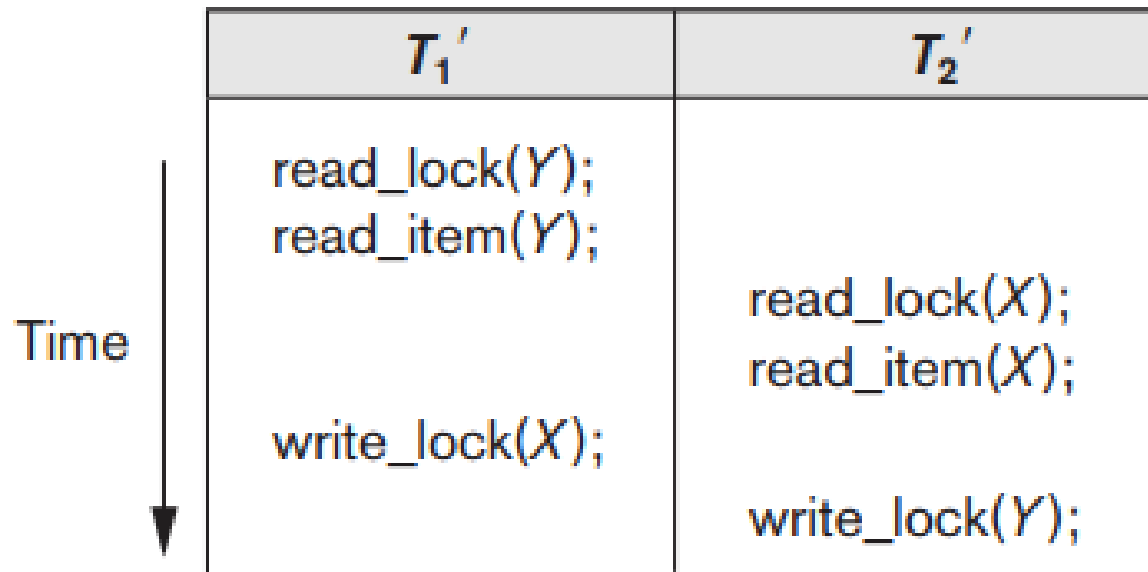
Two-phase locking techniques for Concurrency control

Dealing with Deadlock and Starvation

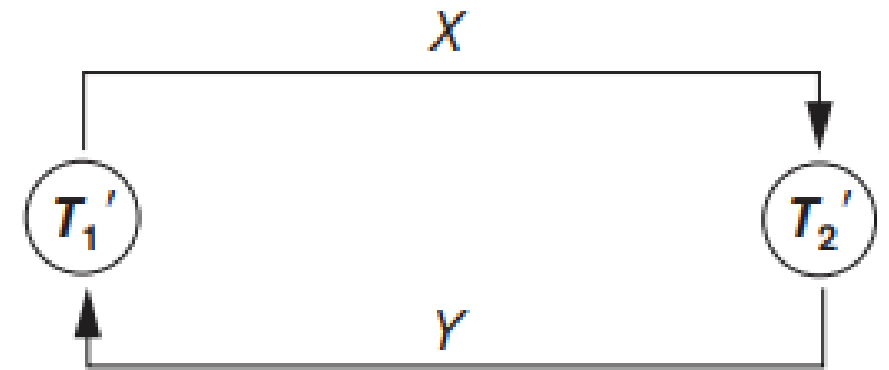
- Deadlock occurs when each transaction T in a set of two or more transactions is waiting for some item that is locked by some other transaction T' in the set.
- Hence, each transaction in the set is in a waiting queue, waiting for one of the other transactions in the set to release the lock on an item.
- But because the other transaction is also waiting, it will never release the lock.

Two-phase locking techniques for Concurrency control

(a)



(b)



Two-phase locking techniques for Concurrency control

Deadlock Prevention Protocols

- One deadlock prevention protocol, which is used in conservative two-phase locking, requires that every transaction lock all the items it needs in advance.
- If any of the items cannot be obtained, none of the items are locked.
- The transaction waits and then tries again to lock all the items it needs. This reduces concurrency.

Two-phase locking techniques for Concurrency control

- A second protocol involves ordering all the items in the database and making sure that a transaction that needs several items will lock them according to that order.
- This also limits concurrency.
- Deadlock prevention scheme: Transaction timestamp $TS(T')$ is a unique identifier assigned to each transaction.
- The timestamps are typically based on the order in which transactions are started; hence, if transaction $T1$ starts before transaction $T2$, then $TS(T1) < TS(T2)$.
- Notice that the older transaction (which starts first) has the smaller timestamp value. Two schemes that prevent deadlock are called wait-die and wound-wait.

Two-phase locking techniques for Concurrency control

- The rules followed by these schemes are:
- Wait-die.
 - If $TS(T_i) < TS(T_j)$, then (T_i older than T_j) T_i is allowed to wait; otherwise (T_i younger than T_j) abort T_i (T_i dies) and restart it later with the same timestamp.
- Wound-wait.
 - If $TS(T_i) < TS(T_j)$, then (T_i older than T_j) abort T_j (T_i wounds T_j) and restart it later with the same timestamp; otherwise (T_i younger than T_j) T_i is allowed to wait.

Two-phase locking techniques for Concurrency control

Another deadlock prevention: No waiting algorithm

- If a transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking whether a deadlock will occur or not.
- No transaction ever waits, so no deadlock will occur. This scheme can cause transactions to abort and restart needlessly.
- The cautious waiting algorithm was proposed to try to reduce the number of needless aborts/restarts.

Two-phase locking techniques for Concurrency control

- Suppose that transaction T_i tries to lock an item X but is not able to do so because X is locked by some other transaction T_j with a conflicting lock.
- The cautious waiting rule is as follows:
 - If T_j is not blocked (not waiting for some other locked item), then T_i is blocked and allowed to wait; otherwise abort T_i .

Deadlock Detection

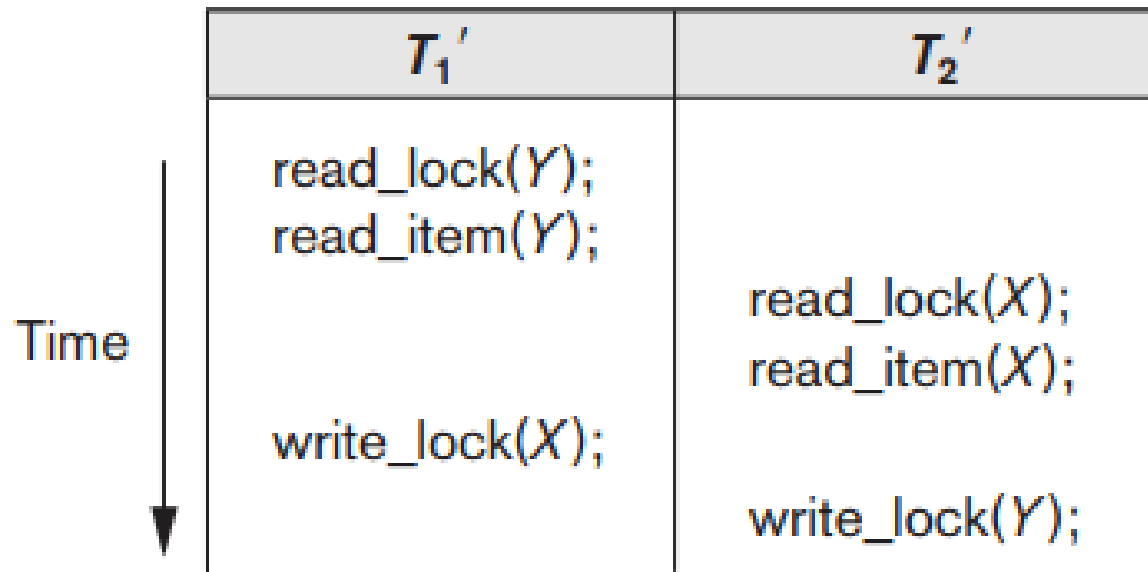
- An alternative approach to dealing with deadlock is deadlock detection, where the system checks if a state of deadlock exists.

Two-phase locking techniques for Concurrency control

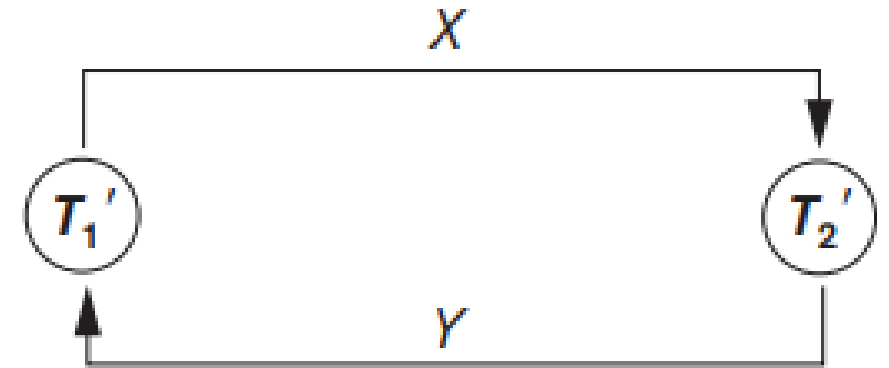
- A simple way to detect a state of deadlock is for the system to construct and maintain a wait-for graph.
- One node is created in the wait-for graph for each transaction that is currently executing.
- Whenever a transaction T_i is waiting to lock an item X that is currently locked by a transaction T_j , a directed edge ($T_i \rightarrow T_j$) is created in the wait-for graph.
- When T_j releases the lock(s) on the items that T_i was waiting for, the directed edge is dropped from the wait-for graph.
- We have a state of deadlock if and only if the wait-for graph has a cycle.

Two-phase locking techniques for Concurrency control

(a)



(b)



Two-phase locking techniques for Concurrency control

- Choosing which transactions to abort is known as victim selection.
- The algorithm for victim selection should generally avoid selecting transactions that have been running for a long time and that have performed many updates.
- It should try instead to select transactions that have not made many changes (younger transactions).
- **Timeouts.** If a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it—regardless of whether a deadlock exists.

Two-phase locking techniques for Concurrency control

- **Starvation.** Another problem that may occur is starvation, which occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally.
- This may occur if the waiting scheme for locked items is unfair in that it gives priority to some transactions over others.
- Solution:
 - Using a first-come-first-served queue.
 - Increase the priority of a transaction the longer it waits, until it eventually gets the highest priority and proceeds.
- Starvation can also occur because of victim selection if the algorithm selects the same transaction as victim repeatedly.

Two-phase locking techniques for Concurrency control

- Use higher priorities for transactions that have been aborted multiple times to avoid this problem.
- A different approach to concurrency control involves using transaction timestamps to order transaction execution for an equivalent serial schedule.

Concurrency control based on Timestamp ordering

Timestamps

- A timestamp is a unique identifier created by the DBMS to identify a transaction.
- Timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the transaction start time.
- Concurrency control techniques based on timestamp ordering do not use locks; hence, deadlocks cannot occur.

Concurrency control based on Timestamp ordering

- Timestamps generation - One possibility is to use a counter that is incremented each time its value is assigned to a transaction.
- The transaction timestamps are numbered 1, 2, 3, ... in this scheme.
- A computer counter has a finite maximum value, so the system must periodically reset the counter to zero when no transactions are executing for some short period of time.
- Another way to implement timestamps is to use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

Concurrency control based on Timestamp ordering

The Timestamp Ordering Algorithm for Concurrency Control

- To enforce the equivalent serial order on the transactions based on their timestamps.
- A schedule in which the transactions participate is then serializable, and the only equivalent serial schedule permitted has the transactions in order of their timestamp values.
- The algorithm allows interleaving of transaction operations, but it must ensure that for each pair of conflicting operations in the schedule, the order in which the item is accessed must follow the timestamp order.

Concurrency control based on Timestamp ordering

The algorithm associates with each database item X two timestamp (TS) values:

1. $\text{read_TS}(X)$. The read timestamp of item X is the largest timestamp among all the timestamps of transactions that have successfully read item X —that is, $\text{read_TS}(X) = \text{TS}(T)$, where T is the youngest transaction that has read X successfully.
2. $\text{write_TS}(X)$. The write timestamp of item X is the largest of all the timestamps of transactions that have successfully written item X —that is, $\text{write_TS}(X) = \text{TS}(T)$, where T is the youngest transaction that has written X successfully.

Concurrency control based on Timestamp ordering

Basic Timestamp Ordering

- Whenever some transaction T tries to issue a $\text{read_item}(X)$ or a $\text{write_item}(X)$ operation, the basic TO algorithm compares the timestamp of T with $\text{read_TS}(X)$ and $\text{write_TS}(X)$ to ensure that the timestamp order of transaction execution is not violated.
- If this order is violated, then transaction T is aborted and resubmitted to the system as a new transaction with a new timestamp.
- Cascading rollback is one of the problems associated with basic TO.

Concurrency control based on Timestamp ordering

- Check whether conflicting operations violate the timestamp ordering in the following two cases:

1. Whenever a transaction T issues a $\text{write_item}(X)$ operation, the following check is performed:

a. If $\text{read_TS}(X) > \text{TS}(T)$ or if $\text{write_TS}(X) > \text{TS}(T)$, then abort and roll back T and reject the operation. Some younger transaction with a timestamp greater than $\text{TS}(T)$ has already read or written the value of item X before T had a chance to write X , thus violating ordering.

b. If the condition in part (a) does not occur, then execute the $\text{write_item}(X)$ operation of T and set $\text{write_TS}(X)$ to $\text{TS}(T)$.

Concurrency control based on Timestamp ordering

2. Whenever a transaction T issues a $\text{read_item}(X)$ operation, the following check is performed:
 - a. If $\text{write_TS}(X) > \text{TS}(T)$, then abort and roll back T and reject the operation. Some younger transaction with timestamp greater than $\text{TS}(T)$ has already written the value of item X before T had a chance to read X .
 - b. If $\text{write_TS}(X) \leq \text{TS}(T)$, then execute the $\text{read_item}(X)$ operation of T and set $\text{read_TS}(X)$ to the larger of $\text{TS}(T)$ and the current $\text{read_TS}(X)$.

Concurrency control based on Timestamp ordering

Strict Timestamp Ordering (TO)

- A variation of basic TO called strict TO ensures that the schedules are both strict (for easy recoverability) and (conflict) serializable.
- A transaction T issues a $\text{read_item}(X)$ or $\text{write_item}(X)$ such that $\text{TS}(T) > \text{write_TS}(X)$ has its read or write operation delayed until the transaction T' that wrote the value of X (hence $\text{TS}(T') = \text{write_TS}(X)$) has committed or aborted.

Concurrency control based on Timestamp ordering

Thomas's Write Rule

- A modification of the basic TO algorithm, known as Thomas's write rule, does not enforce conflict serializability, but it rejects fewer write operations.
 1. If $\text{read_TS}(X) > \text{TS}(T)$, then abort and roll back T and reject the operation.
 2. If $\text{write_TS}(X) > \text{TS}(T)$, then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than $\text{TS}(T)$ —and hence after T in the timestamp ordering—has already written the value of X . We must ignore the $\text{write_item}(X)$ operation of T because it is already outdated and obsolete.

Concurrency control based on Timestamp ordering

3. If neither the condition in part (1) nor the condition in part (2) occurs, then execute the `write_item(X)` operation of `T` and set `write_TS(X)` to `TS(T)`.

Multiversion Concurrency control techniques

-
- These protocols for concurrency control keep copies of the old values of a data item when the item is updated (written); they are known as multiversion concurrency control.
 - Some read operations that would be rejected in other techniques can still be accepted by reading an older version of the item to maintain serializability.
 - More storage is needed to maintain multiple versions of the database items.

Multiversion Concurrency control techniques

Multiversion Technique Based on Timestamp Ordering

- Several versions X_1, X_2, \dots, X_k of each data item X are maintained.
- For each version, the value of version X_i and the following two timestamps associated with version X_i are kept:
 1. $read_TS(X_i)$. The read timestamp of X_i is the largest of all the timestamps of transactions that have successfully read version X_i .
 2. $write_TS(X_i)$. The write timestamp of X_i is the timestamp of the transaction that wrote the value of version X_i .

Multiversion Concurrency control techniques

-
- Whenever a transaction T is allowed to execute a $\text{write_item}(X)$ operation, a new version X_{k+1} of item X is created, with both the $\text{write_TS}(X_{k+1})$ and the $\text{read_TS}(X_{k+1})$ set to $\text{TS}(T)$.
 - When a transaction T is allowed to read the value of version X_i , the value of $\text{read_TS}(X_i)$ is set to the larger of the current $\text{read_TS}(X_i)$ and $\text{TS}(T)$.

Multiversion Concurrency control techniques

Multiversion Two-Phase Locking Using Certify Locks

- In this multiple-mode locking scheme, there are three locking modes for an item— read, write, and certify.
- Read Operation
 - A transaction reads the most recent version of a data item that was committed before the transaction started.
 - If a transaction T1 wants to read a data item X, it reads the version of X that was committed before T1 started.
- Write Operation
 - A transaction creates a new version of the data item without immediately overwriting the existing version.
 - The new version remains invisible to other transactions until the writing transaction commits.

Multiversion Concurrency control techniques

-
- Certify Phase
 - When a transaction is ready to commit, it enters the certify phase and acquires certify locks on all the data items it has modified.
 - A certify lock ensures that no other transaction can access the data item for reading or writing until it commits or aborts.
 - The transaction checks if any other transaction has read or written the data items it modified.
 - If a conflict is detected (e.g., another transaction has read the old version while the current transaction was writing), the transaction may need to abort to maintain serializability.

Validation Concurrency control techniques

- Concurrency issue checking is done before a database operation can be executed and is an overhead during transaction execution.
- In optimistic concurrency control techniques (validation or certification techniques) no checking is done while the transaction is executing.

Validation-Based (Optimistic) Concurrency Control

- In this scheme, updates in the transaction are not applied directly to the database items on disk until the transaction reaches its end and is validated.
- During transaction execution, all updates are applied to local copies of the data items that are kept for the transaction.

Validation Concurrency control techniques

- A validation phase checks whether any of the transaction's updates violate serializability.
- If it is not violated, the transaction is committed and the database is updated from the local copies; otherwise, the transaction is aborted and then restarted later.
- There are three phases for this concurrency control protocol:
 1. Read phase
 - A transaction can read values of committed data items from the database.
 - Updates are applied only to local copies (versions) of the data items kept in the transaction workspace.
 2. Validation phase
 - Checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.

Validation Concurrency control techniques

3. Write phase

- If the validation phase is successful, the transaction updates are applied to the database; otherwise, the updates are discarded, and the transaction is restarted.
- Idea is to do all the checks at once; hence, transaction execution proceeds with a minimum of overhead.
- Optimistic because they assume that little interference will occur and hence most transaction will be validated successfully.

Validation Concurrency control techniques

Concurrency Control Based on Snapshot Isolation

- Snapshot Isolation (SI) is a concurrency control mechanism used to ensure that transactions can execute without interfering with each other, while still providing a consistent view of the database.
- Snapshot
 - At the start of a transaction, the database system provides a snapshot of the data as it existed at a specific point in time.
 - This snapshot remains consistent for the duration of the transaction.

Validation Concurrency control techniques

- Versioning
 - The database maintains multiple versions of data items.
 - Each version corresponds to the state of a data item at the time a transaction modified it.
- Commit Protocol
 - Transactions modify data items in a private workspace and make their changes visible to other transactions only at commit time.

Working:

1. Starting a Transaction
 - When a transaction T starts, it gets a consistent snapshot of the database.
 - This snapshot includes all data items as they were at the start time of T .

Validation Concurrency control techniques

Read Operations

- Reads are directed to the snapshot.
- If T reads a data item X , it sees the version of X that was committed before T started.

Write Operations

- Writes create new versions of data items.
- These new versions are not visible to other transactions until T commits.

Validation Concurrency control techniques

Commit Protocol

- When T is ready to commit, it checks for write-write conflicts.
- A write-write conflict occurs if another transaction U has committed a write to a data item X that T has also modified, where U's commit timestamp is after T's start time.
- If no conflicts are detected, T commits its changes, making the new versions of the data items visible to other transactions.
- If a conflict is detected, T must abort and retry.
- Variations of snapshot isolation (SI) techniques, known as serializable snapshot isolation (SSI), have been proposed and implemented.

Granularity of Data items and Multiple Granularity Locking

-
- A database item could be chosen to be one of the following:
 - A database record
 - A field value of a database record
 - A disk block
 - A whole file
 - The whole database

Granularity Level Considerations for Locking

- The size of data items is often called the data item granularity.
- Fine granularity refers to small item sizes, whereas coarse granularity refers to large item sizes.

Granularity of Data items and Multiple Granularity Locking

-
- **Larger** the data item size is, the lower the degree of concurrency permitted.
 - **Smaller** the data item size is, the more the number of items in the database.
 - Because every item is associated with a lock, the system will have a larger number of active locks to be handled by the lock manager.
 - More lock and unlock operations will be performed, causing a higher overhead.
 - More storage space will be required for the lock table.

Granularity of Data items and Multiple Granularity Locking

-
- What is the best item size? The answer is that it depends on the types of transactions involved.

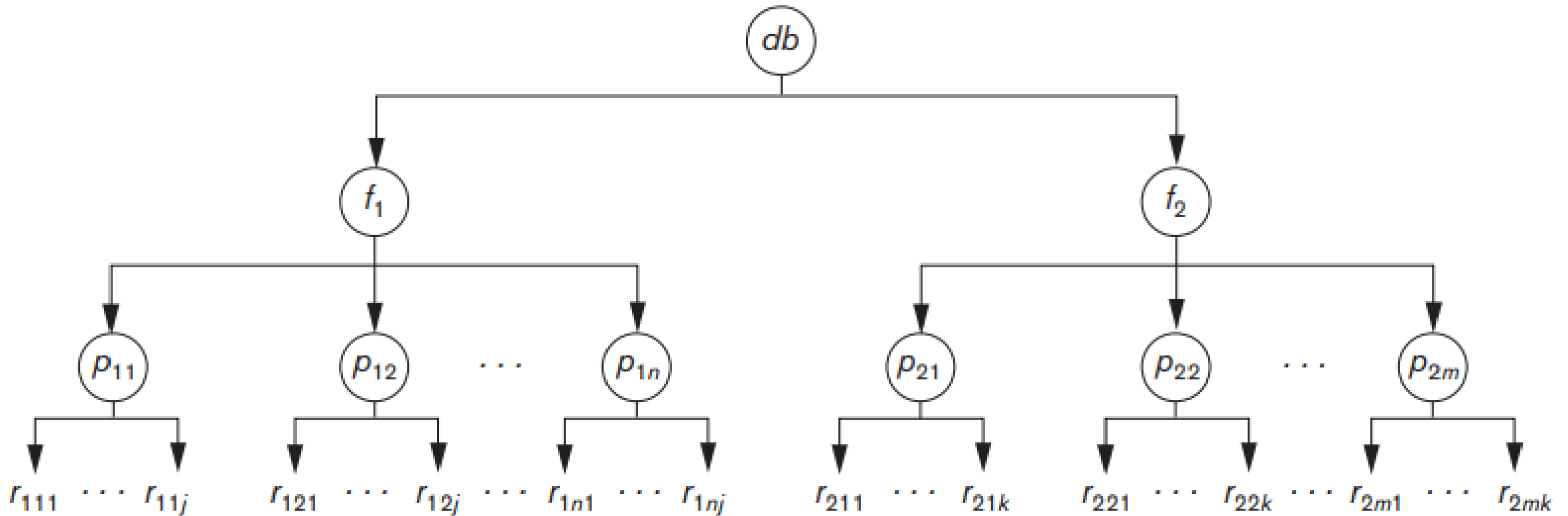
Multiple Granularity Level Locking

- Database system should support multiple levels of granularity, where the granularity level can be adjusted dynamically for various mixes of transactions.
- To make multiple granularity level locking practical, additional types of locks, called intention locks, are needed.

Granularity of Data items and Multiple Granularity Locking

-
- There are three types of intention locks:
 1. Intention-shared (IS) indicates that one or more shared locks will be requested on some descendant node(s).
 2. Intention-exclusive (IX) indicates that one or more exclusive locks will be requested on some descendant node(s).
 3. Shared-intention-exclusive (SIX) indicates that the current node is locked in shared mode but that one or more exclusive locks will be requested on some descendant node(s).

Granularity of Data items and Multiple Granularity Locking



Chapter 2: NoSQL Databases and Big Data Storage Systems

Introduction to NOSQL Systems

- NOSQL
 - Not only SQL
- Most NOSQL systems are distributed databases or distributed storage systems
 - Focus on semi-structured data storage, high performance, availability, data replication, and scalability

Introduction to NOSQL Systems

- NOSQL systems focus on storage of “big data”
- Typical applications that use NOSQL
 - Social media
 - Web links
 - User profiles
 - Marketing and sales
 - Posts and tweets
 - Road maps and spatial data
 - Email



Introduction to NOSQL Systems

- BigTable
 - Google's proprietary NOSQL system
 - Column-based or wide column store
- DynamoDB (Amazon)
 - Key-value data store
- Cassandra (Facebook)
 - Uses concepts from both key-value store and column-based systems



Introduction to NOSQL Systems

- MongoDB and CouchDB
 - Document stores
- Neo4J and GraphBase
 - Graph-based NOSQL systems
- OrientDB
 - Combines several concepts
- Database systems classified on the object model
 - Or native XML model



Introduction to NOSQL Systems

- NOSQL characteristics related to distributed databases and distributed systems
 - Scalability
 - Availability, replication, and eventual consistency
 - Replication models
 - Master-slave
 - Master-master
 - Sharding of files
 - High performance data access

Introduction to NOSQL Systems

- NOSQL characteristics related to data models and query languages
 - Schema not required
 - Less powerful query languages
 - Versioning



Introduction to NOSQL Systems

- Categories of NOSQL systems
 - Document-based NOSQL systems
 - NOSQL key-value stores
 - Column-based or wide column NOSQL systems
 - Graph-based NOSQL systems
 - Hybrid NOSQL systems
 - Object databases
 - XML databases



The CAP Theorem

- Various levels of consistency among replicated data items
 - Enforcing serializability the strongest form of consistency
 - High overhead – can reduce read/write operation performance
- CAP theorem
 - Consistency, availability, and partition tolerance
 - Not possible to guarantee all three simultaneously
 - In distributed system with data replication



The CAP Theorem

- Designer can choose two of three to guarantee
 - Weaker consistency level is often acceptable in NOSQL distributed data store
 - Guaranteeing availability and partition tolerance more important
 - Eventual consistency often adopted



Document- Based NOSQL Systems and MongoDB

- Document stores
 - Collections of similar documents
- Individual documents resemble complex objects or XML documents
 - Documents are self-describing
 - Can have different data elements
- Documents can be specified in various formats
 - XML
 - JSON



Document-Based NOSQL Systems and MongoDB

- Documents stored in binary JSON (BSON) format
- Individual documents stored in a collection
- Example command
 - First parameter specifies name of the collection
 - Collection options include limits on size and number of documents
- Each document in collection has unique ObjectId field called `_id`

```
db.createCollection("project", { capped : true, size : 1310720, max : 500 } )
```

Document-Based NOSQL Systems and MongoDB

- A collection does not have a schema
 - Structure of the data fields in documents chosen based on how documents will be accessed
 - User can choose normalized or denormalized design
- Document creation using insert operation
- Document deletion using remove operation

```
db.<collection_name>.insert(<document(s)>)
```

```
db.<collection_name>.remove(<condition>)
```

Document-Based NOSQL Systems and MongoDB

- Example of simple documents in MongoDB

- (a) Denormalized document design with embedded subdocuments
- (b) Embedded array of document references

(a) project document with an array of embedded workers:

```
{
  _id: "P1",
  Pname: "ProductX",
  Plocation: "Bellaire",
  Workers: [
    { Ename: "John Smith",
      Hours: 32.5
    },
    { Ename: "Joyce English",
      Hours: 20.0
    }
  ]
};
```

(b) project document with an embedded array of worker ids:

```
{
  _id: "P1",
  Pname: "ProductX",
  Plocation: "Bellaire",
  Workerids: [ "W1", "W2" ]
}

{ _id: "W1",
  Ename: "John Smith",
  Hours: 32.5
}

{ _id: "W2",
  Ename: "Joyce English",
  Hours: 20.0
}
```

Document-Based NOSQL Systems and MongoDB

- Example of simple documents in MongoDB
Normalized documents Inserting the documents into their collections

(c) normalized project and worker documents (not a fully normalized design for M:N relationships):

```
{
  _id:      "P1",
  Pname:    "ProductX",
  Plocation: "Bellaire"
}
{
  _id:      "W1",
  Ename:    "John Smith",
  ProjectId: "P1",
  Hours:    32.5
}
```

```
{
  _id:      "W2",
  Ename:    "Joyce English",
  ProjectId: "P1",
  Hours:    20.0
}
```

(d) inserting the documents in (c) into their collections "project" and "worker":

```
db.project.insert( { _id: "P1", Pname: "ProductX", Plocation: "Bellaire" } )
db.worker.insert( [ { _id: "W1", Ename: "John Smith", ProjectId: "P1", Hours: 32.5 },
                    { _id: "W2", Ename: "Joyce English", ProjectId: "P1",
                      Hours: 20.0 } ] )
```

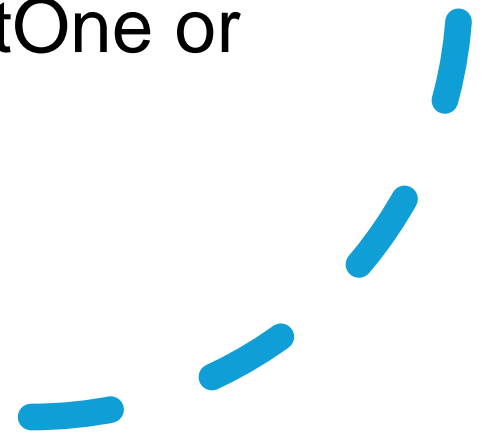
Document- Based NOSQL Systems and MongoDB

CRUD Operations in MongoDB

- CRUD operations in MongoDB refer to the basic operations you can perform on a database: Create, Read, Update, and Delete.

Create

- To insert data into a MongoDB collection, you use the insertOne or insertMany methods.



Document- Based NOSQL Systems and MongoDB

// Insert a single document

```
db.collection.insertOne({  
  name: "Alice",  
  age: 25,  
  address: "123 Apple St."
```

```
});
```

// Insert multiple documents

```
db.collection.insertMany([  
  { name: "Bob", age: 30, address: "456 Orange St." },  
  { name: "Charlie", age: 35, address: "789 Banana  
St." }  
]);
```

Document- Based NOSQL Systems and MongoDB

2. Read

- To read or retrieve data from a MongoDB collection, you use the find method.

// Find all documents

```
db.collection.find({});
```

// Find documents with a query

```
db.collection.find({ age: { $gt: 25 } });
```

// Find a single document

```
db.collection.findOne({ name: "Alice" });
```

Document- Based NOSQL Systems and MongoDB

3. Update

- To update data in a MongoDB collection, you use the `updateOne`, `updateMany`, or `replaceOne` methods.

```
// Update a single document
db.collection.updateOne(
  { name: "Alice" },
  { $set: { age: 26 } }
);
```



Document- Based NOSQL Systems and MongoDB

```
// Update multiple documents
db.collection.updateMany(
  { age: { $lt: 30 } },
  { $set: { status: "young" } }
);
// Replace a single document
db.collection.replaceOne(
  { name: "Alice" },
  { name: "Alice", age: 26, address: "123
Apple St.", status: "young" }
);
```

Document- Based NOSQL Systems and MongoDB

4. Delete

- To delete data from a MongoDB collection, you use the `deleteOne` or `deleteMany` methods.

// Delete a single document

```
db.collection.deleteOne({ name: "Alice" });
```

// Delete multiple documents

```
db.collection.deleteMany({ age: { $gt: 30 }  
});
```



Document- Based NOSQL Systems and MongoDB

- Two-phase commit method
 - Used to ensure atomicity and consistency of multidocument transactions
- Replication in MongoDB
 - Concept of replica set to create multiple copies on different nodes
 - Variation of master-slave approach
 - Primary copy, secondary copy, and arbiter
 - Arbiter participates in elections to select new primary if needed

Document- Based NOSQL Systems and MongoDB

- Replication in MongoDB (cont'd.)
 - All write operations applied to the primary copy and propagated to the secondaries
 - User can choose read preference
 - Read requests can be processed at any replica
- Sharding in MongoDB
 - Horizontal partitioning divides the documents into disjoint partitions (shards)
 - Allows adding more nodes as needed
 - Shards stored on different nodes to achieve load balancing

Document- Based NOSQL Systems and MongoDB

- Sharding in MongoDB (cont'd.)
 - Partitioning field (shard key) must exist in every document in the collection
 - Must have an index
 - Range partitioning
 - Creates chunks by specifying a range of key values
 - Works best with range queries
 - Hash partitioning
 - Partitioning based on the hash values of each shard key

NOSQL KeyValue Stores

- Key-value stores focus on high performance, availability, and scalability
 - Can store structured, unstructured, or semi- structured data
- Key: unique identifier associated with a data item
 - Used for fast retrieval
- Value: the data item itself
 - Can be string or array of bytes
 - Application interprets the structure
- No query language

NOSQL KeyValue Stores

- DynamoDB part of Amazon's Web Services/SDK platforms
 - Proprietary
- Table holds a collection of self-describing items
- Item consists of attribute-value pairs
 - Attribute values can be single or multi-valued
- Primary key used to locate items within a table
 - Can be single attribute or pair of attributes

NOSQL KeyValue Stores

- Voldemort: open source key-value system similar to DynamoDB
- Voldemort features
 - Simple basic operations (get, put, and delete)
 - High-level formatted data values
 - Consistent hashing for distributing (key, value) pairs
 - Consistency and versioning
 - Concurrent writes allowed
 - Each write associated with a vector clock



NOSQL KeyValue Stores

- Oracle key-value store
 - Oracle NOSQL Database
- Redis key-value cache and store
 - Caches data in main memory to improve performance
 - Offers master-slave replication and high availability
 - Offers persistence by backing up cache to disk
- Apache Cassandra
 - Offers features from several NOSQL categories
 - Used by Facebook and others

Column- Based or Wide Column NOSQL Systems

- BigTable: Google's distributed storage system for big data
 - Used in Gmail
 - Uses Google File System for data storage and distribution
- Apache Hbase a similar, open source system
 - Uses Hadoop Distributed File System (HDFS) for data storage
 - Can also use Amazon's Simple Storage System (S3)

Column- Based or Wide Column NOSQL Systems

Hbase Data Model and Versioning

- Data organization concepts
 - Namespaces
 - Tables
 - Column families
 - Column qualifiers
 - Columns
 - Rows
 - Data cells
- Data is self-describing



Column-Based or Wide Column NOSQL Systems

- HBase stores multiple versions of data items
 - Timestamp associated with each version
- Each row in a table has a unique row key
- Table associated with one or more column families
- Column qualifiers can be dynamically specified as new table rows are created and inserted
- Namespace is collection of tables
- Cell holds a basic data item

Column-Based or Wide Column NOSQL Systems

(a) **creating a table:**

```
create 'EMPLOYEE', 'Name', 'Address', 'Details'
```

(b) **Inserting some row data in the EMPLOYEE table:**

```
put 'EMPLOYEE', 'row1', 'Name:Fname', 'John'  
put 'EMPLOYEE', 'row1', 'Name:Lname', 'Smith'  
put 'EMPLOYEE', 'row1', 'Name:Nickname', 'Johnny'  
put 'EMPLOYEE', 'row1', 'Details:Job', 'Engineer'  
put 'EMPLOYEE', 'row1', 'Details:Review', 'Good'  
put 'EMPLOYEE', 'row2', 'Name:Fname', 'Alicia'  
put 'EMPLOYEE', 'row2', 'Name:Lname', 'Zelaya'  
put 'EMPLOYEE', 'row2', 'Name:MName', 'Jennifer'  
put 'EMPLOYEE', 'row2', 'Details:Job', 'DBA'  
put 'EMPLOYEE', 'row2', 'Details:Supervisor', 'James Borg'  
put 'EMPLOYEE', 'row3', 'Name:Fname', 'James'  
put 'EMPLOYEE', 'row3', 'Name:Minit', 'E'  
put 'EMPLOYEE', 'row3', 'Name:Lname', 'Borg'  
put 'EMPLOYEE', 'row3', 'Name:Suffix', 'Jr.'  
put 'EMPLOYEE', 'row3', 'Details:Job', 'CEO'  
put 'EMPLOYEE', 'row3', 'Details:Salary', '1,000,000'
```

(c) **Some Hbase basic CRUD operations:**

Creating a table: create <tablename>, <column family>, <column family>, ...

Inserting Data: put <tablename>, <rowid>, <column family>:<column qualifier>, <value>

Reading Data (all data in a table): scan <tablename>

Retrieve Data (one item): get <tablename>,<rowid>

Column- Based or Wide Column NOSQL Systems

- Provides only low-level CRUD (create, read, update, delete) operations
- Application programs implement more complex operations
- Create - Creates a new table and specifies one or more column families associated with the table
- Put - Inserts new data or new versions of existing data items
- Get - Retrieves data associated with a single row
- Scan - Retrieves all the rows

Column- Based or Wide Column NOSQL Systems

- Each Hbase table divided into several regions
 - Each region holds a range of the row keys in the table
 - Row keys must be lexicographically ordered
 - Each region has several stores
 - Column families are assigned to stores
- Regions assigned to region servers for storage
 - Master server responsible for monitoring the region servers
- Hbase uses Apache Zookeeper and HDFS

End of Module 5

