Lecture Notes

DATABASE MANAGEMENT SYSTEM (BCS403)

Module – 5

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, NOSQL Graph Databases and Neo4j.

Textbook 1: Chapter 21.1 to 21.5, Chapter 24.1 to 24.6 **RBT: L1, L2, L3**

Teaching-Learning Process	Chalk and board, MOOC

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.

Definitions:

Concurrency control techniques are used to ensure the noninterference or isolation property of concurrently executing transactions.

Most of these techniques ensure serializability of schedules using **concurrency control protocols** (sets of rules) that guarantee serializability.

Two-phase locking protocols—employs the technique of **locking** data items to prevent multiple transactions from accessing the items concurrently.

A **timestamp** is a unique identifier for each transaction, generated by the system. Timestamp values are generated in the same order as the transaction start times. There are concurrency control protocols based on timestamp.

Validation or **certification** of a transaction after it executes its operations; these are sometimes called **optimistic protocols.**

Another factor that affects concurrency control is the **granularity** of the data items—that is, what portion of the database a data item represents.

1. Two-Phase Locking Techniques for Concurrency Control

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. Generally, there is one lock for each data item in the database. Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

Types of Locks and System Lock Tables

Several types of locks are used in 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 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 (the transaction **locks** the item) 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 (**unlocks** the item) so that *X* may be accessed by other transactions. Hence, a binary lock enforces **mutual exclusion** on the data item.

```
lock_item(X):
    B: if LOCK(X) = 0 (*item is unlocked*)
       then LOCK(X) ←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) ← 0; (* unlock the item *)
         if any transactions are waiting
          then wakeup one of the waiting
             transactions;
```

The DBMS has a **lock manager subsystem** to keep track of and control access to locks.

If the simple binary locking scheme described here is used, 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.*1
4. A transaction *T* will not issue an unlock_item(*X*) operation unless it already holds the lock on item *X*.

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.*

A different type of lock, called a **multiple-mode lock**, is used. In this scheme—called **shared/exclusive** or **read/write** locks—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*.

read-locked item is also called **share-locked** because other transactions are allowed to read the item, whereas a **write-locked item** is called **exclusive-locked** because a single transaction exclusively holds the lock on the item.

```
read_lock(X):
B:  if LOCK(X) = "unlocked"
        then begin LOCK(X) ← "read-locked";
            no_of_reads(X) ← 1
            end
    else if LOCK(X) = "read-locked"
        then no_of_reads(X) ← no_of_reads(X) + 1
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;

write_lock(X):
B:  if LOCK(X) = "unlocked"
        then LOCK(X) ← "write-locked"
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
```

```
unlock (X):
    if LOCK(X) = "write-locked"
        then begin LOCK(X) ← "unlocked";
                wakeup one of the waiting transactions, if any
                end
    else it LOCK(X) = "read-locked"
        then begin
                no_of_reads(X) ← no_of_reads(X) −1;
                if no_of_reads(X) = 0
                    then begin LOCK(X) = "unlocked";
                            wakeup one of the waiting transactions, if any
                            end
                end;
```

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.
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. This rule may be relaxed for downgrading of locks.
5. **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. This rule may also be relaxed for upgrading of locks.
6. **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.

Conversion (Upgrading, Downgrading) of Locks.

It is desirable to relax conditions 4 and 5 in the preceding list in order to allow **lock conversion**; that is, 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.

For example, it is possible for a transaction *T* to issue a read_lock(*X*) and then later to **upgrade** the lock by issuing a write_lock(*X*) operation.

If *T* is the only transaction holding a read lock on *X* at the time it 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. When upgrading and downgrading of locks is used, the lock table must include transaction identifiers in the record structure for each lock (in the locking_transaction(s) field) to store the information on which transactions hold locks on the item.

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.

Such 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; and

a **shrinking (second) phase**, during which existing locks can be released but no new locks can be acquired.

If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase.

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

Initial values: X=20, Y=30

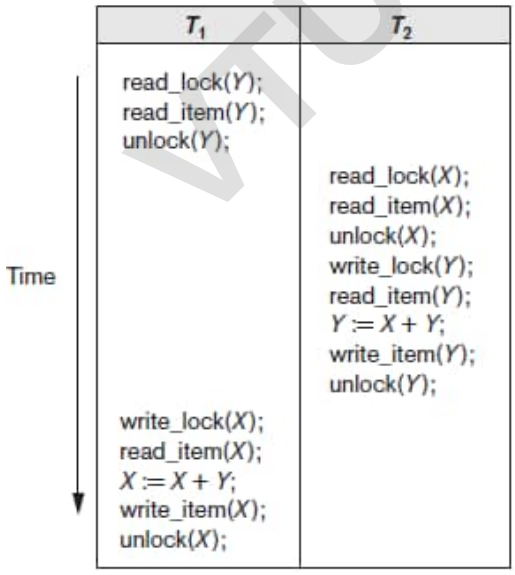
Result serial schedule $T1$

followed by $T2$: X=50, Y=80

Result of serial schedule $T2$

followed by $T1$: X=70, Y=50

Transactions $T1$ and $T2$ in Figure do not follow the two-phase locking protocol because the write_lock(X) operation follows the unlock(Y) operation in $T1$, and similarly the write_lock(Y) operation follows the unlock(X) operation in $T2$.



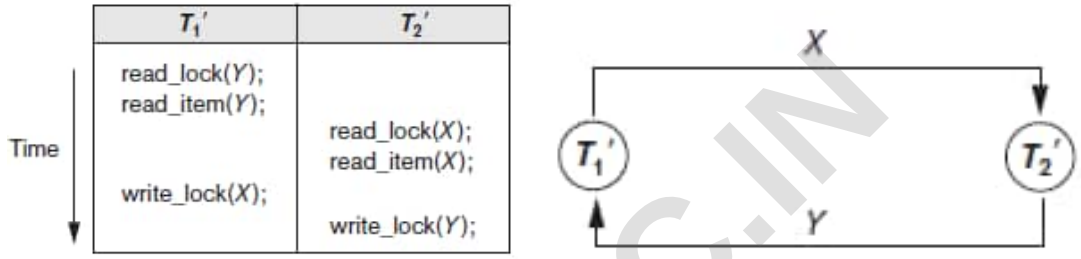
T_1'	T_2'

read_lock(Y);
read_item(Y);
write_lock(X);
unlock(Y)
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);
	read_lock(X);
	read_item(X);
	write_lock(Y);
	unlock(X)
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

Transactions T1′ and T2′, which are the same as T1 and T2 in Figure but follow the two-phase locking protocol. Note that they can produce a deadlock.

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.



Deadlock Prevention Protocols. One way to prevent deadlock is to use a **deadlock prevention protocol**. One deadlock prevention protocol, which is used in conservative two-phase locking, requires that every transaction lock *all the items it needs in advance* (which is generally not a practical assumption)—if any of the items cannot be obtained, none of the items are locked.

Transaction timestamp TS(T′), which 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).

Two schemes that prevent deadlock are called wait-die and wound-wait. Suppose that transaction Ti tries to lock an item X but is not able to because X is locked by some other transaction Tj with a conflicting lock. The rules followed by these schemes are:

■ **Wait-die**. If TS(Ti) < TS(Tj), then (Ti older than Tj) Ti is allowed to wait; otherwise (Ti younger than Tj) abort Ti (Ti dies) and restart it later with the same timestamp.

■ **Wound-wait**. If TS(Ti) < TS(Tj), then (Ti older than Tj) abort Tj (Ti wounds Tj) and restart it later with the same timestamp; otherwise (Ti younger than Tj) Ti is allowed to wait.

Starvation.

Another problem that may occur when we use locking 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.

One solution for starvation is to have a fair waiting scheme, such as using a **first-come-first-served** queue; transactions are enabled to lock an item in the order in which they originally requested the lock.

Another scheme 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 proceeds.

Starvation can also occur because of victim selection if the algorithm selects the same transaction as victim repeatedly, thus causing it to abort and never finish execution.

The use of locking, combined with the 2PL protocol, guarantees serializability of schedules.

The serializable schedules produced by 2PL have their equivalent serial schedules based on the order in which executing transactions lock the items they acquire.

If a transaction needs an item that is already locked, it may be forced to wait until the item is released.

Some transactions may be aborted and restarted because of the deadlock problem.

A different approach to concurrency control involves using transaction timestamps to order transaction execution for an equivalent serial schedule.

2. Concurrency Control Based on Timestamp Ordering

Timestamps

A timestamp is a unique identifier created by the DBMS to identify a transaction.

Typically, 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*. We will refer to the timestamp of transaction *T* as **TS(*T*)**.

Concurrency control techniques based on timestamp ordering do not use locks; hence, *deadlocks cannot occur*.

Timestamps can be generated in several ways. 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.

The Timestamp Ordering Algorithm for Concurrency Control

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.

This is called **timestamp ordering (TO)**.

Notice how this differs from 2PL, where a schedule is serializable by being equivalent to some serial schedule allowed by the locking protocols.

In timestamp ordering, however, the schedule is equivalent to the *particular serial order* corresponding to the order of the transaction timestamps.

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

1. **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, read_TS(*X*) = TS(*T*), where *T* is the *youngest* transaction that has read *X* successfully.

2. **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, write_TS(*X*) = TS(*T*), where *T* is the *youngest* transaction that has written *X* successfully. Based on the algorithm, *T* will also be the last transaction to write item *X*

Basic Timestamp Ordering (TO).

Whenever some transaction *T* tries to issue a read_item(*X*) or a write_item(*X*) operation, the **basic TO** algorithm compares the timestamp of *T* with read_TS(*X*) and write_TS(*X*) to ensure that the timestamp order of transaction execution is not violated.

The concurrency control algorithm must check whether conflicting operations violate the timestamp ordering in the following two cases:

1. Whenever a transaction *T* issues a write_item(*X*) operation, the following check is performed:
    a. If read_TS(*X*) > TS(*T*) or if write_TS(*X*) > TS(*T*), then abort and roll back *T* and reject the operation. This should be done because some *younger* transaction with a timestamp greater than TS(*T*)—and hence *after T* in the timestamp ordering—has already read or written the value of item *X* before *T* had a chance to write *X*, thus violating the timestamp ordering.
    b. If the condition in part (a) does not occur, then execute the write_item(*X*) operation of *T* and set write_TS(*X*) to TS(*T*).
2. Whenever a transaction *T* issues a read_item(*X*) operation, the following check is performed:
    a. If write_TS(*X*) > TS(*T*), then abort and roll back *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* had a chance to read *X*.
    b. If write_TS(*X*) ≤ TS(*T*), then execute the read_item(*X*) operation of *T* and set read_TS(*X*) to the *larger* of TS(*T*) and the current read_TS(*X*).

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.

In this variation, a transaction T issues a read_item(X) or write_item(X) such that TS(T) > write_TS(X) has its read or write operation delayed until the transaction T′ that wrote the value of X (hence TS(T′) = write_TS(X)) has committed or aborted.

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 by modifying the checks for the write_item(X) operation as follows:

1. If read_TS(X) > TS(T), then abort and roll back T and reject the operation.
2. If write_TS(X) > TS(T), then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than TS(T)—and hence after T in the timestamp ordering—has already written the value of X. Thus, we must ignore the write_item(X) operation of T because it is already outdated and obsolete. Notice that any conflict arising from this situation would be detected by case (1).
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).

3. 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** because several versions (values) of an item are kept by the system.

When a transaction requests to read an item, the *appropriate* version is chosen to maintain the serializability of the currently executing schedule.

One reason for keeping multipleversions is that 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. When a transaction writes an item, it writes a *new version* and the old version(s) of the item is retained.

An obvious drawback of multiversion techniques is that more storage is needed to maintain multiple versions of the database items. In some cases, older versions can be kept in a temporary store.

In this method, several versions *X*1, *X*2, … , *Xk* of each data item *X* are maintained. For *each version*, the value of version *Xi* and the following two timestamps associated

with version *Xi* are kept:

1. **read_TS(*Xi*).** The **read timestamp** of *Xi* is the largest of all the timestamps of transactions that have successfully read version *Xi*.
2. **write_TS(*Xi*).** The **write timestamp** of *Xi* is the timestamp of the transaction that wrote the value of version *Xi*.

Whenever a transaction *T* is allowed to execute a write_item(*X*) operation, a new version *Xk*+1 of item *X* is created, with both the write_TS(*Xk*+1) and the read_TS(*Xk*+1) set to TS(*T*). Correspondingly, when a transaction *T* is allowed to read the value of version *Xi*, the value of read_TS(*Xi*) is set to the larger of the current read_TS(*Xi*) and TS(*T*).

To ensure serializability, the following rules are used:

1. If transaction *T* issues a write_item(*X*) operation, and version *i* of *X* has the highest write_TS(*Xi*) of all versions of *X* that is also *less than or equal to* TS(*T*), and read_TS(*Xi*) > TS(*T*), then abort and roll back transaction *T*; otherwise, create a new version *Xj* of *X* with read_TS(*Xj*) = write_TS(*Xj*) = TS(*T*).
2. If transaction *T* issues a read_item(*X*) operation, find the version *i* of *X* that has the highest write_TS(*Xi*) of all versions of *X* that is also *less than or equal to* TS(*T*); then return the value of *Xi* to transaction *T,* and set the value of read_TS(*Xi*) to the larger of TS(*T*) and the current read_TS(*Xi*).

In this multiple-mode locking scheme, there are *three locking modes* for an item— read, write, and *certify*—instead of just the two modes (read, write).

	Read	Write
Read	Yes	No
Write	No	No

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No

We can describe the relationship between read and write locks in the standard scheme by means of the **lock compatibility table.**

The idea behind multiversion 2PL is to allow other transactions T′ to read an item X while a single transaction T holds a write lock on X. This is accomplished by allowing two versions for each item X; one version, the committed version, must always have been written by some committed transaction.

The second local version X′ can be created when a transaction T acquires a write lock on X. Other transactions can continue to read the committed version of X while T holds the write lock.

Transaction T can write the value of X′ as needed, without affecting the value of the committed version X. However, once T is ready to commit, it must obtain a certify lock on all items that it currently holds write locks on before it can commit; this is another form of lock upgrading.

4. Validation (Optimistic) Techniques and Snapshot Isolation Concurrency Control

In **optimistic concurrency control techniques**, also known as **validation** or **certification techniques**, *no checking* is done while the transaction is executing. Several concurrency control methods are based on the validation technique.

The concurrency control techniques that are based on the concept of **snapshot isolation**.

The implementations of these concurrency control methods can utilize a combination of the concepts from validation-based techniques and versioning techniques, as well as utilizing timestamps.

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. At the end of transaction execution, a **validation phase** checks whether any of the transaction's updates violate serializability.

There are three phases for this concurrency control protocol:

1. **Read phase.** A transaction can read values of committed data items from the database. However, 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.
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.

The validation phase for *Ti* checks that, for *each* such transaction *Tj* that is either recently committed or is in its validation phase, *one* of the following conditions holds:

1. Transaction *Tj* completes its write phase before *Ti* starts its read phase.
2. *Ti* starts its write phase after *Tj* completes its write phase, and the read_set of *Ti* has no items in common with the write_set of *Tj*.
3. Both the read_set and write_set of *Ti* have no items in common with the write_set of *Tj*, and *Tj* completes its read phase before *Ti* completes its read phase.

Concurrency Control Based on Snapshot Isolation

The basic definition of **snapshot isolation** is that a transaction sees the data items that it reads based on the committed values of the items in the *database snapshot* (or database state) when the transaction starts.

Snapshot isolation will ensure that the phantom record problem does not occur, since the database transaction, or, in some cases, the database statement, will only see the records that were committed in the database at the time the transaction started.

In this scheme, read operations do not require read locks to be applied to the items, thus reducing the overhead associated with two-phase locking. However, write operations do require write locks.

Thus, for transactions that have many reads, the performance is much better than 2PL.

When writes do occur, the system will have to keep track of older versions of the updated items in a **temporary version store** (sometimes known as tempstore), with the timestamps of when the version was created.

5. Granularity of Data Items and Multiple Granularity Locking

All concurrency control techniques assume that the database is formed of a number of named data items. 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. Several tradeoffs must be considered in choosing the data item size.

First, notice that the larger the data item size is, the lower the degree of concurrency permitted. For example, if the data item size is a disk block, a transaction *T* that needs to lock a single record *B* must lock the whole disk block *X* that contains *B* because a lock is associated with the whole data item (block).

On the other hand, the 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.

Multiple Granularity Level Locking

Since the best granularity size depends on the given transaction, it seems appropriate that a database system should support multiple levels of granularity, where the granularity level can be adjusted dynamically for various mixes of transactions.

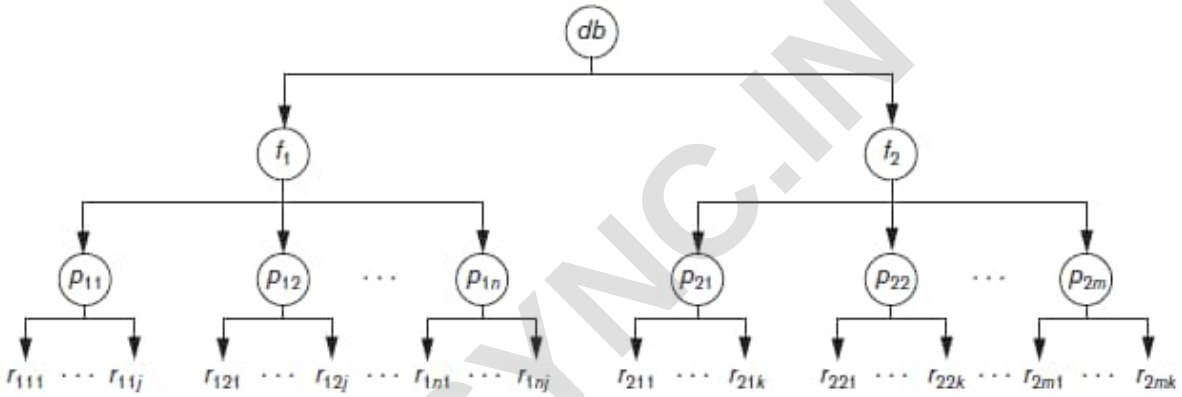


Figure shows a simple granularity hierarchy with a database containing two files, each file containing several disk pages, and each page containing several records.

This can be used to illustrate a **multiple granularity level** 2PL protocol, with shared/exclusive locking modes, where a lock can be requested at any level. However, additional types of locks will be needed to support such a protocol efficiently.

To make multiple granularity level locking practical, additional types of locks, called **intention locks**, are needed. The idea behind intention locks is for a transaction to indicate, along the path from the root to the desired node, what type of lock (shared or exclusive) it will require from one of the node's descendants. 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).

The **multiple granularity locking (MGL)** protocol consists of the following rules:

1. The lock compatibility (based on Figure 21.8) must be adhered to.
2. The root of the tree must be locked first, in any mode.
3. A node *N* can be locked by a transaction *T* in S or IS mode only if the parent node *N* is already locked by transaction *T* in either IS or IX mode.
4. A node *N* can be locked by a transaction *T* in X, IX, or SIX mode only if the parent of node *N* is already locked by transaction *T* in either IX or SIX mode.
5. A transaction *T* can lock a node only if it has not unlocked any node (to enforce the 2PL protocol).
6. A transaction *T* can unlock a node, *N*, only if none of the children of node *N* are currently locked by *T.*

Chapter 2:

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, NOSQL Graph Databases and Neo4j

1. Introduction to NOSQL Systems

The term **NOSQL** is generally interpreted as Not Only SQL—rather than NO to SQL—and is meant to convey that many applications need systems other than traditional relational SQL systems to augment their data management needs. Most NOSQL systems are distributed databases or distributed storage systems, with a focus on semistructured data storage, high performance, availability, data replication, and scalability as opposed to an emphasis on immediate data consistency, powerful query languages, and structured data storage.

Emergence of NOSQL Systems

Many companies and organizations are faced with applications that store vast amounts of data. Consider a free e-mail application, such as Google Mail or Yahoo Mail or other similar service—this application can have millions of users, and each user can have thousands of e-mail messages.

There is a need for a storage system that can manage all these e-mails; a structured relational SQL system may not be appropriate because
(1) SQL systems offer too many services (powerful query language, concurrency control, etc.), which this application may not need; and
(2) a structured data model such the traditional relational model may be too restrictive.

Consider an application such as Facebook, with millions of users who submit posts, many with images and videos; then these posts must be displayed on pages of other users using the social media relationships among the users. User profiles, user relationships, and posts

must all be stored in a huge collection of data stores, and the appropriate posts must be made available to the sets of users that have signed up to see these posts.

Google developed a proprietary NOSQL system known as **BigTable**, which is used in many of Google's applications that require vast amounts of data storage, such as Gmail, Google Maps, and Web site indexing.

Amazon developed a NOSQL system called **DynamoDB** that is available through Amazon's cloud services.

Other software companies started developing their own solutions and making them available to users who need these capabilities—for example, **MongoDB** and **CouchDB**, which are classified as **document-based** NOSQL systems or **document stores.**

Another category of NOSQL systems is the **graph-based** NOSQL systems, or **graph databases**; these include **Neo4J** and **GraphBase**, among others.

Characteristics of NOSQL Systems*

NOSQL characteristics related to distributed databases and distributed systems. NOSQL systems emphasize high availability, so replicating the data is inherent in many of these systems.

1. **Scalability: T**here are two kinds of scalability in distributed systems: horizontal and vertical. In NOSQL systems, **horizontal scalability** is generally used, where the distributed system is expanded by adding more nodes for data storage and processing as the volume of data grows. Vertical scalability, on the other hand, refers to expanding the storage and computing power of existing nodes.
2. **Availability, Replication and Eventual Consistency:** Many applications that use NOSQL systems require continuous system availability. To accomplish this, data is replicated over two or more nodes in a transparent manner, so that if one node fails, the data is still available on other nodes. Replication improves data availability and can also improve read performance, because read requests an often be serviced from any of the replicated data nodes. However, write performance becomes more

cumbersome because an update must be applied to every copy of the replicated data items; this can slow down write performance if serializable consistency is required.

3. **Replication Models:** Two major replication models are used in NOSQL systems: master-slave and master-master replication.
    a. **Master-slave replication** requires one copy to be the master copy; all write operations must be applied to the master copy and then propagated to the slave copies, usually using eventual consistency (the slave copies will *eventually* be the same as the master copy).
    b. The **master-master replication** allows reads and writes at any of the replicas but may not guarantee that reads at nodes that store different copies see the same values. Different users may write the same data item concurrently at different nodes of the system, so the values of the item will be temporarily inconsistent.
4. S**harding of Files:** In many NOSQL applications, files (or collections of data objects) can have many millions of records (or documents or objects), and these records can be accessed concurrently by thousands of users. So it is not practical to store the whole file in one node. **Sharding** (also known as **horizontal partitioning**) of the file records is often employed in NOSQL systems. This serves to distribute the load of accessing the file records to multiple nodes.
5. **High-Performance Data Access:** In many NOSQL applications, it is necessary to find individual records or objects (data items) from among the millions of data records or objects in a file. To achieve this, most systems use one of two techniques: hashing or range partitioning on object keys. The majority of accesses to an object will be by providing the key value rather than by using complex query conditions.

NOSQL characteristics related to data models and query languages.

NOSQL systems emphasize performance and flexibility over modelling power and complex querying. We discuss some of these characteristics next.

1. **Not Requiring a Schema:** The flexibility of not requiring a schema is achieved in many NOSQL systems by allowing semi-structured, selfdescribing data. The users can specify a partial schema in some systems to improve storage efficiency, but it is *not required to have a schema* in most of the NOSQL systems. As there may not be a schema to specify constraints, any constraints on the data would have to be programmed in the application programs that access the data items. There are various languages for describing semistructured data, such as JSON (JavaScript Object Notation) and XML (Extensible Markup Language).
2. **Less Powerful Query Languages:** Many applications that use NOSQL systems may not require a powerful query language such as SQL, because search (read) queries in these systems often locate single objects in a single file based on their object keys. NOSQL systems typically provide a set of functions and operations as a programming API (application programming interface), so reading and writing the data objects is accomplished by calling the appropriate operations by the programmer. In many cases, the operations are called **CRUD operations**, for Create, Read, Update, and Delete. In other cases, they are known as **SCRUD** because of an added Search (or Find) operation. Some NOSQL systems also provide a high-level query language, but it may not have the full power of SQL; only a subset of SQL querying capabilities would be provided. In particular, many NOSQL systems do not provide join operations as part of the query language itself; the joins need to be implemented in the application programs.
3. **Versioning:** Some NOSQL systems provide storage of multiple versions of the data items, with the timestamps of when the data version was created.

Categories of NOSQL Systems*

NOSQL systems have been characterized into four major categories, with some additional categories that encompass other types of systems. The most common categorization lists the following four major categories:

1. **Document-based NOSQL systems:** These systems store data in the form of documents using well-known formats, such as JSON (JavaScript Object Notation).

Documents are accessible via their document id, but can also be accessed rapidly using other indexes.

2. **NOSQL key-value stores:** These systems have a simple data model based on fast access by the key to the value associated with the key; the value can be a record or an object or a document or even have a more complex data structure.
3. **Column-based or wide column NOSQL systems:** These systems partition a table by column into column families where each column family is stored in its own files. They also allow versioning of data values.
4. **Graph-based NOSQL systems:** Data is represented as graphs, and related nodes can be found by traversing the edges using path expressions.

Additional categories can be added as follows to include some systems that are not easily categorized into the above four categories, as well as some other types of systems that have been available even before the term NOSQL became widely used.

5. **Hybrid NOSQL systems:** These systems have characteristics from two or more of the above four categories.
6. **Object databases:** Database systems that were based on the object data model were known originally as object-oriented databases (OODBs) but are now referred to as **object databases** (**ODBs**).
7. **XML databases:** A new language—namely, XML (Extensible Markup Language)—has emerged as the standard for structuring and exchanging data over the Web in text files. Another language that can be used for the same purpose is JSON (JavaScript Object Notation).

Even keyword-based search engines store large amounts of data with fast search access, so the stored data can be considered as large NOSQL big data stores.

2 The CAP Theorem*

The CAP theorem, which was originally introduced as the CAP principle, can be used to explain some of the competing requirements in a distributed system with replication.

The three letters in CAP refer to three desirable properties of distributed systems with replicated data:

- **Consistency** (among replicated copies),
- **Availability** (of the system for read and write operations) and
- **Partition tolerance** (in the face of the nodes in the system being partitioned by a network fault).

Availability means that each read or write request for a data item will either be processed successfully or will receive a message that the operation cannot be completed.

Partition tolerance means that the system can continue operating if the network connecting the nodes has a fault that results in two or more partitions, where the nodes in each partition can only communicate among each other.

Consistency means that the nodes will have the same copies of a replicated data item visible for various transactions.

It is important to note here that the use of the word *consistency* in CAP and its use in ACID *do not refer to the same identical concept*.

In CAP, the term *consistency* refers to the consistency of the values in different copies of the same data item in a replicated distributed system. In ACID, it refers to the fact that a transaction will not violate the integrity constraints specified on the database schema. However, if we consider that the consistency of replicated copies is a *specified constraint*, then the two uses of the term *consistency* would be related.

The **CAP theorem** states that it *is not possible to guarantee all three* of the desirable properties—consistency, availability, and partition tolerance—at the same time in a distributed system with data replication. If this is the case, then the distributed system designer would have to choose two properties out of the three to guarantee.

It is generally assumed that in many traditional (SQL) applications, guaranteeing consistency through the ACID properties is important. On the other hand, in a NOSQL distributed data store, a weaker consistency level is often acceptable, and guaranteeing the other two

properties (availability, partition tolerance) is important. Hence, weaker consistency levels are often used in NOSQL system instead of guaranteeing serializability. In particular, a form of consistency known as **eventual consistency** is often adopted in NOSQL systems.

3 Document-Based NOSQL Systems and MongoDB

Document-based or document-oriented NOSQL systems typically store data as **collections** of similar **documents**. These types of systems are also sometimes known as **document stores**. The individual documents somewhat resemble *complex* or XML documents, but a major difference between document-based systems versus object and object-relational systems and XML is that there is no requirement to specify a schema—rather, the documents are specified as **self-describing data**. Although the documents in a collection should be *similar*, they can have different data elements (attributes), and new documents can have new data elements that do not exist in any of the current documents in the collection.

MongoDB Data Model

MongoDB documents are stored in BSON (Binary JSON) format, which is a variation of JSON with some additional data types and is more efficient for storage than JSON. Individual **documents** are stored in a **collection**.

We will use a simple example based on our COMPANY database that we used throughout this reading.

The operation createCollection is used to create each collection. For example, the following command can be used to create a collection called **project** to hold PROJECT objects from the COMPANY database:

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

The first parameter “project” is the **name** of the collection, which is followed by an optional document that specifies **collection options**. In our example, the collection is **capped**; this means it has upper limits on its storage space (**size**) and number of documents (**max**). The capping parameters help the system choose the storage options for each collection.

For our example, we will create another document collection called **worker** to hold information about the EMPLOYEEs who work on each project; for example:

db.createCollection("worker", { capped : true, size : 5242880, max : 2000 }))

Each document in a collection has a unique **ObjectId** field, called **_id**, which is automatically indexed in the collection unless the user explicitly requests no index for the _id field. The value of ObjectId can be *specified by the user*, or it can be *system-generated* if the user does not specify an _id field for a particular document.

System-generated ObjectIds have a specific format, which combines the timestamp when the object is created (4 bytes, in an internal MongoDB format), the node id (3 bytes), the process id (2 bytes), and a counter (3 bytes) into a 16-byte Id value.

User-generated ObjectsIds can have any value specified by the user as long as it uniquely identifies the document and so these Ids are similar to primary keys in relational systems.

```
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
                  }
              ]
);
```

Denormalized document design with embedded subdocuments.

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
}
```

Embedded array of document references.

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
}
```

Normalized documents.

```
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 } ] )
```

Inserting the documents into their collections.

MongoDB CRUD Operations

MongoDb has several **CRUD operations**, where CRUD stands for (create, read, update, delete). Documents can be *created* and inserted into their collections using the **insert** operation, whose format is:

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

The parameters of the insert operation can include either a single document or an array of documents, as shown in Figure 24.1(d). The *delete* operation is called **remove**, and the format is:

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

The documents to be removed from the collection are specified by a Boolean condition on some of the fields in the collection documents. There is also an **update** operation, which has a condition to select certain documents, and a *$set* clause to specify the update. It is also possible to use the update operation to replace an existing document with another one but keep the same ObjectId.

For *read* queries, the main command is called **find**, and the format is:

db.<collection_name>.find(<condition>)

General Boolean conditions can be specified as <condition>, and the documents in the collection that return **true** are selected for the query result.

MongoDB Distributed Systems Characteristics*

Most MongoDB updates are atomic if they refer to a single document, but MongoDB also provides a pattern for specifying transactions on multiple documents. Since MongoDB is a distributed system, the **two-phase commit** method is used to ensure atomicity and consistency of multidocument transactions.

Replication in MongoDB.

The concept of **replica set** is used in MongoDB to create multiple copies of the same data set on different nodes in the distributed system, and it uses a variation of the **master-slave** approach for replication. For example, suppose that we want to replicate a particular document collection C.

A replica set will have one **primary copy** of the collection C stored in one node N1, and at least one **secondary copy** (replica) of C stored at another node N2. Additional copies can be stored in nodes N3, N4, etc., as needed, but the cost of storage and update (write) increases with the number of replicas.

The total number of participants in a replica set must be at least three, so if only one secondary copy is needed, a participant in the replica set known as an **arbiter** must run on the third node N3.

The arbiter does not hold a replica of the collection but participates in **elections** to choose a new primary if the node storing the current primary copy fails. If the total number of members in a replica set is n (one primary plus i secondaries, for a total of $n = i + 1$), then n must be an odd number; if it is not, an *arbiter* is added to ensure the election process works correctly if the primary fails.

In MongoDB replication, all write operations must be applied to the primary copy and then propagated to the secondaries. For read operations, the user can choose the particular **read preference** for their application.

The *default read preference* processes all reads at the primary copy, so all read and write operations are performed at the primary node. In this case, secondary copies are mainly to make sure that the system continues operation if the primary fails, and MongoDB can ensure that every read request gets the latest document value.

To increase read performance, it is possible to set the read preference so that *read requests can be processed at any replica* (primary or secondary); however, a read at a secondary is not guaranteed to get the latest version of a document because there can be a delay in propagating writes from the primary to the secondaries.

Sharding in MongoDB.

When a collection holds a very large number of documents or requires a large storage space, storing all the documents in one node can lead to performance problems, particularly if there are many user operations accessing the documents concurrently using various CRUD operations.

Sharding of the documents in the collection—also known as *horizontal partitioning*—divides the documents into disjoint partitions known as **shards**. This allows the system to add more nodes as needed by a process known as **horizontal scaling** of the distributed system, and to store the shards of the collection on different nodes to achieve load balancing. Each node will process only those operations pertaining to the documents in the shard stored at that node. Also, each shard will contain fewer documents than if the entire collection were stored at one node, thus further improving performance.

There are two ways to partition a collection into shards in MongoDB—**range partitioning** and **hash partitioning**. Both require that the user specify a particular document field to be used as the basis for partitioning the documents into shards.

The *partitioning field*—known as the **shard key** in MongoDB—must have two characteristics: it must exist in *every document* in the collection, and it must have an *index*. The ObjectId can be used, but any other field possessing these two characteristics can also be used as the basis for sharding. The values of the shard key are divided into **chunks** either through range partitioning or hash partitioning, and the documents are partitioned based on the chunks of shard key values.

Range partitioning creates the chunks by specifying a range of key values; for example, if the shard key values ranged from one to ten million, it is possible to create ten ranges—1 to 1,000,000; 1,000,001 to 2,000,000; ... ; 9,000,001 to 10,000,000—and each chunk would contain the key values in one range. *Hash partitioning* applies a hash function *h(K)* to each shard key *K*, and the partitioning of keys into chunks is based on the hash values.

In general, if **range queries** are commonly applied to a collection (for example, retrieving all documents whose shard key value is between 200 and 400), then range partitioning is preferred because each range query will typically be submitted to a single node that contains all the required documents in one shard. If most searches retrieve one document at a time, hash partitioning may be preferable because it randomizes the distribution of shard key values into chunks.

When sharding is used, MongoDB queries are submitted to a module called the **query router**, which keeps track of which nodes contain which shards based on the particular partitioning method used on the shard keys. The query (CRUD operation) will be routed to the nodes that contain the shards that hold the documents that the query is requesting.

If the system cannot determine which shards hold the required documents, the query will be submitted to all the nodes that hold shards of the collection.

Sharding and replication are used together; sharding focuses on improving performance via load balancing and horizontal scalability, whereas replication focuses on ensuring system availability when certain nodes fail in the distributed system.

NOSQL Key-Value Stores

Key-value stores focus on high performance, availability, and scalability by storing data in a distributed storage system. The data model used in key-value stores is relatively simple, and in many of these systems, there is no query language but rather a set of operations that can be used by the application programmers.

The **key** is a unique identifier associated with a data item and is used to locate this data item rapidly.

The **value** is the data item itself, and it can have very different formats for different key-value storage systems.

In some cases, the value is just a *string of bytes* or an *array of bytes*, and the application using the key-value store has to interpret the structure of the data value. In other cases, some standard formatted data is allowed; for example, structured data rows (tuples) similar to relational data, or semistructured data using JSON or some other self-describing data format.

Different key-value stores can thus store unstructured, semistructured, or structured data items.

The main characteristic of key-value stores is the fact that every value (data item) must be associated with a unique key, and that retrieving the value by supplying the key must be very fast.

There are many systems that fall under the key-value store label, so rather than provide a lot of details on one particular system, we will give a brief introductory overview for some of these systems and their characteristics.

DynamoDB Overview*

The DynamoDB system is an Amazon product and is available as part of Amazon's **AWS/SDK** platforms (Amazon Web Services/Software Development Kit). It can be used as part of Amazon's cloud computing services, for the data storage component.

DynamoDB data model.

The basic data model in DynamoDB uses the concepts of tables, items, and attributes. A **table** in DynamoDB *does not have* a **schema**; it holds a collection of *self-describing items*. Each **item** will consist of a number of (attribute, value) pairs, and attribute values can be single-valued or multivalued. So basically, a table will hold a collection of items, and each item is a self-describing record (or object).

DynamoDB also allows the user to specify the items in JSON format, and the system will convert them to the internal storage format of DynamoDB. When a table is created, it is required to specify a **table name** and a **primary key**; the primary key will be used to rapidly locate the items in the table. Thus, the primary key is the **key** and the item is the **value** for the DynamoDB key-value store.

The primary key attribute must exist in every item in the table. The primary key can be one of the following two types:

■ **A single attribute.** The DynamoDB system will use this attribute to build a hash index on the items in the table. This is called a *hash type primary key*. The items are not ordered in storage on the value of the hash attribute.

■ **A pair of attributes.** This is called a *hash and range type primary key*. The primary key will be a pair of attributes (A, B): attribute A will be used for hashing, and because there will be multiple items with the same value of A, the B values will be used for ordering the records with the same A value. A table with this type of key can have additional secondary indexes defined on its attributes. For example, if we want to store multiple versions of some type of items in a table, we could use ItemID as hash and Date or Timestamp (when the version was created) as range in a hash and range type primary key.

DynamoDB Distributed Characteristics.

Because DynamoDB is proprietary, in the next subsection we will discuss the mechanisms used for replication, sharding, and other distributed system concepts in an open source key-value system called Voldemort. Voldemort is based on many of the techniques proposed for DynamoDB.

Voldemort Key-Value Distributed Data Store*

Voldemort is an open source system available through Apache 2.0 open source licensing rules. It is based on Amazon's DynamoDB.

The focus is on high performance and horizontal scalability, as well as on providing replication for high availability and sharding for improving latency (response time) of read and write requests. All three of those features—replication, sharding, and horizontal scalability—are realized through a technique to distribute the key-value pairs among the nodes of a distributed cluster; this distribution is known as **consistent hashing**.

Voldemort has been used by LinkedIn for data storage. Some of the features of Voldemort are as follows:

■ **Simple basic operations.** A collection of (key, value) pairs is kept in a Voldemort **store**. In our discussion, we will assume the store is called *s*. The basic interface for data storage and retrieval is very simple and includes three operations: get, put, and delete. The operation

s.put(k, v) inserts an item as a key-value pair with key k and value v. The operation s.delete(k) deletes the item whose key is k from the store, and the operation v = s.get(k) retrieves the value v associated with key k. The application can use these basic operations to build its own requirements. At the basic storage level, both keys and values are arrays of bytes (strings).

■ **High-level formatted data values.** The values v in the (k, v) items can be specified in JSON (JavaScript Object Notation), and the system will convert between JSON and the internal storage format. Other data object formats can also be specified if the application provides the conversion (also known as **serialization**) between the user format and the storage format as a *Serializer class*.

The Serializer class must be provided by the user and will include operations to convert the user format into a string of bytes for storage as a value, and to convert back a string (array of bytes) retrieved via *s.get(k)* into the user format. Voldemort has some built-in serializers for formats other than JSON.

■ **Consistent hashing for distributing (key, value) pairs.** A variation of the data distribution algorithm known as **consistent hashing** is used in Voldemort for data distribution among the nodes in the distributed cluster of nodes.

A hash function $h(k)$ is applied to the key k of each (k, v) pair, and $h(k)$ determines where the item will be stored. The method assumes that $h(k)$ is an integer value, usually in the range 0 to $Hmax = 2n-1$, where n is chosen based on the desired range for the hash values. This method is best visualized by considering the range of all possible integer hash values 0 to $Hmax$ to be evenly distributed on a circle (or ring).

The nodes in the distributed system are then also located on the same ring; usually each node will have several locations on the ring. The positioning of the points on the ring that represent the nodes is done in a psuedorandom manner.

An item (k, v) will be stored on the node whose position in the ring *follows* the position of $h(k)$ on the ring *in a clockwise direction*. In Figure below, we assume there are three nodes in the distributed cluster labeled A, B, and C, where node C has a bigger capacity than nodes A and B. In a typical system, there will be many more nodes. On the circle, two instances each of A and B are placed, and three instances of C (because of its higher capacity), in a

pseudorandom manner to cover the circle. Figure below, indicates which (k, v) items are placed in which nodes based on the *h(k)* values.

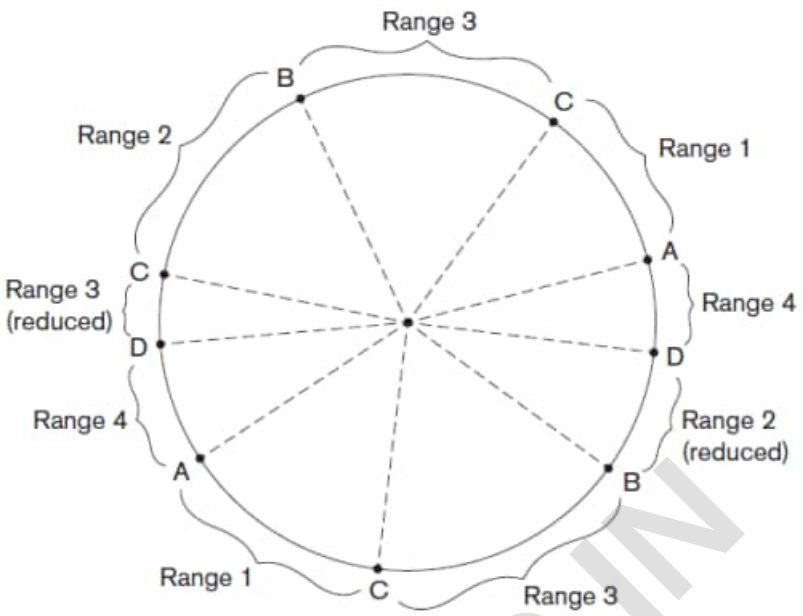
Ring having three nodes A,B, and C, with C having greater capacity. The *h*(*K*) values that map to the circle points in *range 1* have their (k, v) items stored in node A, *range 2* in node B, *range 3* in node C.

The *h*(*k*) values that fall in the parts of the circle marked as *range 1* in Figure above, will have their (k, v) items stored in node A because that is the node whose label follows *h*(*k*) on the ring in a clockwise direction; those in *range 2* are stored in node B; and those in *range 3* are stored in node C.

This scheme allows *horizontal scalability* because when a new node is added to the distributed system, it can be added in one or more locations on the ring depending on the node capacity. Only a limited percentage of the (k, v) items will be reassigned to the new node from the existing nodes based on the consistent hashing placement algorithm.

Also, those items assigned to the new node may not all come from only one of the existing nodes because the new node can have multiple locations on the ring.

For example, if a node D is added and it has two placements on the ring as shown in Figure below, then some of the items from nodes B and C would be moved to node D. The items whose keys hash to *range 4* on the circle (see Figure below) would be migrated to node D.



Adding a node D to the ring. Items in *range 4* are moved to the node D from node B (*range 2* is reduced) and node C (*range 3* is reduced).

This scheme also allows *replication* by placing the number of specified replicas of an item on successive nodes on the ring in a clockwise direction.

The *sharding* is built into the method, and different items in the store (file) are located on different nodes in the distributed cluster, which means the items are horizontally partitioned (sharded) among the nodes in the distributed system.

■ **Consistency and versioning.** Voldemort uses a method similar to the one developed for DynamoDB for consistency in the presence of replicas. Basically, concurrent write operations are allowed by different processes so there could exist two or more different values associated with the same key at different nodes when items are replicated.

Consistency is achieved when the item is read by using a technique known as *versioning and read repair*.

Concurrent writes are allowed, but each write is associated with a *vector clock* value. When a read occurs, it is possible that different versions of the same value (associated with the same key) are read from different nodes.

If the system can reconcile to a single final value, it will pass that value to the read; otherwise, more than one version can be passed back to the application, which will reconcile the various

versions into one version based on the application semantics and give this reconciled value back to the nodes.

Column-Based or Wide Column NOSQL Systems*

Another category of NOSQL systems is known as **column-based** or **wide column** systems. The Google distributed storage system for big data, known as **BigTable**, is a well-known example of this class of NOSQL systems, and it is used in many Google applications that require large amounts of data storage, such as Gmail. Big-Table uses the **Google File System** (**GFS**) for data storage and distribution. An open source system known as **Apache Hbase** is somewhat similar to Google Big-Table, but it typically uses **HDFS** (**Hadoop Distributed File System**) for data storage.

HDFS is used in many cloud computing applications. Hbase can also use Amazon's **Simple Storage System** (known as **S3**) for data storage. Another well-known example of column-based NOSQL systems is Cassandra, which we discussed briefly in Section 24.4.3 because it can also be characterized as a key-value store. We will focus on Hbase in this section as an example of this category of NOSQL systems.

BigTable (and Hbase) is sometimes described as a *sparse multidimensional distributed persistent sorted map*, where the word *map* means a *collection of (key, value) pairs* (the key is *mapped* to the value). One of the main differences that distinguish column-based systems from key-value stores is the *nature of the key*.

In column-based systems such as Hbase, the key is *multidimensional* and so has several components: typically, a combination of table name, row key, column, and timestamp. As we shall see, the column is typically composed of two components: column family and column qualifier.

Hbase Data Model and Versioning

Hbase data model. The data model in Hbase organizes data using the concepts of *namespaces*, *tables*, *column families*, *column qualifiers*, *columns*, *rows*, and *data cells*. A column is identified by a combination of (column family:column qualifier). Data is stored in a self-

describing form by associating columns with data values, where data values are strings. Hbase also stores *multiple versions* of a data item, with a *timestamp* associated with each version, so versions and timestamps are also part of the Hbase data model.

As with other NOSQL systems, unique keys are associated with stored data items for fast access, but the keys identify *cells* in the storage system. Because the focus is on high performance when storing huge amounts of data, the data model includes some storage-related concepts.

It is important to note that the use of the words *table*, *row*, and *column* is not identical to their use in relational databases, but the uses are related.

■ **Tables and Rows.** Data in Hbase is stored in **tables**, and each table has a table name. Data in a table is stored as self-describing **rows**. Each row has a unique **row key**, and row keys are strings that must have the property that they can be lexicographically ordered, so characters that do not have a lexicographic order in the character set cannot be used as part of a row key.

■ **Column Families, Column Qualifiers, and Columns.** A table is associated with one or more **column families**. Each column family will have a name, and the column families associated with a table *must be specified* when the table is created and cannot be changed later. The below create statement shows how a table may be created; the table name is followed by the names of the column families associated with the table.

When the data is loaded into a table, each column family can be associated with many **column qualifiers**, but the column qualifiers *are not specified* as part of creating a table. So the column qualifiers make the model a self-describing data model because the qualifiers can be dynamically specified as new rows are created and inserted into the table.

A **column** is specified by a combination of ColumnFamily:ColumnQualifier. Basically, column families are a way of grouping together related columns (attributes in relational terminology) for storage purposes, except that the column qualifier names are not specified during table creation.

Rather, they are specified when the data is created and stored in rows, so the data is *selfdescribing* since any column qualifier name can be used in a new row of data that added using put statement.

However, it is important that the application programmers know which column qualifiers belong to each column family, even though they have the flexibility to create new column qualifiers on the fly when new data rows are created.

The concept of column family is somewhat similar to *vertical partitioning*, because columns (attributes) that are accessed together because they belong to the same column family are stored in the same files. Each column family of a table is stored in its own files using the HDFS file system.

■ **Versions and Timestamps.** Hbase can keep several **versions** of a data item, along with the **timestamp** associated with each version. The timestamp is a long integer number that represents the system time when the version was created, so newer versions have larger timestamp values.

Hbase uses midnight 'January 1, 1970 UTC' as timestamp value zero, and uses a long integer that measures the number of milliseconds since that time as the system timestamp value (this is similar to the value returned by the Java utility java.util.Date.getTime() and is also used in MongoDB).

It is also possible for the user to define the timestamp value explicitly in a Date format rather than using the system-generated timestamp.

■ **Cells.** A **cell** holds a basic data item in Hbase. The key (address) of a cell is specified by a combination of (table, rowid, columnfamily, columnqualifier, timestamp).

If timestamp is left out, the latest version of the item is retrieved unless a default number of versions is specified, say the latest three versions.

The default number of versions to be retrieved, as well as the default number of versions that the system needs to keep, are parameters that can be specified during table creation.

■ **Namespaces.** A **namespace** is a collection of tables. A namespace basically specifies a collection of one or more tables that are typically used together by user applications, and it corresponds to a database that contains a collection of tables in relational terminology.

Examples in Hbase.

(a) Creating a table called EMPLOYEE with three column families: Name, Address, and Details.

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

(b) Inserting some in the EMPLOYEE table; different rows can have different self-describing column qualifiers (Fname, Lname, Nickname, Mname, Minit, Suffix, … for column family Name; Job, Review, Supervisor, Salary for column family Details).

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'

Hbase CRUD Operations

Hbase has low-level CRUD (create, read, update, delete) operations, as in many of the NOSQL systems. The formats of some of the basic CRUD operations in Hbase are shown below.

(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>

Hbase only provides low-level CRUD operations. It is the responsibility of the application programs to implement more complex operations, such as joins between rows in different tables. The *create* operation creates a new table and specifies one or more column families associated with that table, but it does not specify the column qualifiers, as we discussed earlier. The *put* operation is used for inserting new data or new versions of existing data items. The get operation is for retrieving the data associated with a single row in a table, and the *scan* operation retrieves all the rows.

Hbase Storage and Distributed System Concepts

Each Hbase table is divided into a number of **regions**, where each region will hold a *range* of the row keys in the table; this is why the row keys must be lexicographically ordered. Each region will have a number of **stores**, where each column family is assigned to one store within the region. Regions are assigned to **region servers** (storage nodes) for storage. A **master server** (master node) is responsible for monitoring the region servers and for splitting a table into regions and assigning regions to region servers.

Hbase uses the **Apache Zookeeper** open source system for services related to managing the naming, distribution, and synchronization of the Hbase data on the distributed Hbase server nodes, as well as for coordination and replication services.

Hbase also uses Apache HDFS (Hadoop Distributed File System) for distributed file services. So Hbase is built on top of both HDFS and Zookeeper. Zookeeper can itself have several replicas on several nodes for availability, and it keeps the data it needs in main memory to speed access to the master servers and region servers.

NOSQL Graph Databases and Neo4j*

Another category of NOSQL systems is known as **graph databases** or **graphoriented NOSQL** systems. The data is represented as a graph, which is a collection of vertices (nodes) and edges. Both nodes and edges can be labeled to indicate the types of entities and relationships they represent, and it is generally possible to store data associated with both individual nodes and individual edges. Many systems can be categorized as graph databases.

We will focus our discussion on one particular system, Neo4j, which is used in many applications. Neo4j is an open source system, and it is implemented in Java.

Neo4j Data Model

The data model in Neo4j organizes data using the concepts of **nodes** and **relationships**. Both nodes and relationships can have **properties**, which store the data items associated with nodes and relationships. Nodes can have **labels**; the nodes that have the *same label* are grouped into a collection that identifies a subset of the nodes in the database graph for querying purposes. A node can have zero, one, or several labels.

Relationships are directed; each relationship has a *start node* and *end node* as well as a **relationship type**, which serves a similar role to a node label by identifying similar relationships that have the same relationship type. Properties can be specified via a **map pattern**, which is made of one or more "name : value" pairs enclosed in curly brackets; for example {Lname : 'Smith', Fname : 'John', Minit : 'B'}.

In conventional graph theory, nodes and relationships are generally called *vertices* and *edges*. The Neo4j graph data model somewhat resembles how data is represented in the ER and EER models, but with some notable differences.

Comparing the Neo4j graph model with ER/EER concepts, nodes correspond to *entities*, node labels correspond to *entity types and subclasses*, relationships correspond to *relationship instances*, relationship types correspond to *relationship types*, and properties correspond to *attributes*.

One notable difference is that a relationship is *directed* in Neo4j, but is not in ER/EER. Another is that a node may have no label in Neo4j, which is not allowed in ER/EER because every entity must belong to an entity type.

A third crucial difference is that the graph model of Neo4j is used as a basis for an actual high-performance distributed database system whereas the ER/EER model is mainly used for database design.

The create command shows how a few nodes can be created in Neo4j. There are various ways in which nodes and relationships can be created; for example, by calling appropriate Neo4j operations from various Neo4j APIs.

We will just show the high-level syntax for creating nodes and relationships; to do so, we will use the Neo4j CREATE command, which is part of the high-level declarative query language **Cypher**.

Neo4j has many options and variations for creating nodes and relationships using various scripting interfaces, but a full discussion is outside the scope of our presentation.

■ **Labels and properties.** When a node is created, the node label can be specified. It is also possible to create nodes without any labels.

In create command, the node labels are EMPLOYEE, DEPARTMENT, PROJECT, and LOCATION, and the created nodes correspond to some of the data from the COMPANY database with a few modifications; for example, we use EmpId instead of SSN, and we only include a small subset of the data for illustration purposes.

Properties are enclosed in curly brackets { ... }. It is possible that some nodes have multiple labels; for example the same node can be labeled as PERSON and EMPLOYEE and MANAGER by listing all the labels separated by the colon symbol as follows: PERSON:EMPLOYEE:MANAGER.

Having multiple labels is similar to an entity belonging to an entity type (PERSON) plus some subclasses of PERSON (namely EMPLOYEE and MANAGER) in the EER model (see Chapter 4) but can also be used for other purposes.

■ **Relationships and relationship types.** The create statement in (b) shows a few example relationships in Neo4j based on the COMPANY database. The → specifies the direction of the relationship, but the relationship can be traversed in either direction. The relationship types (labels) in (b) are WorksFor, Manager, LocatedIn, and WorksOn; only relationships with the relationship type WorksOn have properties (Hours) in (b).

■ **Paths.** A **path** specifies a traversal of part of the graph. It is typically used as part of a query to specify a pattern, where the query will retrieve from the graph data that matches the pattern. A path is typically specified by a start node, followed by one or more relationships, leading to one or more end nodes that satisfy the pattern.

■ **Optional Schema.** A **schema** is optional in Neo4j. Graphs can be created and used without a schema, but in Neo4j version 2.0, a few schema-related functions were added. The main features related to schema creation involve creating indexes and constraints based on the labels and properties. For example, it is possible to create the equivalent of a key constraint

on a property of a label, so all nodes in the collection of nodes associated with the label must have unique values for that property.

■ **Indexing and node identifiers.** When a node is created, the Neo4j system creates an internal unique system-defined identifier for each node. To retrieve individual nodes using other properties of the nodes efficiently, the user can create **indexes** for the collection of nodes that have a particular label. Typically, one or more of the properties of the nodes in that collection can be indexed. For example, Empid can be used to index nodes with the EMPLOYEE label, Dno to index the nodes with the DEPARTMENT label, and Pno to index the nodes with the PROJECT label.

The Cypher Query Language of Neo4j

Neo4j has a high-level query language, Cypher. There are declarative commands for creating nodes and relationships, as well as for finding nodes and relationships based on specifying patterns. Deletion and modification of data is also possible in Cypher.

A Cypher query is made up of *clauses*. When a query has several clauses, the result from one clause can be the input to the next clause in the query. The Cyber language can specify complex queries and updates on a graph database. We will give a few of examples to illustrate simple Cyber queries in (d).

Examples in Neo4j using the Cypher language. (a) Creating some nodes. (b) Creating some relationships. (c) Basic syntax of Cypher queries. (d) Examples of Cypher queries.

(a) creating some nodes for the COMPANY data (from Figure 5.6):

CREATE (e1: EMPLOYEE, {Empid: '1', Lname: 'Smith', Fname: 'John', Minit: 'B'})
CREATE (e2: EMPLOYEE, {Empid: '2', Lname: 'Wong', Fname: 'Franklin'})
CREATE (e3: EMPLOYEE, {Empid: '3', Lname: 'Zelaya', Fname: 'Alicia'})
CREATE (e4: EMPLOYEE, {Empid: '4', Lname: 'Wallace', Fname: 'Jennifer', Minit: 'S'})
…
CREATE (d1: DEPARTMENT, {Dno: '5', Dname: 'Research'})
CREATE (d2: DEPARTMENT, {Dno: '4', Dname: 'Administration'})
…

CREATE (p1: PROJECT, {Pno: '1', Pname: 'ProductX'})

CREATE (p2: PROJECT, {Pno: '2', Pname: 'ProductY'})

CREATE (p3: PROJECT, {Pno: '10', Pname: 'Computerization'})

CREATE (p4: PROJECT, {Pno: '20', Pname: 'Reorganization'})

…

CREATE (loc1: LOCATION, {Lname: 'Houston'})

CREATE (loc2: LOCATION, {Lname: 'Stafford'})

CREATE (loc3: LOCATION, {Lname: 'Bellaire'})

CREATE (loc4: LOCATION, {Lname: 'Sugarland'})

…

(b) creating some relationships for the COMPANY data (from Figure 5.6):

CREATE (e1) – [: WorksFor] –> (d1)

CREATE (e3) – [: WorksFor] –> (d2)

…

CREATE (d1) – [: Manager] –> (e2)

CREATE (d2) – [: Manager] –> (e4)

…

CREATE (d1) – [: LocatedIn] –> (loc1)

CREATE (d1) – [: LocatedIn] –> (loc3)

CREATE (d1) – [: LocatedIn] –> (loc4)

CREATE (d2) – [: LocatedIn] –> (loc2)

…

CREATE (e1) – [: WorksOn, {Hours: '32.5'}] –> (p1)

CREATE (e1) – [: WorksOn, {Hours: '7.5'}] –> (p2)

CREATE (e2) – [: WorksOn, {Hours: '10.0'}] –> (p1)

CREATE (e2) – [: WorksOn, {Hours: 10.0}] –> (p2)

CREATE (e2) – [: WorksOn, {Hours: '10.0'}] –> (p3)

CREATE (e2) – [: WorksOn, {Hours: 10.0}] –> (p4)

…

Basic simplified syntax of some common Cypher clauses:

Finding nodes and relationships that match a pattern: MATCH <pattern>

Specifying aggregates and other query variables: WITH <specifications>

Specifying conditions on the data to be retrieved: WHERE <condition>

Specifying the data to be returned: RETURN <data>

Ordering the data to be returned: ORDER BY <data>

Limiting the number of returned data items: LIMIT <max number>

Creating nodes: CREATE <node, optional labels and properties>

Creating relationships: CREATE <relationship, relationship type and optional properties>

Deletion: DELETE <nodes or relationships>

Specifying property values and labels: SET <property values and labels>

Removing property values and labels: REMOVE <property values and labels>

(d) Examples of simple Cypher queries:

1. MATCH (d : DEPARTMENT {Dno: '5'}) – [: LocatedIn] → (loc)
RETURN d.Dname , loc.Lname
2. MATCH (e: EMPLOYEE {Empid: '2'}) – [w: WorksOn] → (p)
RETURN e.Ename , w.Hours, p.Pname
3. MATCH (e) – [w: WorksOn] → (p: PROJECT {Pno: 2})
RETURN p.Pname, e.Ename , w.Hours
4. MATCH (e) – [w: WorksOn] → (p)
RETURN e.Ename , w.Hours, p.Pname
ORDER BY e.Ename
5. MATCH (e) – [w: WorksOn] → (p)
RETURN e.Ename , w.Hours, p.Pname
ORDER BY e.Ename
LIMIT 10
6. MATCH (e) – [w: WorksOn] → (p)
WITH e, COUNT(p) AS numOfprojs
WHERE numOfprojs > 2
RETURN e.Ename , numOfprojs
ORDER BY numOfprojs

7. MATCH (e) – [w: WorksOn] → (p)

RETURN e , w, p

ORDER BY e.Ename

LIMIT 10

8. MATCH (e: EMPLOYEE {Empid: '2'})

SET e.Job = 'Engineer'

Neo4j Interfaces and Distributed System Characteristics

Neo4j has other interfaces that can be used to create, retrieve, and update nodes and relationships in a graph database. It also has two main versions: the enterprise edition, which comes with additional capabilities, and the community edition.

■ **Enterprise edition vs. community edition.** Both editions support the Neo4j graph data model and storage system, as well as the Cypher graph query language, and several other interfaces, including a high-performance native API, language drivers for several popular programming languages, such as Java, Python, PHP, and the REST (Representational State Transfer) API. In addition, both editions support ACID properties. The enterprise edition supports additional features for enhancing performance, such as caching and clustering of data and locking.

■ **Graph visualization interface.** Neo4j has a graph visualization interface, so that a subset of the nodes and edges in a database graph can be displayed as a graph. This tool can be used to visualize query results in a graph representation.

■ **Master-slave replication.** Neo4j can be configured on a cluster of distributed system nodes (computers), where one node is designated the master node. The data and indexes are fully replicated on each node in the cluster.

Various ways of synchronizing the data between master and slave nodes can be configured in the distributed cluster.

■ **Caching.** A main memory cache can be configured to store the graph data for improved performance.

■ **Logical logs.** Logs can be maintained to recover from failures.