Module-5: Concurrency Control in Databases

5.8 Introduction to Concurrency Control

- Purpose of Concurrency Control
  - To enforce Isolation (through mutual exclusion) among conflicting transactions.
  - To preserve database consistency through consistency preserving execution of transactions.
  - To resolve read-write and write-write conflicts.
- Example:
  - In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

5.9 Two-Phase Locking Techniques for Concurrency Control

- The concept of locking data items is one of the main techniques used for controlling the concurrent execution of transactions.
- A lock is a variable associated with a data item in the database. Generally there is a lock for each data item in the database.
- A lock describes the status of the data item with respect to possible operations that can be applied to that item.
- It is used for synchronizing the access by concurrent transactions to the database items.
- A transaction locks an object before using it
- When an object is locked by another transaction, the requesting transaction must wait

5.9.1 Types of Locks and System Lock Tables

1. **Binary Locks**

- A **binary lock** can have two **states** or **values:** locked and unlocked (or 1 and 0).
- 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
- We refer to the current value (or state) of the lock associated with item X as **lock**(**X**).
- 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;
```

Fig: 2.1.1 Lock and unlock operations for binary licks.

- The lock_item and unlock_item operations must be implemented as indivisible units that is, no interleaving should be allowed once a lock or unlock operation is started until the operation terminates or the transaction waits
- The wait command within the lock_item(*X*) operation is usually implemented by putting the transaction in a waiting queue for item *X* until *X* is unlocked and the transaction can be granted access to it
- Other transactions that also want to access *X* are placed in the same queue.Hence, the wait command is considered to be outside the lock_item operation.
- It is quite simple to implement a binary lock; all that is needed is a binary-valued variable, LOCK, associated with each data item X in the database
- In its simplest form, 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
- 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*.
  4. A transaction *T* will not issue an unlock_item(*X*) operation unless it already holds the lock on item *X*.

2. Shared/Exclusive (or Read/Write) Locks

- binary locking scheme is too restrictive for database items because at most, one transaction can hold a lock on a given item
- should allow several transactions to access the same item X if they all access X for reading purposes only
- 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
- 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 **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
- Method to implement read/write lock is to
  - keep track of the number of transactions that hold a shared (read) lock on an item in the lock table
  - 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*.

```
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.3

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.

Conversion 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
- 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

5.9.2 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
- Such a transaction can be divided into two phases:
  - **Expanding** or **growing (first) phase**, during which new locks on items can be acquired but none can be released
  - **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.
- Transactions *T*1 and *T*2 in Figure 22.3(a) do not follow the two-phase locking protocol because the write_lock(*X*) operation follows the unlock(*Y*) operation in *T*1, and similarly the write_lock(*Y*) operation follows the unlock(*X*) operation in *T*2.

(a)

T_1	T_2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
$X := X + Y$;	$Y := X + Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

(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

(c)

Time

T_1	T_2
read_lock(Y);	
read_item(Y);	
unlock(Y);	
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);
write_lock(X);	
read_item(X);	
$X := X + Y$;	
write_item(X);	
unlock(X);	

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

- If we enforce two-phase locking, the transactions can be rewritten as *T*1’ and *T*2’ as shown in Figure 22.4.
- Now, the schedule shown in Figure 22.3(c) is not permitted for *T*1_ and *T*2_ (with their modified order of locking and unlocking operations) under the rules of locking because *T*1_ will issue its write_lock(*X*) *before* it unlocks item *Y*; consequently, when *T*2_ issues its read_lock(*X*), it is forced to wait until *T*1_ releases the lock by issuing an unlock (*X*) in the schedule.

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*);

Figure 22.4
Transactions T_1' and T_2', which are the same as T_1 and T_2 in Figure 22.3, but follow the two-phase locking protocol. Note that they can produce a deadlock.

- If every transaction in a schedule follows the two-phase locking protocol, schedule guaranteed to be serializable
- Two-phase locking may limit the amount of concurrency that can occur in a schedule
- Some serializable schedules will be prohibited by two-phase locking protocol

5.10 Variations of Two-Phase Locking

- **Basic 2PL**
  - Technique described previously
- **Conservative (static) 2PL**
  - Requires a transaction to lock all the items it accesses before the transaction begins execution by predeclaring read-set and write-set
  - Its Deadlock-free protocol

- **Strict 2PL**
  - guarantees strict schedules
  - Transaction does not release exclusive locks until after it commits or aborts
  - 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

- **Rigorous 2PL**
  - guarantees strict schedules
  - Transaction does not release any locks until after it commits or aborts
  - easier to implement than strict 2PL

5.11 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.
- A simple example is shown in Figure 22.5(a), where the two transactions *T*1' and *T*2_'are deadlocked in a partial schedule; *T*1' is in the waiting queue for *X*, which is locked by *T*2', while *T*2' is in the waiting queue for *Y*, which is locked by *T*1'. Meanwhile, neither *T*1' nor *T*2' nor any other transaction can access items *X* and *Y*

(a)

Time ↓

T_1'	T_2'
read_lock(*Y*); read_item(*Y*);	
	read_lock(*X*); read_item(*X*);
write_lock(*X*);	
	write_lock(*Y*);

(b)

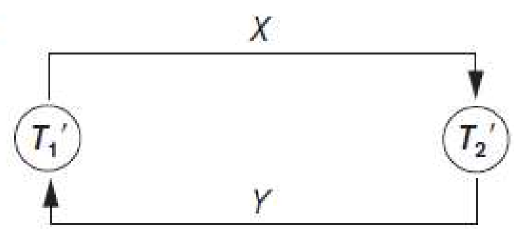

Figure 22.5 Illustrating the deadlock problem (a) A partial schedule of *T*1′ and *T*2′ that is in a state of deadlock (b) A wait-for graph for the partial schedule in (a)

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*. If any of the items cannot be obtained, none of the items are locked. Rather, the transaction waits and then tries again to lock all the items it needs.
- A second protocol, which also limits concurrency, 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 requires that the programmer (or the system) is aware of the chosen order of the items
- Both approaches impractical
- Some of these techniques use the concept of **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 T_1 starts before transaction T_2, then TS(T_1) < TS(T_2).
- The *older* transaction (which starts first) has the *smaller* timestamp value.
- Protocols based on a timestamp
  - Wait-die
  - 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.
- In wait-die, an older transaction is allowed to *wait for a younger transaction*, whereas a younger transaction requesting an item held by an older transaction is aborted and restarted.
- The wound-wait approach does the opposite: A younger transaction is allowed to *wait for an older one*, whereas an older transaction requesting an item held by a younger transaction *preempts* the younger transaction by aborting it.

- Both schemes end up aborting the *younger* of the two transactions (the transaction that started later) that *may be involved* in a deadlock, assuming that this will waste less processing.
- It can be shown that these two techniques are *deadlock-free*, since in wait-die, transactions only wait for younger transactions so no cycle is created.
- Similarly, in wound-wait, transactions only wait for older transactions so no cycle is created.
- Another group of protocols that prevent deadlock do not require timestamps. These include the
    - • no waiting (NW) and
    - • cautious waiting (CW) algorithms
  - **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 actually occur or not.
    - – no transaction ever waits, so no deadlock will occur
    - – this scheme can cause transactions to abort and restart needlessly
  - **cautious waiting**
    - – try to reduce the number of needless aborts/restarts
    - – Suppose that transaction *Ti* tries to lock an item *X* but is not able to do so because *X* is locked by some other transaction *Tj* with a conflicting lock.
    - – The cautious waiting rules are as follows:
      - If *Tj* is not blocked (not waiting for some other locked item), then *Ti* is blocked and allowed to wait; otherwise abort *Ti.*
    - – It can be shown that cautious waiting is deadlock-free, because no transaction will ever wait for another blocked transaction.

5.12 Deadlock Detection.

- A second, more practical approach to dealing with deadlock is **deadlock detection**, where the system checks if a state of deadlock actually exists.
- This solution is attractive if we know there will be little interference among the transactions—that is, if different transactions will rarely access the same items at the same time.

- This can happen if the transactions are short and each transaction locks only a few items, or if the transaction load is light.
- On the other hand, if transactions are long and each transaction uses many items, or if the transaction load is quite heavy, it may be advantageous to use a deadlock prevention scheme.
- 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 *Ti* is waiting to lock an item *X* that is currently locked by a transaction *Tj*, a directed edge (*Ti* → *Tj*) is created in the wait-for graph.
- When *Tj* releases the lock(s) on the items that *Ti* 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.
- One problem with this approach is the matter of determining *when* the system should check for a deadlock.
- One possibility is to check for a cycle every time an edge is added to the wait-for graph, but this may cause excessive overhead.
- Criteria such as the number of currently executing transactions or the period of time several transactions have been waiting to lock items may be used instead to check for a cycle. Figure 22.5(b) shows the wait-for graph for the (partial) schedule shown in Figure 22.5(a).

– If the system is in a state of deadlock, some of the transactions causing the deadlock must be aborted.

– 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, and it should try instead to select transactions that have not made many changes (younger transactions).

- **Timeouts**
  - Another simple scheme to deal with deadlock is the use of **timeouts**.
  - This method is practical because of its low overhead and simplicity.
  - In this method, 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 actually exists or not.

- **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, giving 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 algorithm can use higher priorities for transactions that have been aborted multiple times to avoid this problem.

5.13 Concurrency Control Based on Timestamp Ordering

guarantees serializability using transaction timestamps to order transaction execution for an equivalent serial schedule

5.13.1 Timestamps

- **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.

5.13.2 The Timestamp Ordering Algorithm

- The idea for this scheme is to order 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. This is called **timestamp ordering (TO)**.
- The algorithm must ensure that, for each item accessed by *conflicting Operations* in the schedule, the order in which the item is accessed does not violate the timestamp order.
- To do this, 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.

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.
- If this order is violated, then transaction *T* is aborted and resubmitted to the system as a new transaction with a *new timestamp*.
- If *T* is aborted and rolled back, any transaction *T*1 that may have used a value written by *T* must also be rolled back.

- Similarly, any transaction *T*2 that may have used a value written by *T*1 must also be rolled back, and so on. This effect is known as **cascading rollback** and is one of the problems associated with basic TO, since the schedules produced are not guaranteed to be recoverable.
- An *additional protocol* must be enforced to ensure that the schedules are recoverable, cascadeless, or strict.
- **The basic TO algorithm :**
  - 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 is checked:
     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 is checked:
     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*).

  - Whenever the basic TO algorithm detects two *conflicting operations* that occur in the incorrect order, it rejects the later of the two operations by aborting the transaction that issued it. The schedules produced by basic TO are hence guaranteed to be *conflict serializable*

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* that 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.
- To implement this algorithm, it is necessary to simulate the locking of an item *X* that has been written by transaction *T*' until *T*' is either committed or aborted. This algorithm *does not cause deadlock*, since *T* waits for *T*' only if TS(*T*) > TS(*T*_).

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).

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*).

5.14 Multiversion Concurrency Control Techniques

- Other protocols for concurrency control keep the old values of a data item when the item is updated. These are known as **multiversion concurrency control**, because several versions (values) of an item are maintained
- When a transaction requires access to an item, an *appropriate* version is chosen to maintain the serializability of the currently executing schedule, if possible.
- The idea 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 are retained
- An obvious drawback of multiversion techniques is that more storage is needed to maintain multiple versions of the database items

5.14.1 Multiversion Technique Based on Timestamp Ordering

- In this method, several versions $X_1, X_2, ..., X_k$ of each data item X are maintained.
- For *each version*, the value of version X_i and the following two timestamps 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.
- Whenever a transaction T is allowed to execute a write_item(X) operation, a new version X_{k+1} of item X is created, with both the write_TS(X_{k+1}) and the read_TS(X_{k+1}) set to TS(T)
- Correspondingly, when a transaction T is allowed to read the value of version X_i, the value of read_TS(X_i) is set to the larger of the current read_TS(X_i) 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(X_i) of all versions of X that is also *less than or equal to* TS(T), and read_TS(X_i) > TS(T), then abort and roll back transaction T; otherwise, create a new version X_j of X with read_TS(X_j) = write_TS(X_j) = TS(T).
  2. If transaction T issues a read_item(X) operation, find the version i of X that has the highest write_TS(X_i) of all versions of X that is also *less than or equal to* TS(T); then return the value of X_i to transaction T, and set the value of read_TS(X_i) to the larger of TS(T) and the current read_TS(X_i).

5.14.2 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*
- Hence, the state of LOCK(X) for an item X can be one of read-locked, writelocked, certify-locked, or unlocked
- We can describe the relationship between read and write locks in the standard scheme by means of the **lock compatibility table** shown in Figure 22.6(a)
- An entry of *Yes* means that if a transaction T holds the type of lock specified in the column header on item X and if transaction T_ requests the type of lock specified in

the row header on the same item *X*, then *T_ can obtain the lock* because the locking modes are compatible

(a)

	Read	Write
Read	Yes	No
Write	No	No

(b)

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

Figure 22.6: Lock compatibility tables. (a) A compatibility table for read/write locking scheme. (b) A compatibility table for read/write/certify locking scheme.

- On the other hand, an entry of *No* in the table indicates that the locks are not compatible, so *T' must wait* until *T releases* the lock
- 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 must always have been written by some committed transaction
- The second version *X'* is created when a transaction *T* acquires a write lock on the item

5.15Validation (Optimistic) Concurrency Control Techniques

- In **optimistic concurrency control techniques**, also known as **validation** or **certification techniques**, *no checking* is done while the transaction is executing
- In this scheme, updates in the transaction are *not* applied directly to the database items until the transaction reaches its end

- 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 idea behind optimistic concurrency control is to do all the checks at once; hence, transaction execution proceeds with a minimum of overhead until the validation phase is reached
- The techniques are called *optimistic* because they assume that little interference will occur and hence that there is no need to do checking during transaction execution.
- The validation phase for *Ti* checks that, for *each* such transaction *Tj* that is either 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.

5.16 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

- The granularity can affect the performance of concurrency control and recovery

5.16.1 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
- 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 record *B* must lock the whole disk block *X* that contains *B* because a lock is associated with the whole data item (block). Now, if another transaction *S* wants to lock a different record *C* that happens to reside in the same block *X* in a conflicting lock mode, it is forced to wait. If the data item size was a single record, transaction *S* would be able to proceed, because it would be locking a different data item (record).
- 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. More lock and unlock operations will be performed, causing a higher overhead
- The best item size *depends on the types of transactions involved*.
- If a typical transaction accesses a small number of records, it is advantageous to have the data item granularity be one record
- On the other hand, if a transaction typically accesses many records in the same file, it may be better to have block or file granularity so that the transaction will consider all those records as one (or a few) data items

5.16.2 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 different for various mixes of transactions
- Figure 22.7 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, where a lock can be requested at any level

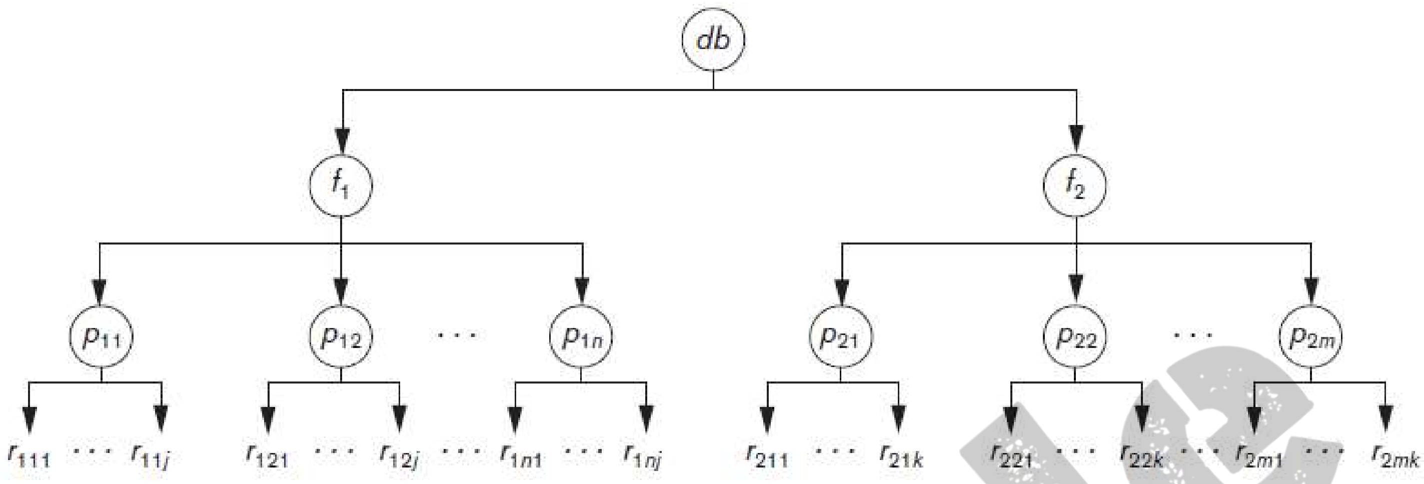

Figure 22.7 A granularity hierarchy for illustrating multiple granularity level locking

- 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 compatibility table of the three intention locks, and the shared and exclusive locks, is shown in Figure 22.8.

IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

Figure 22.8: Lock compatibility matrix for multiple granularity locking.

- The **multiple granularity locking (MGL)** protocol consists of the following rules:
    1. The lock compatibility (based on Figure 22.8) 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*.
- The multiple granularity level protocol is especially suited when processing a mix of transactions that include
  (1) short transactions that access only a few items (records or fields) and
  (2) long transactions that access entire files.

24.1 Introduction to NOSQL Systems

24.1.1 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. Although newer relational systems do have more complex object-relational modeling options (see Chapter 12), they still require schemas, which are not required by many of the NOSQL systems.

As another example, 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. Some of the data for this type of application is not suitable for a traditional relational system and typically needs multiple types of databases and data storage systems.

Some of the organizations that were faced with these data management and storage applications decided to develop their own systems:

- 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. Apache Hbase is an open source NOSQL system based on similar concepts. Google's innovation led to the category of NOSQL systems known as **column-based** or **wide column** stores; they are also sometimes referred to as **column family** stores.
- Amazon developed a NOSQL system called **DynamoDB** that is available through Amazon's cloud services. This innovation led to the category known as **key-value** data stores or sometimes **key-tuple** or **key-object** data stores.
- Facebook developed a NOSQL system called **Cassandra**, which is now open source and known as Apache Cassandra. This NOSQL system uses concepts from both key-value stores and column-based systems.
- 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.

- Some NOSQL systems, such as **OrientDB**, combine concepts from many of the categories discussed above.
- In addition to the newer types of NOSQL systems listed above, it is also possible to classify database systems based on the object model (see Chapter 12) or on the native XML model (see Chapter 13) as NOSQL systems, although they may not have the high-performance and replication characteristics of the other types of NOSQL systems.

These are just a few examples of NOSQL systems that have been developed. There are many systems, and listing all of them is beyond the scope of our presentation.

24.1.2 Characteristics of NOSQL Systems

We now discuss the characteristics of many NOSQL systems, and how these systems differ from traditional SQL systems. We divide the characteristics into two categories—those related to distributed databases and distributed systems, and those related to data models and query languages.

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. Scalability is another important characteristic, because many of the applications that use NOSQL systems tend to have data that keeps growing in volume. High performance is another required characteristic, whereas serializable consistency may not be as important for some of the NOSQL applications. We discuss some of these characteristics next.

1. **Scalability:** As we discussed in Section 23.1.4, there 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. In NOSQL systems, horizontal scalability is employed while the system is operational, so techniques for distributing the existing data among new nodes without interrupting system operation are necessary. We will discuss some of these techniques in Sections 24.3 through 24.6 when we discuss specific systems.
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 can 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 (see Section 23.3). Many NOSQL applications do not require serializable

consistency, so more relaxed forms of consistency known as **eventual consistency** are used. We discuss this in more detail in Section 24.2.

3. **Replication Models:** Two major replication models are used in NOSQL systems: master-slave and master-master replication. **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). For read, the master-slave paradigm can be configured in various ways. One configuration requires all reads to also be at the master copy, so this would be similar to the primary site or primary copy methods of distributed concurrency control (see Section 23.3.1), with similar advantages and disadvantages. Another configuration would allow reads at the slave copies but would not guarantee that the values are the latest writes, since writes to the slave nodes can be done after they are applied to the master copy. 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. A reconciliation method to resolve conflicting write operations of the same data item at different nodes must be implemented as part of the master-master replication scheme.

4. **Sharding 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** ; see Section 23.2) of the file records is often employed in NOSQL systems. This serves to distribute the load of accessing the file records to multiple nodes. The combination of sharding the file records and replicating the shards works in tandem to improve load balancing as well as data availability. We will discuss some of the sharding techniques in Sections 24.3 through 24.6 when we discuss specific systems.

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. The object key is similar to the concept of object id (see Section 12.1). In **hashing**, a hash function $h(K)$ is applied to the key K, and the location of the object with key K is determined by the value of $h(K)$. In **range partitioning**, the location is determined via a range of key values; for example, location i would hold the objects whose key values K are in the range $Ki_{\min} \leq K \leq Ki_{\max}$. In applications that require range queries, where multiple objects within a range of key values are retrieved, range partitioned is preferred. Other indexes can also be used to locate objects based on attribute conditions different from the key K. We

will discuss some of the hashing, partitioning, and indexing techniques in Sections 24.3 through 24.6 when we discuss specific systems.

NOSQL characteristics related to data models and query languages. NOSQL systems emphasize performance and flexibility over modeling 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, self-describing data (see Section 13.1). 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; see Chapter 13). JSON is used in several NOSQL systems, but other methods for describing semi-structured data can also be used. We will discuss JSON in Section 24.3 when we present document-based NOSQL systems.
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. We will discuss this aspect in Section 24.5 when we present column-based NOSQL systems.

In the next section, we give an overview of the various categories of NOSQL systems.

24.1.3 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 (a form of vertical partitioning; see Section 23.2), 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:** These systems were discussed in Chapter 12.
7. **XML databases:** We discussed XML in Chapter 13.

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.

The rest of this chapter is organized as follows. In each of Sections 24.3 through 24.6, we will discuss one of the four main categories of NOSQL systems, and elaborate further on which characteristics each category focuses on. Before that, in Section 24.2, we discuss in more detail the concept of eventual consistency, and we discuss the associated CAP theorem.

24.2 The CAP Theorem

When we discussed concurrency control in distributed databases in Section 23.3, we assumed that the distributed database system (DDBS) is required to enforce the ACID properties (atomicity, consistency, isolation, durability) of transactions that are running concurrently (see Section 20.3). In a system with data replication, concurrency control becomes more complex because there can be multiple copies of each data item. So if an update is applied to one copy of an item, it must be applied to all other copies in a consistent manner. The possibility exists that one copy of an item X is updated by a transaction T_1 whereas another copy is updated by a transaction T_2, so two inconsistent copies of the same item exist at two different nodes in the distributed system. If two other transactions T_3 and T_4 want to read X, each may read a different copy of item X.

We saw in Section 23.3 that there are distributed concurrency control methods that do not allow this inconsistency among copies of the same data item, thus enforcing serializability and hence the isolation property in the presence of replication. However, these techniques often come with high overhead, which would defeat the purpose of creating multiple copies to improve performance and availability in distributed database systems such as NOSQL. In the field of distributed systems, there are various levels of consistency among replicated data items, from weak consistency to strong consistency. Enforcing serializability is considered the strongest form of consistency, but it has high overhead so it can reduce performance of read and write operations and hence adversely affect system performance.

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. In Sections 24.3 through 24.6, we will discuss some of the consistency models used in specific NOSQL systems.

The next four sections of this chapter discuss the characteristics of the four main categories of NOSQL systems. We discuss document-based NOSQL systems in Section 24.3, and we use MongoDB as a representative system. In Section 24.4, we discuss

NOSQL systems known as key-value stores. In Section 24.5, we give an overview of column-based NOSQL systems, with a discussion of Hbase as a representative system. Finally, we introduce graph-based NOSQL systems in Section 24.6.

24.3 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 objects* (see Section 12.3) or XML documents (see Chapter 13), 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** (see Section 13.1). 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. The system basically extracts the data element names from the self-describing documents in the collection, and the user can request that the system create indexes on some of the data elements. Documents can be specified in various formats, such as XML (see Chapter 13). A popular language to specify documents in NOSQL systems is **JSON** (JavaScript Object Notation).

There are many document-based NOSQL systems, including MongoDB and CouchDB, among many others. We will give an overview of MongoDB in this section. It is important to note that different systems can use different models, languages, and implementation methods, but giving a complete survey of all document-based NOSQL systems is beyond the scope of our presentation.

24.3.1 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 book. 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 (see Figures 5.5 and 5.6):

```
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. There are other collection options, but we will not discuss them here.

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.

A collection does not have a schema. The structure of the data fields in documents is chosen based on how documents will be accessed and used, and the user can choose a normalized design (similar to normalized relational tuples) or a denormalized design (similar to XML documents or complex objects). Interdocument references can be specified by storing in one document the ObjectId or ObjectIds of other related documents. Figure 24.1(a) shows a simplified MongoDB document showing some of the data from Figure 5.6 from the COMPANY database example that is used throughout the book. In our example, the _id values are user-defined, and the documents whose _id starts with P (for project) will be stored in the "project" collection, whereas those whose _id starts with W (for worker) will be stored in the "worker" collection.

In Figure 24.1(a), the workers information is *embedded in the project document*; so there is no need for the "worker" collection. This is known as the *denormalized pattern*, which is similar to creating a complex object (see Chapter 12) or an XML document (see Chapter 13). A list of values that is enclosed in *square brackets* […] within a document represents a field whose value is an **array**.

Another option is to use the design in Figure 24.1(b), where *worker references* are embedded in the project document, but the worker documents themselves are stored in a separate "worker" collection. A third option in Figure 24.1(c) would use a normalized design, similar to First Normal Form relations (see Section 14.3.4). The choice of which design option to use depends on how the data will be accessed.

It is important to note that the simple design in Figure 24.1(c) *is not the general normalized design* for a many-to-many relationship, such as the one between employees and projects; rather, we would need three collections for "project", "employee", and "works_on", as we discussed in detail in Section 9.1. Many of the design tradeoffs that were discussed in Chapters 9 and 14 (for first normal form relations and for ER-to-relational mapping options), and Chapters 12 and 13 (for complex objects and XML) are applicable for choosing the appropriate design for document structures

Figure 24.1
Example of simple documents in MongoDB.
(a) Denormalized document design with embedded subdocuments.
(b) Embedded array of document references.
(c) Normalized documents.

(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
}
```

(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 } ] )
```

Figure 24.1 (continued)
Example of simple documents in MongoDB. (d) Inserting the documents in Figure 24.1(c) into their collections.

and document collections, so we will not repeat the discussions here. In the design in Figure 24.1(c), an EMPLOYEE who works on several projects would be represented by *multiple worker documents* with different _id values; each document would represent the employee *as worker for a particular project*. This is similar to the design decisions for XML schema design (see Section 13.6). However, it is again important to note that the typical document-based system *does not have a schema*, so the design rules would have to be followed whenever individual documents are inserted into a collection.

24.3.2 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. For a full discussion of the MongoDb CRUD operations, see the MongoDB online documentation in the chapter references.

24.3.3 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. We discussed the atomicity and consistency properties of transactions in Section 20.3, and the two-phase commit protocol in Section 22.6.

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. We discussed elections in distributed systems in Section 23.3.1.

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 (see Section 23.1.4), 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 (we discussed hashing and its advantages and disadvantages in Section 16.8). 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.

There are many additional details about the distributed system architecture and components of MongoDB, but a full discussion is outside the scope of our presentation. MongoDB also provides many other services in areas such as system administration, indexing, security, and data aggregation, but we will not discuss these features here. Full documentation of MongoDB is available online (see the bibliographic notes).

24.4 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 (see Section 13.1). 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.

24.4.1 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.

24.4.2 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 $Hmax = 2^{n-1}$, 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 (see Figure 24.2). 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 24.2(a), 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 24.2(a) indicates which (k, v) items are placed in which nodes based on the *h(k)* values.

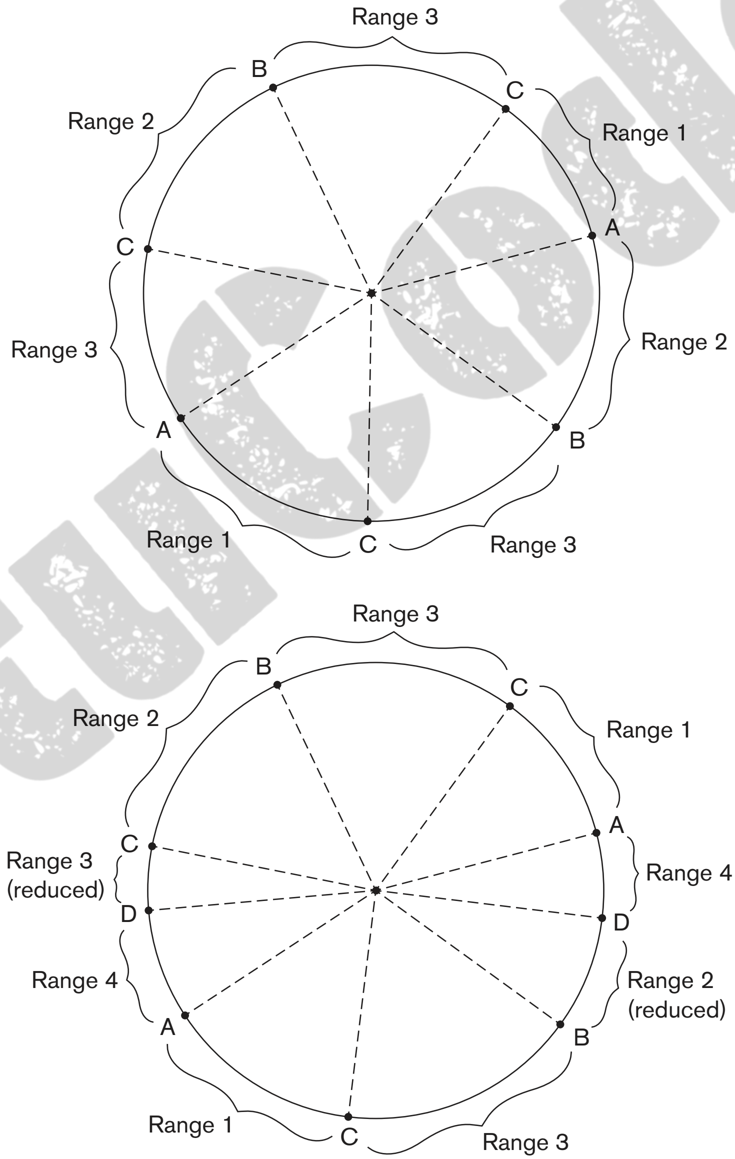

Figure 24.2
Example of consistent hashing. (a) Ring having three nodes A, B, and C, with C having greater capacity. The *h*(*K*) values that map to the circle points in *range 1* have their (k, v) items stored in node A, *range 2* in node B, *range 3* in node C. (b) Adding a node D to the ring. Items in *range 4* are moved to the node D from node B (*range 2* is reduced) and node C (*range 3* is reduced).

- The $h(k)$ values that fall in the parts of the circle marked as *range 1* in Figure 24.2(a) 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 24.2(b), 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 24.2(b)) would be migrated to node D. 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. When a node fails, its load of data items can be distributed to the other existing nodes whose labels follow the labels of the failed node in the ring. And nodes with higher capacity can have more locations on the ring, as illustrated by node C in Figure 24.2(a), and thus store more items than smaller-capacity nodes.
- **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.

24.4.3 Examples of Other Key-Value Stores

In this section, we briefly review three other key-value stores. It is important to note that there are many systems that can be classified in this category, and we can only mention a few of these systems.

Oracle key-value store. Oracle has one of the well-known SQL relational database systems, and Oracle also offers a system based on the key-value store concept; this system is called the **Oracle NoSQL Database**.

Redis key-value cache and store. **Redis** differs from the other systems discussed here because it caches its data in main memory to further improve performance. It offers master-slave replication and high availability, and it also offers persistence by backing up the cache to disk.

Apache Cassandra. **Cassandra** is a NOSQL system that is not easily categorized into one category; it is sometimes listed in the column-based NOSQL category (see Section 24.5) or in the key-value category. If offers features from several NOSQL categories and is used by Facebook as well as many other customers.

24.5 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. BigTable uses the **Google File System** (**GFS**) for data storage and distribution. An open source system known as **Apache Hbase** is somewhat similar to Google BigTable, but it typically uses **HDFS** (**Hadoop Distributed File System**) for data storage. HDFS is used in many cloud computing applications, as we shall discuss in Chapter 25. 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 (see Section 24.4) 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. We discuss these concepts in more detail next as they are realized in Apache Hbase.

24.5.1 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 (this is similar to the concept of attribute versioning in temporal databases, which we shall discuss in Section 26.2). 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. We discuss the Hbase data modeling concepts and define the terminology next. 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. Figure 24.3(a) 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 *self-describing* since any column qualifier name can be used in a new row of data (see Figure 24.3(b)). 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* (see Section 23.2), 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

Figure 24.3
Examples in Hbase. (a) Creating a table called EMPLOYEE with three column families: Name, Address, and 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). (c) Some CRUD operations of Hbase.

(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>
```

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.

24.5.2 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 in Figure 24.3(c).

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.

24.5.3 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.

We will not cover the many additional details about the distributed system architecture and components of Hbase; a full discussion is outside the scope of our presentation. Full documentation of Hbase is available online (see the bibliographic notes).

24.6 NOSQL Graph Databases and Neo4j

Another category of NOSQL systems is known as **graph databases** or **graph-oriented 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. We will discuss the Neo4j data model in Section 24.6.1, and give an introduction to the Neo4j querying capabilities in Section 24.6.2. Section 24.6.3 gives an overview of the distributed systems and some other characteristics of Neo4j.

24.6.1 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 (see Chapters 3 and 4) , 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.

Figure 24.4(a) 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 Figure 24.4(a), the node labels are EMPLOYEE, DEPARTMENT, PROJECT, and LOCATION, and the created nodes correspond to some of the data from the COMPANY database in Figure 5.6 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.** Figure 24.4(b) shows a few example relationships in Neo4j based on the COMPANY database in Figure 5.6. The → specifies the direction of the relationship, but the relationship can be traversed in either direction. The relationship types (labels) in Figure 24.4(b) are WorksFor, Manager, LocatedIn, and WorksOn; only relationships with the relationship type WorksOn have properties (Hours) in Figure 24.4(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. It is somewhat similar to the concepts of path expressions that we discussed in Chapters 12 and 13 in the context of query languages for object databases (OQL) and XML (XPath and XQuery).
- **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.

24.6.2 The Cypher Query Language of Neo4j

Neo4j has a high-level query language, Cypher. There are declarative commands for creating nodes and relationships (see Figures 24.4(a) and (b)), as well as for finding nodes and relationships based on specifying patterns. Deletion and modification of data is also possible in Cypher. We introduced the CREATE command in the previous section, so we will now give a brief overview of some of the other features of 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. We will give a flavor of the language by discussing some of the clauses using examples. Our presentation is not meant to be a detailed presentation on Cypher, just an introduction to some of the languages features. Figure 24.4(c) summarizes some of the main clauses that can be part of a Cyber 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 Figure 24.4(d).

Figure 24.4
Examples in Neo4j using the Cypher language. (a) Creating some nodes. (b) Creating some relationships.

(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)
…
```

Figure 24.4 (continued)
Examples in Neo4j using the Cypher language. (c) Basic syntax of Cypher queries. (d) Examples of Cypher queries.

(c) 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'
```

Query 1 in Figure 24.4(d) shows how to use the MATCH and RETURN clauses in a query, and the query retrieves the locations for department number 5. Match specifies the *pattern* and the *query variables* (*d* and *loc*) and RETURN specifies the query result to be retrieved by refering to the query variables. Query 2 has three variables (*e*, *w*, and *p*), and returns the projects and hours per week that the employee with

Empid = 2 works on. Query 3, on the other hand, returns the employees and hours per week who work on the project with Pno = 2. Query 4 illustrates the ORDER BY clause and returns all employees and the projects they work on, sorted by Ename. It is also possible to limit the number of returned results by using the LIMIT clause as in query 5, which only returns the first 10 answers.

Query 6 illustrates the use of WITH and aggregation, although the WITH clause can be used to separate clauses in a query even if there is no aggregation. Query 6 also illustrates the WHERE clause to specify additional conditions, and the query returns the employees who work on more than two projects, as well as the number of projects each employee works on. It is also common to return the nodes and relationships themselves in the query result, rather than the property values of the nodes as in the previous queries. Query 7 is similar to query 5 but returns the nodes and relationships only, and so the query result can be displayed as a graph using Neo4j's visualization tool. It is also possible to add or remove labels and properties from nodes. Query 8 shows how to add more properties to a node by adding a Job property to an employee node.

The above gives a brief flavor for the Cypher query language of Neo4j. The full language manual is available online (see the bibliographic notes).

24.6.3 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. We discuss some of the additional features of Neo4j in this subsection.

- **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.