Module 17: Transactions

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Outline

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g. transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

Example of Fund Transfer

- Transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
- **Atomicity requirement**
  - If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Example of Fund Transfer (Cont.)

- **Consistency requirement** in above example:
  - The sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer (Cont.)

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

T1	T2
1. **read**(A)	
2. $A := A - 50$	
3. **write**(A)	
	read(A), read(B), print(A+B)
4. **read**(B)	
5. $B := B + 50$	
6. **write**(B	

- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items.To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** -- after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction
    - can be done only if no internal logical error
  - kill the transaction
- **Committed** – after successful completion.

Transaction State (Cont.)

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
    - e.g., one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
    - Will study in Chapter 15, after studying notion of correctness of concurrent executions.

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - By default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedule 1

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- A serial schedule in which T_1 is followed by T_2 :

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedule 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of (A + B).

T_1	T_2
read (A)	
A := A – 50	
	read (A)
	$temp$:= A * 0.1
	A := A - $temp$
	write (A)
	read (B)
write (A)	
read (B)	
B := B + 50	
write (B)	
commit	
	B := B + $temp$
	write (B)
	commit

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus, serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. **conflict serializability**
  2. **view serializability**

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

- Instructions I_i and I_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.
  1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j don't conflict.
  2. I_i = **read**(Q), I_j = **write**(Q). They conflict.
  3. I_i = **write**(Q), I_j = **read**(Q). They conflict
  4. I_i = **write**(Q), I_j = **write**(Q). They conflict
- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them.
  - If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

- If a schedule *S* can be transformed into a schedule *S'* by a series of swaps of non-conflicting instructions, we say that *S* and *S'* are **conflict equivalent**.
- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

View Serializability

- Let *S* and *S'* be two schedules with the same set of transactions. *S* and *S'* are **view equivalent** if the following three conditions are met, for each data item *Q*,
  1. If in schedule S, transaction T_i reads the initial value of *Q*, then in schedule *S'* also transaction T_i must read the initial value of *Q*.
  2. If in schedule S transaction T_i executes **read**(*Q*), and that value was produced by transaction T_j (if any), then in schedule *S'* also transaction T_i must read the value of *Q* that was produced by the same **write**(Q) operation of transaction T_j .
  3. The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S'*.
- As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

View Serializability (Cont.)

- A schedule *S* is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes**.

Other Notions of Serializability

- The schedule below produces same outcome as the serial schedule < T_1, T_5 >, yet is not conflict equivalent or view equivalent to it.

T_1	T_5
read (A)	
A := A – 50	
write (A)	
	read (B)
	B := B - 10
	write (B)
read (B)	
B := B + 50	
write (B)	
	read (A)
	A := A + 10
	write (A)

- Determining such equivalence requires analysis of operations other than read and write.

Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- **Example 1**

Test for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$
    - Are there others?

(a)

(b)

(c)

Test for View Serializability

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.
  - Thus. existence of an efficient algorithm is *extremely* unlikely.
- However practical algorithms that just check some **sufficient conditions** for view serializability can still be used.

Recoverable Schedules

Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule** — if a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i appears before the commit operation of T_j.
- The following schedule (Schedule 11) is not recoverable

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

- If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

If T_{10} fails, T_{11} and T_{12} must also be rolled back.

- Can lead to the undoing of a significant amount of work

Cascadeless Schedules

- **Cascadeless schedules** — cascading rollbacks cannot occur;
  - For each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.

Concurrency Control (Cont.)

- Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, and preferably cascadeless.
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency.
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.
- Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.

Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless .
- Concurrency control protocols (generally) do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids non-serializable schedules.
  - We study such protocols in Chapter 16.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g., a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g., database statistics computed for query optimization can be approximate (why?)
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read.
  - Repeated reads of same record must return same value.
  - However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read.
  - Successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.

Levels of Consistency

- Lower degrees of consistency useful for gathering approximate information about the database
- Warning: some database systems do not ensure serializable schedules by default
- E.g., Oracle (and PostgreSQL prior to version 9) by default support a level of consistency called snapshot isolation (not part of the SQL standard)

Transaction Definition in SQL

- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - E.g., in JDBC -- connection.setAutoCommit(false);
- Isolation level can be set at database level
- Isolation level can be changed at start of transaction
    - E.g. In SQL **set transaction isolation level serializable**
    - E.g. in JDBC -- connection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE)

Implementation of Isolation Levels

Overview

- Locking
  - Lock on whole database vs lock on items
  - How long to hold lock?
  - Shared vs exclusive locks
- Timestamps
  - Transaction timestamp assigned e.g. when a transaction begins
  - Data items store two timestamps
    - Read timestamp
    - Write timestamp
  - Timestamps are used to detect out of order accesses
- Multiple versions of each data item
  - Allow transactions to read from a “snapshot” of the database

Transactions as SQL Statements

- E.g. Transaction 1:
  select *ID, name* **from** *instructor* **where** *salary* > 90000
- Transaction 2:
  insert into *instructor* **values** ('11111', 'James', 'Marketing', 100000)
- Suppose
  - T1 starts, finds tuples salary > 90000 using index and locks them
  - And then T2 executes.
  - Do T1 and T2 conflict? Does tuple level locking detect the conflict?
  - Instance of the **phantom phenomenon**
- Also consider T3 below, with Wu's salary = 90000
  update *instructor*
  set *salary* = *salary* * 1.1
  where *name* = 'Wu'
- Key idea: Detect "**predicate**" conflicts, and use some form of "**predicate locking**"

End of Chapter 17

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Chapter 18 : Concurrency Control

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Outline

- Lock-Based Protocols
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiple Granularity
- Multiversion Schemes
- Insert and Delete Operations
- Concurrency in Index Structures

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
  1. **exclusive** *(X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. **shared** *(S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-Based Protocols (Cont.)

- **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item,
- But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.

Schedule With Lock Grants

- Grants omitted in rest of chapter
  - Assume grant happens just before the next instruction following lock request
- This schedule is not serializable (why?)
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks.
- Locking protocols enforce serializability by restricting the set of possible schedules.

T_1	T_2	concurrency-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

Deadlock

- Consider the partial schedule

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

- Neither T_3 nor T_4 can make progress — executing **lock-S***(B)* causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X***(A)* causes T_3 to wait for T_4 to release its lock on *A*.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

Deadlock (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

The Two-Phase Locking Protocol

- A protocol which ensures conflict-serializable schedules.
- Phase 1: **Growing Phase**
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: **Shrinking Phase**
  - Transaction may release locks
  - Transaction may not obtain locks

- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e., the point where a transaction acquired its final lock).

The Two-Phase Locking Protocol (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks
- Extensions to basic two-phase locking needed to ensure recoverability of freedom from cascading roll-back
  - **Strict two-phase locking:** a transaction must hold all its exclusive locks till it commits/aborts.
    - Ensures recoverability and avoids cascading roll-backs
  - **Rigorous two-phase locking**: a transaction must hold *all* locks till commit/abort.
    - Transactions can be serialized in the order in which they commit.
- Most databases implement rigorous two-phase locking, *but refer to it as simply two-phase locking*

The Two-Phase Locking Protocol (Cont.)

- Two-phase locking is not a necessary condition for serializability
  - There are conflict serializable schedules that cannot be obtained if the two-phase locking protocol is used.
- In the absence of extra information (e.g., ordering of access to data), two-phase locking is necessary for conflict serializability *in the following sense*:
  - *Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable.*

T_1	T_2
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
unlock(B)	
	lock-S(A)
	read(A)
	unlock(A)
	lock-S(B)
	read(B)
	unlock(B)
	display($A + B$)
lock-X(A)	
read(A)	
$A := A + 50$	
write(A)	
unlock(A)	

Locking Protocols

- Given a locking protocol (such as 2PL)
  - A schedule S is **legal** under a locking protocol if it can be generated by a set of transactions that follow the protocol
  - A protocol **ensures** serializability if all legal schedules under that protocol are serializable

Lock Conversions

- Two-phase locking protocol with lock conversions:
  - Growing Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can **convert** a lock-S to a lock-X (**upgrade**)
  - Shrinking Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (**downgrade**)
- This protocol ensures serializability

Automatic Acquisition of Locks

- A transaction T_i issues the standard read/write instruction, without explicit locking calls.
- The operation **read**(D) is processed as:

if T_i has a lock on D

then

read(D)

else begin

if necessary wait until no other transaction has a **lock-X** on D

grant T_i a **lock-S** on D;

read(D)

end

Automatic Acquisition of Locks (Cont.)

- **write**(*D*) is processed as:

```
if Ti has a  lock-X on D
  then
    write(D)
  else begin
      if necessary wait until no other trans. has any lock on D,
      if Ti has a lock-S on D
          then
             upgrade lock on D  to lock-X
         else
             grant Ti a lock-X on D
         write(D)
   end;
```

- **All locks are released after commit or abort**

Implementation of Locking

- A **lock manager** can be implemented as a separate process
- Transactions can send lock and unlock requests as messages
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
  - The requesting transaction waits until its request is answered
- The lock manager maintains an in-memory data-structure called a **lock table** to record granted locks and pending requests

Lock Table

- Dark rectangles indicate granted locks, light colored ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager may keep a list of locks held by each transaction, to implement this efficiently

Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering $\rightarrow$ on the set **D** = $\{d_1, d_2, ..., d_h\}$ of all data items.
  - If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j.
  - Implies that the set **D** may now be viewed as a directed acyclic graph, called a *database graph*.
- The *tree-protocol* is a simple kind of graph protocol.

Tree Protocol

Tree protocol:

1. Only exclusive locks are allowed.
2. The first lock by T_i may be on any data item. Subsequently, a data Q can be locked by T_i only if the parent of Q is currently locked by T_i.
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i

Graph-Based Protocols (Cont.)

- The tree protocol ensures conflict serializability as well as freedom from deadlock.
- Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.
  - Shorter waiting times, and increase in concurrency
  - Protocol is deadlock-free, no rollbacks are required
- Drawbacks
  - Protocol does not guarantee recoverability or cascade freedom
    - Need to introduce commit dependencies to ensure recoverability
  - Transactions may have to lock data items that they do not access.
    - increased locking overhead, and additional waiting time
    - potential decrease in concurrency
- Schedules not possible under two-phase locking are possible under the tree protocol, and vice versa.

Deadlock Handling

- System is **deadlocked** if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Deadlock Handling

- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies:
  - Require that each transaction locks all its data items before it begins execution (pre-declaration).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

More Deadlock Prevention Strategies

- **wait-die** scheme — non-preemptive
  - Older transaction may wait for younger one to release data item.
  - Younger transactions never wait for older ones; they are rolled back instead.
  - A transaction may die several times before acquiring a lock
- **wound-wait** scheme — preemptive
  - Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it.
  - Younger transactions may wait for older ones.
  - Fewer rollbacks than *wait-die* scheme.
- In both schemes, a rolled back transactions is restarted with its original timestamp.
  - Ensures that older transactions have precedence over newer ones, and starvation is thus avoided.

Deadlock prevention (Cont.)

- **Timeout-Based Schemes**:
  - A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - Ensures that deadlocks get resolved by timeout if they occur
  - Simple to implement
  - But may roll back transaction unnecessarily in absence of deadlock
    - difficult to determine good value of the timeout interval.
  - Starvation is also possible

Deadlock Detection

- **Wait-for graph**
  - *Vertices:* transactions
  - *Edge from* $T_i \rightarrow T_j$. : if T_i is waiting for a lock held in conflicting mode by T_j
- The system is in a deadlock state if and only if the wait-for graph has a cycle.
- Invoke a deadlock-detection algorithm periodically to look for cycles.

Wait-for graph without a cycle

Wait-for graph with a cycle

Deadlock Recovery

- When deadlock is detected :
  - Some transaction will have to rolled back (made a **victim**) to break deadlock cycle.
    - Select that transaction as victim that will incur minimum cost
  - Rollback -- determine how far to roll back transaction
    - **Total rollback**: Abort the transaction and then restart it.
    - **Partial rollback**: Roll back victim transaction only as far as necessary to release locks that another transaction in cycle is waiting for
- Starvation can happen (why?)
  - One solution: oldest transaction in the deadlock set is never chosen as victim

Multiple Granularity

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
- Can be represented graphically as a tree (but don't confuse with tree-locking protocol)
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents in the same mode.
- Granularity of locking (level in tree where locking is done):
  - **Fine granularity** (lower in tree): high concurrency, high locking overhead
  - **Coarse granularity** (higher in tree): low locking overhead, low concurrency

Example of Granularity Hierarchy

The levels, starting from the coarsest (top) level are

- *database*
- *area*
- *file*
- *record*

Intention Lock Modes

- In addition to S and X lock modes, there are three additional lock modes with multiple granularity:
  - ***intention-shared*** (IS): indicates explicit locking at a lower level of the tree but only with shared locks.
  - ***intention-exclusive*** (IX): indicates explicit locking at a lower level with exclusive or shared locks
  - ***shared and intention-exclusive*** (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.
- intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Compatibility Matrix with Intention Lock Modes

- The compatibility matrix for all lock modes is:

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Multiple Granularity Locking Scheme

- Transaction T_i can lock a node Q, using the following rules:
  1. The lock compatibility matrix must be observed.
  2. The root of the tree must be locked first, and may be locked in any mode.
  3. A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
  4. A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
  5. T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
  6. T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.
- Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.
- **Lock granularity escalation**: in case there are too many locks at a particular level, switch to higher granularity S or X lock

Insert/Delete Operations and Predicate Reads

- Locking rules for insert/delete operations
  1. An exclusive lock must be obtained on an item before it is deleted
  2. A transaction that inserts a new tuple into the database is automatically given an X-mode lock on the tuple
- Ensures that
  - reads/writes conflict with deletes
  - Inserted tuple is not accessible by other transactions until the transaction that inserts the tuple commits

Phantom Phenomenon

- Example of **phantom phenomenon**.
  - A transaction T1 that performs **predicate read** (or scan) of a relation
    - **select count**(*)
      from *instructor*
      where *dept_name* = 'Physics'
  - and a transaction T2 that inserts a tuple while T1 is active but after predicate read
    - **insert into** *instructor* **values** ('11111', 'Feynman', 'Physics', 94000)

    (conceptually) conflict in spite of not accessing any tuple in common.
- If only tuple locks are used, non-serializable schedules can result
  - E.g. the scan transaction does not see the new instructor, but may read some other tuple written by the update transaction
- Can also occur with updates
  - E.g. update Wu's department from Finance to Physics

Non-Serializable Execution Due to Phantom Phenomenon

T1	T2
Read(instructor where dept_name='Physics')	
	Insert Instructor in Physics
	Insert Instructor in Comp. Sci.
	Commit
Read(instructor where dept_name='Comp. Sci.')	

Another Example: T1 and T2 both find maximum instructor ID in parallel, and create new instructors with ID = maximum ID + 1

- Both instructors get same ID, not possible in serializable schedule

Handling Phantoms

- There is a conflict at the data level
  - The transaction performing predicate read or scanning the relation is reading information that indicates what tuples the relation contains
  - The transaction inserting/deleting/updating a tuple updates the same information.
  - The conflict should be detected, e.g. by locking the information.
- One solution:
  - Associate a data item with the relation, to represent the information about what tuples the relation contains.
  - Transactions scanning the relation acquire a shared lock in the data item,
  - Transactions inserting or deleting a tuple acquire an exclusive lock on the data item. (Note: locks on the data item do not conflict with locks on individual tuples.)
- Above protocol provides very low concurrency for insertions/deletions.

Index Locking To Prevent Phantoms

- **Index locking protocol** to prevent phantoms
  - Every relation must have at least one index.
  - A transaction can access tuples only after finding them through one or more indices on the relation
  - A transaction T_i that performs a lookup must lock all the index leaf nodes that it accesses, in S-mode
    - Even if the leaf node does not contain any tuple satisfying the index lookup (e.g. for a range query, no tuple in a leaf is in the range)
  - A transaction T_i that inserts, updates or deletes a tuple t_i in a relation r
    - must update all indices to r
    - must obtain exclusive locks on all index leaf nodes affected by the insert/update/delete
  - The rules of the two-phase locking protocol must be observed
- Guarantees that phantom phenomenon won't occur

Next-Key Locking to Prevent Phantoms

- Index-locking protocol to prevent phantoms locks entire leaf node
  - Can result in poor concurrency if there are many inserts
- **Next-key locking protocol**: provides higher concurrency
  - Lock all values that satisfy index lookup (match lookup value, or fall in lookup range)
  - Also lock next key value in index
    - even for inserts/deletes
  - Lock mode: S for lookups, X for insert/delete/update
- Ensures detection of query conflicts with inserts, deletes and updates

Consider B+-tree leaf nodes as below, with query predicate $7 \leq X \leq 16$. Check what happens with next-key locking when inserting: (i) 15 and (ii) 7

TIMESTAMP BASED CONCURRENCY CONTROL

Timestamp-Based Protocols

- Each transaction T_i is issued a timestamp TS(T_i) when it enters the system.
  - Each transaction has a *unique* timestamp
  - Newer transactions have timestamps strictly greater than earlier ones
  - Timestamp could be based on a logical counter
    - Real time may not be unique
    - Can use (wall-clock time, logical counter) to ensure
- Timestamp-based protocols manage concurrent execution such that
  time-stamp order = serializability order
- Several alternative protocols based on timestamps

Timestamp-Ordering Protocol

The **timestamp ordering (TSO) protocol**

- Maintains for each data *Q* two timestamp values:
  - **W-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **write**(*Q*) successfully.
  - **R-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **read**(*Q*) successfully.
- Imposes rules on read and write operations to ensure that
  - any conflicting operations are executed in timestamp order
  - out of order operations cause transaction rollback

Timestamp-Based Protocols (Cont.)

- Suppose a transaction T_i issues a **read**(Q)
  1. If TS(T_i) ≤ **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten.
     - Hence, the **read** operation is rejected, and T_i is rolled back.
  2. If TS(T_i) ≥ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to

     max(R-timestamp(Q), TS(T_i)).

Timestamp-Based Protocols (Cont.)

- Suppose that transaction T_i issues **write**(Q).
  1. If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.
     - Hence, the **write** operation is rejected, and T_i is rolled back.
  2. If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q.
     - Hence, this **write** operation is rejected, and T_i is rolled back.
  3. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

Example of Schedule Under TSO

- Is this schedule valid under TSO?

  Assume that initially:
  R-TS(A) = W-TS(A) = 0
  R-TS(B) = W-TS(B) = 0
  Assume TS(T_{25}) = 25 and
  TS(T_{26}) = 26

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

- And how about this one, where initially
  R-TS(Q)=W-TS(Q)=0

T_{27}	T_{28}
read(Q)	
	write(Q)
write(Q)	

Another Example Under TSO

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5, with all R-TS and W-TS = 0 initially

T_1	T_2	T_3	T_4	T_5
				read (X)
	read (Y)			
read (Y)				
		write (Y)		
		write (Z)		
				read (Z)
	read (Z)			
	abort			
read (X)				
			read (W)	
		write (W)		
		abort		
				write (Y)
				write (Z)

Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

  Thus, there will be no cycles in the precedence graph
- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be cascade-free, and may not even be recoverable.

Recoverability and Cascade Freedom

- Solution 1:
  - A transaction is structured such that its writes are all performed at the end of its processing
  - All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
  - A transaction that aborts is restarted with a new timestamp
- Solution 2: Limited form of locking: wait for data to be committed before reading it
- Solution 3: Use commit dependencies to ensure recoverability

Thomas' Write Rule

- Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances.
- When T_i attempts to write data item Q, if TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of {Q}.
  - Rather than rolling back T_i as the timestamp ordering protocol would have done, this {**write**} operation can be ignored.
- Otherwise this protocol is the same as the timestamp ordering protocol.
- Thomas' Write Rule allows greater potential concurrency.
  - Allows some view-serializable schedules that are not conflict-serializable.

Validation-Based Protocol

- Idea: can we use commit time as serialization order?
- To do so:
  - Postpone writes to end of transaction
  - Keep track of data items read/written by transaction
  - **Validation** performed at commit time, detect any out-of-serialization order reads/writes
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation

Validation-Based Protocol

- Execution of transaction T_i is done in three phases.

1. **Read and execution phase**: Transaction T_i writes only to temporary local variables
2. **Validation phase**: Transaction T_i performs a "validation test" to determine if local variables can be written without violating serializability.
3. **Write phase**: If T_i is validated, the updates are applied to the database; otherwise, T_i is rolled back.

- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
  - We assume for simplicity that the validation and write phase occur together, atomically and serially
    - I.e., only one transaction executes validation/write at a time.

Validation-Based Protocol (Cont.)

- Each transaction T_i has 3 timestamps
  - **StartTS**(T_i) : the time when T_i started its execution
  - **ValidationTS**(T_i): the time when T_i entered its validation phase
  - **FinishTS**(T_i) : the time when T_i finished its write phase
- Validation tests use above timestamps and read/write sets to ensure that serializability order is determined by validation time
  - Thus, TS(T_i) = ValidationTS(T_i)
- Validation-based protocol has been found to give greater degree of concurrency than locking/TSO if probability of conflicts is low.

Validation Test for Transaction T_j

- If for all T_i with TS (T_i) < TS (T_j) either one of the following condition holds:
  - **finishTS**(T_i) < **startTS**(T_j)
  - **startTS**(T_j) < **finishTS**(T_i) < **validationTS**(T_j) **and** the set of data items written by T_i does not intersect with the set of data items read by T_j.

  then validation succeeds and T_j can be committed.
- Otherwise, validation fails and T_j is aborted.
- *Justification*:
  - First condition applies when execution is not concurrent
    - The writes of T_j do not affect reads of T_i since they occur after T_i has finished its reads.
  - If the second condition holds, execution is concurrent, T_j does not read any item written by T_i.

Schedule Produced by Validation

- Example of schedule produced using validation

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
<validate>	
display($A + B$)	
	<validate>
	write(B)
	write(A)

MULTIVERSION CONCURRENCY CONTROL

Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency. Several variants:
  - **Multiversion Timestamp Ordering**
  - **Multiversion Two-Phase Locking**
  - **Snapshot isolation**
- Key ideas:
  - Each successful **write** results in the creation of a new version of the data item written.
  - Use timestamps to label versions.
  - When a **read**(Q) operation is issued, select an appropriate version of Q based on the timestamp of the transaction issuing the read request, and return the value of the selected version.
- **read**s never have to wait as an appropriate version is returned immediately.

Multiversion Timestamp Ordering

- Each data item Q has a sequence of versions $<Q_1, Q_2, ..., Q_m>$. Each version Q_k contains three data fields:
  - **Content** -- the value of version Q_k.
  - **W-timestamp**(Q_k) -- timestamp of the transaction that created (wrote) version Q_k
  - **R-timestamp**(Q_k) -- largest timestamp of a transaction that successfully read version Q_k

Multiversion Timestamp Ordering (Cont)

- Suppose that transaction T_i issues a **read**(Q) or **write**(Q) operation. Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i).
  1. If transaction T_i issues a **read**(Q), then
     - the value returned is the content of version Q_k
     - If R-timestamp(Q_k) < TS(T_i), set R-timestamp(Q_k) = TS(T_i),
  2. If transaction T_i issues a **write**(Q)
     1. if TS(T_i) < R-timestamp(Q_k), then transaction T_i is rolled back.
     2. if TS(T_i) = W-timestamp(Q_k), the contents of Q_k are overwritten
     3. Otherwise, a new version Q_i of Q is created
        - W-timestamp(Q_i) and R-timestamp(Q_i) are initialized to TS(T_i).

Multiversion Timestamp Ordering (Cont)

- Observations
  - Reads always succeed
  - A write by T_i is rejected if some other transaction T_j that (in the serialization order defined by the timestamp values) should read T_i's write, has already read a version created by a transaction older than T_i.
- Protocol guarantees serializability

Multiversion Two-Phase Locking

- Differentiates between read-only transactions and update transactions
- **Update transactions** acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
  - Read of a data item returns the latest version of the item
  - The first **write** of Q by T_i results in the creation of a new version Q_i of the data item Q written
    - W-timestamp(Q_i) set to ∞ initially
  - When update transaction T_i completes, commit processing occurs:
    - Value **ts-counter** stored in the database is used to assign timestamps
      - **ts-counter** is locked in two-phase manner
    - Set TS(T_i) = **ts-counter** + 1
    - Set W-timestamp(Q_i) = TS(T_i) for all versions Q_i that it creates
    - **ts-counter** = **ts-counter +** 1

Multiversion Two-Phase Locking (Cont.)

- **Read-only transactions**
  - are assigned a timestamp = **ts-counter** when they start execution
  - follow the multiversion timestamp-ordering protocol for performing reads
    - Do not obtain any locks
- Read-only transactions that start after T_i increments **ts-counter** will see the values updated by T_i.
- Read-only transactions that start before T_i increments the **ts-counter** will see the value before the updates by T_i.
- Only serializable schedules are produced.

MVCC: Implementation Issues

- Creation of multiple versions increases storage overhead
  - Extra tuples
  - Extra space in each tuple for storing version information
- Versions can, however, be garbage collected
  - E.g. if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp > 9, than Q5 will never be required again
- Issues with
  - primary key and foreign key constraint checking
  - Indexing of records with multiple versions

  See textbook for details

Snapshot Isolation

- Motivation: Decision support queries that read large amounts of data have concurrency conflicts with OLTP transactions that update a few rows
  - Poor performance results
- Solution 1: Use multiversion 2-phase locking
  - Give logical "snapshot" of database state to read only transaction
    - Reads performed on snapshot
  - Update (read-write) transactions use normal locking
  - Works well, but how does system know a transaction is read only?
- (Partial) Solution 2: Give snapshot of database state to every transaction
  - Reads performed on snapshot
  - Use 2-phase locking on updated data items
  - Problem: variety of anomalies such as lost update can result
  - Better solution: snapshot isolation level (next slide)

Snapshot Isolation

- A transaction T1 executing with Snapshot Isolation
  - takes snapshot of committed data at start
  - always reads/modifies data in its own snapshot
  - updates of concurrent transactions are not visible to T1
  - writes of T1 complete when it commits
  - **First-committer-wins rule**:
    - Commits only if no other concurrent transaction has already written data that T1 intends to write.

T1	T2	T3
W(Y := 1) Commit		
	Start R(X) → 0 R(Y)→ 1	
		W(X:=2) W(Z:=3) Commit
	R(Z) → 0 R(Y) → 1 W(X:=3) Commit-Req Abort	

Concurrent updates not visible
Own updates are visible
Not first-committer of X
Serialization error, T2 is rolled back

Snapshot Read

- Concurrent updates invisible to snapshot read

$X_0 = 100,\ Y_0 = 0$

T_1 deposits 50 in Y	T_2 withdraws 50 from X
$r_1(X_0, 100)$	
$r_1(Y_0, 0)$	
	$r_2(Y_0, 0)$
	$r_2(X_0, 100)$
	$w_2(X_2, 50)$
$w_1(Y_1, 50)$	
$r_1(X_0, 100)$ (update by T_2 not seen)	
$r_1(Y_1, 50)$ (can see its own updates)	
	$r_2(Y_0, 0)$ (update by T_1 not seen)

$X_2 = 50,\ Y_1 = 50$

Snapshot Write: First Committer Wins

$X_0 = 100$

T_1 deposits 50 in X	T_2 withdraws 50 from X
$r_1(X_0, 100)$	
	$r_2(X_0, 100)$
	$w_2(X_2, 50)$
$w_1(X_1, 150)$	
$commit_1$	
	$commit_2$ (Serialization Error T_2 is rolled back)

$X_1 = 150$

- Variant: "**First-updater-wins**"
  - Check for concurrent updates when write occurs by locking item
    - But lock should be held till all concurrent transactions have finished
  - (Oracle uses this plus some extra features)
  - Differs only in when abort occurs, otherwise equivalent

Benefits of SI

- Reads are *never* blocked,
  - and also don't block other txns activities
- Performance similar to Read Committed
- Avoids several anomalies
  - No dirty read, i.e. no read of uncommitted data
  - No lost update
    - i.e. update made by a transaction is overwritten by another transaction that did not see the update)
  - No non-repeatable read
    - i.e. if read is executed again, it will see the same value
- Problems with SI
  - SI does not always give serializable executions
    - Serializable: among two concurrent txns, one sees the effects of the other
    - In SI: neither sees the effects of the other
  - Result: Integrity constraints can be violated

Snapshot Isolation

- E.g. of problem with SI
  - Initially A = 3 and B = 17
    - Serial execution: A = ??, B = ??
    - if both transactions start at the same time, with snapshot isolation: A = ?? , B = ??
- Called **skew write**
- Skew also occurs with inserts
  - E.g:
    - Find max order number among all orders
    - Create a new order with order number = previous max + 1
    - Two transaction can both create order with same number
      - Is an example of phantom phenomenon

T_i	T_j
read(A)	
read(B)	
	read(A)
	read(B)
A=B	
	B=A
write(A)	
	write(B)

Snapshot Isolation Anomalies

- SI breaks serializability when transactions modify *different* items, each based on a previous state of the item the other modified
  - Not very common in practice
    - E.g., the TPC-C benchmark runs correctly under SI
    - when txns conflict due to modifying different data, there is usually also a shared item they both modify, so SI will abort one of them
  - But problems do occur
    - Application developers should be careful about write skew
- SI can also cause a read-only transaction anomaly, where read-only transaction may see an inconsistent state even if updaters are serializable
  - We omit details
- Using snapshots to verify primary/foreign key integrity can lead to inconsistency
  - Integrity constraint checking usually done outside of snapshot

Serializable Snapshot Isolation

- **Serializable snapshot isolation (SSI)**: extension of snapshot isolation that ensures serializability
- Snapshot isolation tracks write-write conflicts, but does not track read-write conflicts
  - where T_i writes a data a data item Q, T_j reads an earlier version of Q, but T_j is serialized after T_i
- Idea: track read-write dependencies separately, and roll-back transactions where cycles can occur
  - Ensures serializability
  - Details in book
- Implemented in PostgreSQL from version 9.1 onwards
  - PostgreSQL implementation of SSI also uses index locking to detect phantom conflicts, thus ensuring true serializability

SI Implementations

- Snapshot isolation supported by many databases
  - Including Oracle, PostgreSQL, SQL Server, IBM DB2, etc
  - Isolation level can be set to snapshot isolation
- Oracle implements "first updater wins" rule (variant of "first committer wins")
  - concurrent writer check is done at time of write, not at commit time
  - Allows transactions to be rolled back earlier
- **Warning**: *even if isolation level is set to serializable, Oracle actually uses snapshot isolation*
  - Old versions of PostgreSQL prior to 9.1 did this too
  - Oracle and PostgreSQL < 9.1 do not support true serializable execution

Working Around SI Anomalies

- Can work around SI anomalies for specific queries by using **select .. for update** (supported e.g. in Oracle)
  - E.g.,
    - **select max**(orderno) **from** orders **<u>for update</u>**
    - read value into local variable maxorder
    - insert into orders (maxorder+1, …)
- **select for update (SFU) clause** treats all data read by the query as if it were also updated, preventing concurrent updates
- Can be added to queries to ensure serializability in many applications
  - Does not handle phantom phenomenon/predicate reads though

WEAK LEVELS OF CONCURRENCY

Weak Levels of Consistency

- **Degree-two consistency**: differs from two-phase locking in that S-locks may be released at any time, and locks may be acquired at any time
  - X-locks must be held till end of transaction
  - Serializability is not guaranteed, programmer must ensure that no erroneous database state will occur]
- **Cursor stability**:
  - For reads, each tuple is locked, read, and lock is immediately released
  - X-locks are held till end of transaction
  - Special case of degree-two consistency

Weak Levels of Consistency in SQL

- SQL allows non-serializable executions
  - **Serializable**: is the default
  - **Repeatable read**: allows only committed records to be read, and repeating a read should return the same value (so read locks should be retained)
    - However, the phantom phenomenon need not be prevented
      - T1 may see some records inserted by T2, but may not see others inserted by T2
  - **Read committed**: same as degree two consistency, but most systems implement it as cursor-stability
  - **Read uncommitted**: allows even uncommitted data to be read
- In most database systems, read committed is the default consistency level
  - Can be changed as database configuration parameter, or per transaction
    - **set isolation level serializable**

Concurrency Control across User Interactions

- Many applications need transaction support across user interactions
  - Can't use locking for long durations
- Application level concurrency control
  - Each tuple has a version number
  - Transaction notes version number when reading tuple
    - **select** r.balance, r.version **into** :A, :version
      from r **where** acctId =23
  - When writing tuple, check that current version number is same as the version when tuple was read
    - **update** r **set** r.balance = r.balance + :deposit, r.version = r.version+1
      where acctId = 23 **and** r.version = :version

Concurrency Control across User Interactions

- Equivalent to **optimistic concurrency control without validating read set**
  - Unlike SI, reads are not guaranteed to be from a single snapshot.
  - Does not guarantee serializability
  - But avoids some anomalies such as “lost update anomaly”
- Used internally in Hibernate ORM system
- Implemented manually in many applications
- Version numbers stored in tuples can also be used to support first committer wins check of snapshot isolation

ADVANCED TOPICS IN CONCURRENCY CONTROL

Online Index Creation

- Problem: how to create an index on a large relation without affecting concurrent updates
  - Index construction may take a long time
  - Two-phase locking will block all concurrent updates
- Key ideas:
  - build index on a snapshot of the relation, but keep track of all updates that occur after snapshot
    - Updates are not applied on the index at this point
  - Then apply subsequent updates to catch up
  - Acquire relation lock towards end of catchup phase to block concurrent updates
  - Catch up with remaining updates, and add index to system catalog
  - Subsequent transactions will find the index in catalog and update it

Concurrency in Index Structures

- Indices are unlike other database items in that their only job is to help in accessing data.
- Index-structures are typically accessed very often, much more than other database items.
  - Treating index-structures like other database items, e.g. by 2-phase locking of index nodes can lead to low concurrency.
- There are several index concurrency protocols where locks on internal nodes are released early, and not in a two-phase fashion.
  - It is acceptable to have nonserializable concurrent access to an index as long as the accuracy of the index is maintained.
    - In particular, the exact values read in an internal node of a B^+-tree are irrelevant so long as we land up in the correct leaf node.

Concurrency in Index Structures (Cont.)

- **Crabbing protocol** used instead of two-phase locking on the nodes of the B^+-tree during search/insertion/deletion:
  - First lock the root node in shared mode.
  - After locking all required children of a node in shared mode, release the lock on the node
  - During insertion/deletion, upgrade leaf node locks to exclusive mode.
  - When splitting or coalescing requires changes to a parent, lock the parent in exclusive mode.
- Above protocol can cause excessive deadlocks
  - Searches coming down the tree deadlock with updates going up the tree
  - Can abort and restart search, without affecting transaction
- The **B-link tree locking protocol** improves concurrency
  - Intuition: release lock on parent before acquiring lock on child
    - And deal with changes that may have happened between lock release and acquire

Concurrency Control in Main-Memory Databases

- Index locking protocols can be simplified with main-memory databases
  - Short term lock can be obtained on entire index for duration of an operation, serializing updates on the index
    - Avoids overheads of multiple lock acquire/release
    - No major penalty since operations finish fast, since there is no disk wait
- Latch-free techniques for data-structure update can speed up operations further

Latch-Free Data-structure Updates

- This code is not safe without latches if executed concurrently:

```
insert(value, head) {
    node = new node
    node->value = value
    node->next = head
    head = node
}
```

- This code is safe

```
insert latchfree(head, value) {
    node = new node
    node->value = value
    repeat
        oldhead = head
        node->next = oldhead
        result = CAS(head, oldhead, node)
    until (result == success)
}
```

Latch-Free Data-structure Updates

- This code is not safe without latches if executed concurrently:

```
insert(value, head) {
    node = new node
    node->value = value
    node->next = head
    head = node
}
```

- This code is safe

```
insert latchfree(head, value) {
    node = new node
    node->value = value
    repeat
        oldhead = head
        node->next = oldhead
        result = CAS(head, oldhead, node)
    until (result == success)
}
```

Latch-Free Data-structures (Cont.)

- *Consider:*

```
delete latchfree(head) {
    /* This function is not quite safe; see explanation in text. */
    repeat
        oldhead = head
        newhead = oldhead−>next
        result = CAS(head, oldhead, newhead)
    until (result == success)
}
```

- Above code is almost correct, but has a concurrency bug
  - P1 initiates delete with N1 as head; concurrently P2 deletes N1 and next node N2, and then reinserts N1 as head, with N3 as next
  - P1 may set head as N2 instead of N3.
- Known as ABA problem
- See book for details of how to avoid this problem

Concurrency Control with Operations

- Consider this non-two phase schedule, which preserves database integrity constraints
- Can be understood as transaction performing increment operation
  - E.g. increment(A, -50), increment (B, 50)
  - As long as increment operation does not return actual value, increments can be reordered
    - ***Increments commute***
  - New increment-mode lock to support reordering
  - Conflict matrix with increment lock mode
    - *Two increment operations do not conflict with each other*

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

	S	X	I
S	true	false	false
X	false	false	false
I	false	false	true

Concurrency Control with Operations (Cont.)

- Undo of increment(v, n) is performed by increment (v, -n)
- Increment_conditional(v, n):
  - Updates v by adding n to it, as long as final v > 0, fails otherwise
  - Can be used to model, e.g. number of available tickets, *avail_tickets,* for a concert
  - Increment_conditional is NOT commutative
    - E.g. last few tickets for a concert
  - But reordering may still be acceptable

Real-Time Transaction Systems

- Transactions in a system may have deadlines within which they must be completed.
  - Hard deadline: missing deadline is an error
  - Firm deadline: value of transaction is 0 in case deadline is missed
  - Soft deadline: transaction still has some value if done after deadline
- Locking can cause blocking
- Optimistic concurrency control (validation protocol) has been shown to do will in a real-time setting

End of Chapter 18

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

View Serializability

- Let S and S' be two schedules with the same set of transactions. S and S' are **view equivalent** if the following three conditions are met, for each data item Q,
  1. If in schedule S, transaction T_i reads the initial value of Q, then in schedule S' also transaction T_i must read the initial value of Q.
  2. If in schedule S transaction T_i executes **read**(Q), and that value was produced by transaction T_j (if any), then in schedule S' also transaction T_i must read the value of Q that was produced by the same **write**(Q) operation of transaction T_j .
  3. The transaction (if any) that performs the final **write**(Q) operation in schedule S must also perform the final **write**(Q) operation in schedule S'.
- As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

View Serializability (Cont.)

- A schedule *S* is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes**.

Test for View Serializability

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.
  - Thus existence of an efficient algorithm is *extremely* unlikely.
- However practical algorithms that just check some **sufficient conditions** for view serializability can still be used.

Other Notions of Serializability

- The schedule below produces same outcome as the serial schedule < T_1, T_5 >, yet is not conflict equivalent or view equivalent to it.

T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

- Determining such equivalence requires analysis of operations other than read and write.
  - Operation-conflicts, operation locks

Chapter 19: Recovery System

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Outline

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Remote Backup Systems

Failure Classification

- **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error condition
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- **System crash**: a power failure or other hardware or software failure causes the system to crash.
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures

Recovery Algorithms

- Suppose transaction T_i transfers \$50 from account *A* to account *B*
  - Two updates: subtract 50 from A and add 50 to B
- Transaction T_i requires updates to A and B to be output to the database.
  - A failure may occur after one of these modifications have been made but before both of them are made.
  - Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state
  - Not modifying the database may result in lost updates if failure occurs just after transaction commits
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

Storage Structure

- **Volatile storage**:
  - Does not survive system crashes
  - Examples: main memory, cache memory
- **Nonvolatile storage**:
  - Survives system crashes
  - Examples: disk, tape, flash memory, non-volatile RAM
  - But may still fail, losing data
- **Stable storage**:
  - A mythical form of storage that survives all failures
  - Approximated by maintaining multiple copies on distinct nonvolatile media
  - See book for more details on how to implement stable storage

Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies: Block transfer can result in
  - Successful completion
  - Partial failure: destination block has incorrect information
  - Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
  - Execute output operation as follows (assuming two copies of each block):
    1. Write the information onto the first physical block.
    2. When the first write successfully completes, write the same information onto the second physical block.
    3. The output is completed only after the second write successfully completes.

Protecting storage media from failure (Cont.)

- Copies of a block may differ due to failure during output operation.
- To recover from failure:
  1. First find inconsistent blocks:
     1. *Expensive solution*: Compare the two copies of every disk block.
     2. *Better solution*:
        - Record in-progress disk writes on non-volatile storage (Flash, Non-volatile RAM or special area of disk).
        - Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
        - Used in hardware RAID systems
  2. If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.

Data Access

- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
  - **input** (*B*) transfers the physical block *B* to main memory.
  - **output** (*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.

Data Access (Cont.)

- Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept.
  - T_i's local copy of a data item X is called x_i.
- Transferring data items between system buffer blocks and its private work-area done by:
  - **read**(X) assigns the value of data item X to the local variable x_i.
  - **write**(X) assigns the value of local variable x_i to data item {X} in the buffer block.
  - Note: **output**(B_X) need not immediately follow **write**(X). System can perform the **output** operation when it deems fit.
- Transactions
  - Must perform **read**(X) before accessing X for the first time (subsequent reads can be from local copy)
  - **write**(X) can be executed at any time before the transaction commits

Example of Data Access

Recovery and Atomicity

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study **log-based recovery mechanisms** in detail
  - We first present key concepts
  - And then present the actual recovery algorithm
- Less used alternative: **shadow-copy** and **shadow-paging** (brief details in book)

shadow-copy

Log-Based Recovery

- A **log** is a sequence of **log records**. The records keep information about update activities on the database.
  - The **log** is kept on stable storage
- When transaction T_i starts, it registers itself by writing a

  <T_i **start**> log record
- *Before* T_i executes **write**(X), a log record

  <T_i, X, V_1, V_2>

  is written, where V_1 is the value of X before the write (the **old value**), and V_2 is the value to be written to X (the **new value**).
- When T_i finishes it last statement, the log record <T_i **commit**> is written.
- Two approaches using logs
  - Immediate database modification
  - Deferred database modification.

Immediate Database Modification

- The **immediate-modification** scheme allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits
- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
  - (Will see later that how to postpone log record output to some extent)
- Output of updated blocks to disk can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.
- The **deferred-modification** scheme performs updates to buffer/disk only at the time of transaction commit
  - Simplifies some aspects of recovery
  - But has overhead of storing local copy

Transaction Commit

- A transaction is said to have committed when its commit log record is output to stable storage
  - All previous log records of the transaction must have been output already
- Writes performed by a transaction may still be in the buffer when the transaction commits, and may be output later

Immediate Database Modification Example

Log	Write	Output
<T_0 **start**>		
<T_0, A, 1000, 950>		
<T_0, B, 2000, 2050>		
	A = 950	
	B = 2050	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	C = 600	
		B_B , B_C
<T_1 **commit**>		
		B_A

B_C output before T_1 commits

B_A output after T_0 commits

- Note: B_X denotes block containing X.

Concurrency Control and Recovery

- With concurrent transactions, all transactions share a single disk buffer and a single log
  - A buffer block can have data items updated by one or more transactions
- We assume that *if a transaction T_i has modified an item, no other transaction can modify the same item until T_i has committed or aborted*
  - i.e., the updates of uncommitted transactions should not be visible to other transactions
    - Otherwise, how to perform undo if T_1 updates A, then T_2 updates A and commits, and finally T_1 has to abort?
  - Can be ensured by obtaining exclusive locks on updated items and holding the locks till end of transaction (strict two-phase locking)
- Log records of different transactions may be interspersed in the log.

Undo and Redo Operations

- **Undo and Redo of Transactions**
  - **undo**(T_i) -- restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
    - Each time a data item X is restored to its old value V a special log record <T_i , X, V> is written out
    - When undo of a transaction is complete, a log record <T_i **abort**> is written out.
  - **redo**(T_i) -- sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
    - No logging is done in this case

Recovering from Failure

- When recovering after failure:
  - Transaction T_i needs to be undone if the log
    - Contains the record <T_i **start**>,
    - But does not contain either the record <T_i **commit**> *or* <T_i **abort**>.
  - Transaction T_i needs to be redone if the log
    - Contains the records <T_i **start**>
    - And contains the record <T_i **commit**> *or* <T_i **abort**>

Recovering from Failure (Cont.)

- Suppose that transaction T_i was undone earlier and the <T_i **abort**> record was written to the log, and then a failure occurs,
- On recovery from failure transaction T_i is redone
  - Such a **redo** redoes all the original actions of transaction T_i *including the steps that restored old values*
    - Known as **repeating history**
    - Seems wasteful, but simplifies recovery greatly

Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000, and log records <T_0, B, 2000>, <T_0, A, 1000>, <T_0, **abort**> are written out

(b) redo (T_0) and undo (T_1): *A* and *B* are set to 950 and 2050 and C is restored to 700. Log records <T_1, C, 700>, <T_1, **abort**> are written out.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600

Checkpoints

- Redoing/undoing all transactions recorded in the log can be very slow
  - Processing the entire log is time-consuming if the system has run for a long time
  - We might unnecessarily redo transactions which have already output their updates to the database.
- Streamline recovery procedure by periodically performing **checkpointing**
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record < **checkpoint** *L*> onto stable storage where *L* is a list of all transactions active at the time of checkpoint.
  4. All updates are stopped while doing checkpointing

Checkpoints (Cont.)

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
  - Scan backwards from end of log to find the most recent <**checkpoint** *L*> record
  - Only transactions that are in *L* or started after the checkpoint need to be redone or undone
  - Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage.
- Some earlier part of the log may be needed for undo operations
  - Continue scanning backwards till a record <T_i **start**> is found for every transaction T_i in *L*.
  - Parts of log prior to earliest <T_i **start**> record above are not needed for recovery, and can be erased whenever desired.

Example of Checkpoints

- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

Recovery Algorithm

Recovery Algorithm

- **So far:** we covered key concepts
- **Now**: we present the components of the basic recovery algorithm
- **Later**: we present extensions to allow more concurrency

Recovery Algorithm

- **Logging** (during normal operation):
  - <T_i **start**> at transaction start
  - <T_i, X_j, V_1, V_2> for each update, and
  - <T_i **commit**> at transaction end
- **Transaction rollback (during normal operation)**
  - Let T_i be the transaction to be rolled back
  - Scan log backwards from the end, and for each log record of T_i of the form <T_i, X_j, V_1, V_2>
    - Perform the undo by writing V_1 to X_j,
    - Write a log record <T_i , X_j, V_1>
      - such log records are called **compensation log records**
  - Once the record <T_i **start**> is found stop the scan and write the log record <T_i **abort**>

Recovery Algorithm (Cont.)

- **Recovery from failure**: Two phases
  - **Redo phase**: replay updates of **all** transactions, whether they committed, aborted, or are incomplete
  - **Undo phase**: undo all incomplete transactions
- **Redo phase**:
  1. Find last <**checkpoint** *L*> record, and set undo-list to *L*.
  2. Scan forward from above <**checkpoint** *L*> record
     1. Whenever a record <T_i, X_j, V_1, V_2> or <T_i, X_j, V_2> is found, redo it by writing V_2 to X_j
     2. Whenever a log record <T_i **start**> is found, add T_i to undo-list
     3. Whenever a log record <T_i **commit**> *or* <T_i **abort**> is found, remove T_i from undo-list

Recovery Algorithm (Cont.)

- **Undo phase:**
  1. Scan log backwards from end
     1. Whenever a log record <T_i, X_j, V_1, V_2> is found where T_i is in undo-list perform same actions as for transaction rollback:
        1. perform undo by writing V_1 to X_j.
        2. write a log record <T_i , X_j, V_1>
     2. Whenever a log record <T_i **start**> is found where T_i is in undo-list,
        1. Write a log record <T_i **abort**>
        2. Remove T_i from undo-list
     3. Stop when undo-list is empty
        - i.e., <T_i **start**> has been found for every transaction in undo-list
- After undo phase completes, normal transaction processing can commence

Example of Recovery

Log Record Buffering

- **Log record buffering**: log records are buffered in main memory, instead of being output directly to stable storage.
  - Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed.
- Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.
- Several log records can thus be output using a single output operation, reducing the I/O cost.

Log Record Buffering (Cont.)

- The rules below must be followed if log records are buffered:
  - Log records are output to stable storage in the order in which they are created.
  - Transaction T_i enters the commit state only when the log record <T_i **commit**> has been output to stable storage.
  - Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage.
    - This rule is called the **write-ahead logging** or **WAL** rule
    - Strictly speaking, WAL only requires undo information to be output

Database Buffering

- Database maintains an in-memory buffer of data blocks
  - When a new block is needed, if buffer is full an existing block needs to be removed from buffer
  - If the block chosen for removal has been updated, it must be output to disk
- The recovery algorithm supports the **no-force policy**: i.e., updated blocks need not be written to disk when transaction commits
  - **force policy**: requires updated blocks to be written at commit
    - More expensive commit
- The recovery algorithm supports the **steal policy**: i.e., blocks containing updates of uncommitted transactions can be written to disk, even before the transaction commits

Database Buffering (Cont.)

- If a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first
  - (Write ahead logging)
- No updates should be in progress on a block when it is output to disk. Can be ensured as follows.
  - Before writing a data item, transaction acquires exclusive lock on block containing the data item
  - Lock can be released once the write is completed.
    - Such locks held for short duration are called **latches**.
- **To output a block to disk**
  1. First acquire an exclusive latch on the block
     - Ensures no update can be in progress on the block
  2. Then perform a **log flush**
  3. Then output the block to disk
  4. Finally release the latch on the block

Buffer Management (Cont.)

- Database buffer can be implemented either
  - In an area of real main-memory reserved for the database, or
  - In virtual memory
- Implementing buffer in reserved main-memory has drawbacks:
  - Memory is partitioned before-hand between database buffer and applications, limiting flexibility.
  - Needs may change, and although operating system knows best how memory should be divided up at any time, it cannot change the partitioning of memory.

Buffer Management (Cont.)

- Database buffers are generally implemented in virtual memory in spite of some drawbacks:
  - When operating system needs to evict a page that has been modified, the page is written to swap space on disk.
  - When database decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O!
    - Known as **dual paging** problem.
  - Ideally when OS needs to evict a page from the buffer, it should pass control to database, which in turn should
    1. Output the page to database instead of to swap space (making sure to output log records first), if it is modified
    2. Release the page from the buffer, for the OS to use
  - Dual paging can thus be avoided, but common operating systems do not support such functionality.

Fuzzy Checkpointing

- To avoid long interruption of normal processing during checkpointing, allow updates to happen during checkpointing
- **Fuzzy checkpointing** is done as follows:
  1. Temporarily stop all updates by transactions
  2. Write a <**checkpoint** *L*> log record and force log to stable storage
  3. Note list *M* of modified buffer blocks
  4. Now permit transactions to proceed with their actions
  5. Output to disk all modified buffer blocks in list *M*
     - blocks should not be updated while being output
     - Follow WAL: all log records pertaining to a block must be output before the block is output
  6. Store a pointer to the **checkpoint** record in a fixed position **last_checkpoint** on disk

Fuzzy Checkpointing (Cont.)

- When recovering using a fuzzy checkpoint, start scan from the **checkpoint** record pointed to by **last_checkpoint**
  - Log records before **last_checkpoint** have their updates reflected in database on disk, and need not be redone.
  - Incomplete checkpoints, where system had crashed while performing checkpoint, are handled safely

Failure with Loss of Nonvolatile Storage

- So far we assumed no loss of non-volatile storage
- Technique similar to checkpointing used to deal with loss of non-volatile storage
  - Periodically **dump** the entire content of the database to stable storage
  - No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place
    - Output all log records currently residing in main memory onto stable storage.
    - Output all buffer blocks onto the disk.
    - Copy the contents of the database to stable storage.
    - Output a record <**dump**> to log on stable storage.

Recovering from Failure of Non-Volatile Storage

- To recover from disk failure
  - restore database from most recent dump.
  - Consult the log and redo all transactions that committed after the dump
- Can be extended to allow transactions to be active during dump; known as **fuzzy dump** or **online dump**
  - Similar to fuzzy checkpointing

Remote Backup Systems

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Remote Backup Systems

- Remote backup systems provide high availability by allowing transaction processing to continue even if the primary site is destroyed.

Remote Backup Systems (Cont.)

- **Detection of failure**: Backup site must detect when primary site has failed
  - to distinguish primary site failure from link failure maintain several communication links between the primary and the remote backup.
  - Heart-beat messages
- **Transfer of control**:
  - To take over control backup site first perform recovery using its copy of the database and all the long records it has received from the primary.
    - Thus, completed transactions are redone and incomplete transactions are rolled back.
  - When the backup site takes over processing it becomes the new primary
  - To transfer control back to old primary when it recovers, old primary must receive redo logs from the old backup and apply all updates locally.

Remote Backup Systems (Cont.)

- **Time to recover**: To reduce delay in takeover, backup site periodically proceses the redo log records (in effect, performing recovery from previous database state), performs a checkpoint, and can then delete earlier parts of the log.
- **Hot-Spare** configuration permits very fast takeover:
  - Backup continually processes redo log record as they arrive, applying the updates locally.
  - When failure of the primary is detected the backup rolls back incomplete transactions, and is ready to process new transactions.
- Alternative to remote backup: distributed database with replicated data
  - Remote backup is faster and cheaper, but less tolerant to failure
    - more on this in Chapter 19

Remote Backup Systems (Cont.)

- Ensure durability of updates by delaying transaction commit until update is logged at backup; avoid this delay by permitting lower degrees of durability.
- **One-safe**: commit as soon as transaction's commit log record is written at primary
  - Problem: updates may not arrive at backup before it takes over.
- **Two-very-safe**: commit when transaction's commit log record is written at primary and backup
  - Reduces availability since transactions cannot commit if either site fails.
- **Two-safe**: proceed as in two-very-safe if both primary and backup are active. If only the primary is active, the transaction commits as soon as is commit log record is written at the primary.
  - Better availability than two-very-safe; avoids problem of lost transactions in one-safe.

Recovery with Early Lock Release and Logical Undo Operations

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Recovery with Early Lock Release

- Support for high-concurrency locking techniques, such as those used for B^+-tree concurrency control, which release locks early
  - Supports "logical undo"
- Recovery based on "repeating history", whereby recovery executes exactly the same actions as normal processing

Logical Undo Logging

- Operations like B^+-tree insertions and deletions release locks early.
  - They cannot be undone by restoring old values (**physical undo**), since once a lock is released, other transactions may have updated the B^+-tree.
  - Instead, insertions (resp. deletions) are undone by executing a deletion (resp. insertion) operation (known as **logical undo**).
- For such operations, undo log records should contain the undo operation to be executed
  - Such logging is called **logical undo logging**, in contrast to **physical undo logging**
    - Operations are called **logical operations**
  - Other examples:
    - delete of tuple, to undo insert of tuple
      - allows early lock release on space allocation information
    - subtract amount deposited, to undo deposit
      - allows early lock release on bank balance

Physical Redo

- Redo information is logged **physically** (that is, new value for each write) even for operations with logical undo
  - Logical redo is very complicated since database state on disk may not be "operation consistent" when recovery starts
  - Physical redo logging does not conflict with early lock release

Operation Logging

- Operation logging is done as follows:
  1. When operation starts, log <T_i, O_j, **operation-begin**>. Here O_j is a unique identifier of the operation instance.
  2. While operation is executing, normal log records with physical redo and physical undo information are logged.
  3. When operation completes, <T_i, O_j, **operation-end**, U> is logged, where U contains information needed to perform a logical undo information.

Example: insert of (key, record-id) pair (K5, RID7) into index I9 (Key at location X, record-id at location X+8) with old values Old1 and Old2

<T1, O1, operation-begin>

….

<T1, X, Old1, K5>

<T1, X+8, Old2, RID7>

Physical redo of steps in insert

<T1, O1, operation-end, (delete I9, K5, RID7)>

Operation Logging (Cont.)

- If crash/rollback occurs before operation completes:
  - the **operation-end** log record is not found, and
  - the physical undo information is used to undo operation.
- If crash/rollback occurs after the operation completes:
  - the **operation-end** log record is found, and in this case
  - logical undo is performed using *U*; the physical undo information for the operation is ignored.
- <u>Redo of operation (after crash) still uses physical redo information</u>.

Transaction Rollback with Logical Undo

Rollback of transaction T_i is done as follows:

- Scan the log backwards
  1. If a log record <T_i, X, V_1, V_2> is found, perform the undo and log a al <T_i, X, V_1>.
  2. If a <T_i, O_j, **operation-end**, U> record is found
     - Rollback the operation logically using the undo information U.
       - Updates performed during roll back are logged just like during normal operation execution.
       - At the end of the operation rollback, instead of logging an **operation-end** record, generate a record

         <T_i, O_j, **operation-abort**>.
     - Skip all preceding log records for T_i until the record <T_i, O_j **operation-begin**> is found

Transaction Rollback with Logical Undo (Cont.)

- Transaction rollback, scanning the log backwards (cont.):
  3. If a redo-only record is found ignore it
  4. If a <T_i, O_j, **operation-abort**> record is found:
     - skip all preceding log records for T_i until the record <T_i, O_j, **operation-begi**n> is found.
  5. Stop the scan when the record <T_i, start> is found
  6. Add a <T_i, **abort**> record to the log

Some points to note:

- Cases 3 and 4 above can occur only if the database crashes while a transaction is being rolled back.
- Skipping of log records as in case 4 is important to prevent multiple rollback of the same operation.

Transaction Rollback with Logical Undo

- **Transaction rollback during normal operation**

Beginning of log

<T_0 start>
<T_0, B, 2000, 2050>
<T_0, O_1, operation-begin>
<T_0, C, 700, 600>
<T_0, O_1, operation-end, (C, +100)>
<T_1 start>
<T_0, O_2, operation-begin>
<T_1, C, 600, 400>
<T_1, O_2, operation-end, (C, +200)>

<T_0, C, 400, 500>
<T_0, O_1, operation-abort>
<T_0, B, 2000>
<T_0, abort>
<T_1, commit>

If T_0 aborts before operation O_1 ends, undo of update to C will be physical

T_0 has completed operation O_1 on C, releases lower-level lock; physical undo cannot be done anymore, logical undo will add 100 to C

T_1 can update C since T_0 has released lower-level lock on C

T_1 releases lower-level lock on C

T_0 decides to abort

Logical undo of O_1 adds 100 on C

O_1 undo complete

Failure Recovery with Logical Undo

Beginning of log

<T_0 start>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, B, 2050, 2100>
<T_1, O_4, operation-begin>
<checkpoint {T_1}>
<T_1, C, 700, 400>
<T_1, O_4, operation-end (C, +300)>
<T_2 start>
<T_2, O_5, operation-begin>
<T_2, C, 400, 300>

End of log at crash!

Start log records found for all transactions in undo list

Redo Pass

Undo list: T_1, T_2

Undo Pass

Records added during recovery

<T_2, C, 400>
<T_2 abort>
<T_1, C, 400, 700>
<T_1, O_4, operation-abort>
<T_1, B, 2050>
<T_1 abort>

Update of C was part of O_5, undone physically during recovery since O_5 did not complete

Logical undo of O_4 adds 300 to C

Transaction Rollback: Another Example

- Example with a complete and an incomplete operation

  <T1, start>

  <T1, O1, operation-begin>

  ….

  <T1, X, 10, K5>

  <T1, Y, 45, RID7>

  <T1, O1, operation-end, (delete I9, K5, RID7)>

  <T1, O2, operation-begin>

  <T1, Z, 45, 70>

  ← T1 Rollback begins here

  <T1, Z, 45> ← redo-only log record during physical undo (of incomplete O2)

  <T1, Y, .., ..> ← Normal redo records for logical undo of O1

  …

  <T1, O1, operation-abort> ← What if crash occurred immediately after this?

  <T1, abort>

Recovery Algorithm with Logical Undo

Basically same as earlier algorithm, except for changes described earlier for transaction rollback

1. (**Redo phase**): Scan log forward from last < **checkpoint** *L*> record till end of log
   1. **Repeat history** by physically redoing all updates of all transactions,
   2. Create an undo-list during the scan as follows
      - *undo-list* is set to *L* initially
      - Whenever <T_i **start**> is found T_i is added to *undo-list*
      - Whenever <T_i **commit**> or <T_i **abort**> is found, T_i is deleted from *undo-list*

   This brings database to state as of crash, with committed as well as uncommitted transactions having been redone.

   Now *undo-list* contains transactions that are **incomplete**, that is, have neither committed nor been fully rolled back.

Recovery with Logical Undo (Cont.)

Recovery from system crash (cont.)

2. (**Undo phase**): Scan log backwards, performing undo on log records of transactions found in *undo-list*.
   - Log records of transactions being rolled back are processed as described earlier, as they are found
     - Single shared scan for all transactions being undone
   - When <T_i **start**> is found for a transaction T_i in *undo-list*, write a <T_i **abort**> log record.
   - Stop scan when <T_i **start**> records have been found for all T_i in *undo-list*

- This undoes the effects of incomplete transactions (those with neither **commit** nor **abort** log records). Recovery is now complete.

ARIES Recovery Algorithm

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

ARIES

- ARIES is a state of the art recovery method
  - Incorporates numerous optimizations to reduce overheads during normal processing and to speed up recovery
  - The recovery algorithm we studied earlier is modeled after ARIES, but greatly simplified by removing optimizations
- Unlike the recovery algorithm described earlier, ARIES
  1. Uses **log sequence number (LSN)** to identify log records
     - Stores LSNs in pages to identify what updates have already been applied to a database page
  2. Physiological redo
  3. Dirty page table to avoid unnecessary redos during recovery
  4. Fuzzy checkpointing that only records information about dirty pages, and does not require dirty pages to be written out at checkpoint time
     - More coming up on each of the above …

ARIES Optimizations

- **Physiological redo**
  - Affected page is physically identified, action within page can be logical
    - Used to reduce logging overheads
      - e.g. when a record is deleted and all other records have to be moved to fill hole
        - Physiological redo can log just the record deletion
        - Physical redo would require logging of old and new values for much of the page
    - Requires page to be output to disk atomically
      - Easy to achieve with hardware RAID, also supported by some disk systems
      - Incomplete page output can be detected by checksum techniques,
        - But extra actions are required for recovery
        - Treated as a media failure

ARIES Data Structures

- ARIES uses several data structures
  - Log sequence number (LSN) identifies each log record
    - Must be sequentially increasing
    - Typically an offset from beginning of log file to allow fast access
      - Easily extended to handle multiple log files
  - Page LSN
  - Log records of several different types
  - Dirty page table

ARIES Data Structures: Page LSN

- Each page contains a **PageLSN** which is the LSN of the last log record whose effects are reflected on the page
  - To update a page:
    - X-latch the page, and write the log record
    - Update the page
    - Record the LSN of the log record in PageLSN
    - Unlock page
  - To flush page to disk, must first S-latch page
    - Thus page state on disk is operation consistent
      - Required to support physiological redo
  - PageLSN is used during recovery to prevent repeated redo
    - Thus ensuring idempotence

ARIES Data Structures: Log Record

- Each log record contains LSN of previous log record of the same transaction

LSN	TransID	PrevLSN	RedoInfo	UndoInfo

  - LSN in log record may be implicit
- Special redo-only log record called **compensation log record (CLR)** used to log actions taken during recovery that never need to be undone
  - Serves the role of operation-abort log records used in earlier recovery algorithm
  - Has a field UndoNextLSN to note next (earlier) record to be undone
    - Records in between would have already been undone
    - Required to avoid repeated undo of already undone actions

LSN	TransID	UndoNextLSN	RedoInfo

ARIES Data Structures: DirtyPage Table

- **DirtyPageTable**
  - List of pages in the buffer that have been updated
  - Contains, for each such page
    - **PageLSN** of the page
    - **RecLSN** is an LSN such that log records before this LSN have already been applied to the page version on disk
      - Set to current end of log when a page is inserted into dirty page table (just before being updated)
      - Recorded in checkpoints, helps to minimize redo work

ARIES Data Structures

Database Buffer

PageID	PageLSN	RecLSN
4894	7567	7564
7200	7565	7565

Dirty Page Table

Log Buffer (PrevLSN and UndoNextLSN fields not shown)

Stable data

Stable log

ARIES Data Structures: Checkpoint Log

- **Checkpoint log record**
  - Contains:
    - DirtyPageTable and list of active transactions
    - For each active transaction, LastLSN, the LSN of the last log record written by the transaction
  - Fixed position on disk notes LSN of last completed checkpoint log record
- Dirty pages are not written out at checkpoint time
    - Instead, they are flushed out continuously, in the background
- Checkpoint is thus very low overhead
  - can be done frequently

ARIES Recovery Algorithm

ARIES recovery involves three passes

- Analysis pass: Determines
  - Which transactions to undo
  - Which pages were dirty (disk version not up to date) at time of crash
  - RedoLSN: LSN from which redo should start
- Redo pass:
  - Repeats history, redoing all actions from RedoLSN
    - RecLSN and PageLSNs are used to avoid redoing actions already reflected on page
- Undo pass:
  - Rolls back all incomplete transactions
    - Transactions whose abort was complete earlier are not undone
      - Key idea: no need to undo these transactions: earlier undo actions were logged, and are redone as required

Aries Recovery: 3 Passes

- Analysis, redo and undo passes
- Analysis determines where redo should start
- Undo has to go back till start of earliest incomplete transaction

ARIES Recovery: Analysis

Analysis pass

- Starts from last complete checkpoint log record
  - Reads DirtyPageTable from log record
  - Sets RedoLSN = min of RecLSNs of all pages in DirtyPageTable
    - In case no pages are dirty, RedoLSN = checkpoint record's LSN
  - Sets undo-list = list of transactions in checkpoint log record
  - Reads LSN of last log record for each transaction in undo-list from checkpoint log record
- Scans forward from checkpoint
- .. Cont. on next page …

ARIES Recovery: Analysis (Cont.)

Analysis pass (cont.)

- Scans forward from checkpoint
  - If any log record found for transaction not in undo-list, adds transaction to undo-list
  - Whenever an update log record is found
    - If page is not in DirtyPageTable, it is added with RecLSN set to LSN of the update log record
  - If transaction end log record found, delete transaction from undo-list
  - Keeps track of last log record for each transaction in undo-list
    - May be needed for later undo
- At end of analysis pass:
  - RedoLSN determines where to start redo pass
  - RecLSN for each page in DirtyPageTable used to minimize redo work
  - All transactions in undo-list need to be rolled back

ARIES Redo Pass

Redo Pass: Repeats history by replaying every action not already reflected in the page on disk, as follows:

- Scans forward from RedoLSN. Whenever an update log record is found:
  1. If the page is not in DirtyPageTable or the LSN of the log record is less than the RecLSN of the page in DirtyPageTable, then skip the log record
  2. Otherwise fetch the page from disk. If the PageLSN of the page fetched from disk is less than the LSN of the log record, redo the log record

NOTE: if either test is negative the effects of the log record have already appeared on the page. First test avoids even fetching the page from disk!

ARIES Undo Actions

- When an undo is performed for an update log record
  - Generate a CLR containing the undo action performed (actions performed during undo are logged physicaly or physiologically).
    - CLR for record *n* noted as *n*' in figure below
  - Set UndoNextLSN of the CLR to the PrevLSN value of the update log record
    - Arrows indicate UndoNextLSN value
- ARIES supports partial rollback
  - Used e.g. to handle deadlocks by rolling back just enough to release reqd. locks
  - Figure indicates forward actions after partial rollbacks
    - records 3 and 4 initially, later 5 and 6, then full rollback

ARIES: Undo Pass

Undo pass:

- Performs backward scan on log undoing all transaction in undo-list
  - Backward scan optimized by skipping unneeded log records as follows:
    - Next LSN to be undone for each transaction set to LSN of last log record for transaction found by analysis pass.
    - At each step pick largest of these LSNs to undo, skip back to it and undo it
    - After undoing a log record
      - For ordinary log records, set next LSN to be undone for transaction to PrevLSN noted in the log record
      - For compensation log records (CLRs) set next LSN to be undo to UndoNextLSN noted in the log record
        - All intervening records are skipped since they would have been undone already
- Undos performed as described earlier

Recovery Actions in ARIES

Other ARIES Features

- Recovery Independence
  - Pages can be recovered independently of others
    - E.g. if some disk pages fail they can be recovered from a backup while other pages are being used
- Savepoints:
  - Transactions can record savepoints and roll back to a savepoint
    - Useful for complex transactions
    - Also used to rollback just enough to release locks on deadlock

Other ARIES Features (Cont.)

- Fine-grained locking:
  - Index concurrency algorithms that permit tuple level locking on indices can be used
    - These require logical undo, rather than physical undo, as in earlier recovery algorithm
- Recovery optimizations: For example:
  - Dirty page table can be used to prefetch pages during redo
  - Out of order redo is possible:
    - redo can be postponed on a page being fetched from disk, and performed when page is fetched.
    - Meanwhile other log records can continue to be processed

Recovery in Main Memory Databases

- Normal recovery algorithms can be used with main-memory databases
- But optimizations are possible
  - No redo-logging for indices: indices can be rebuilt quickly in-memory on recovery
  - No undo logging if only committed data is written to disk
  - Parallel recovery is important to load large amounts of data in-memory and perform recover with minimum delay

End of Chapter 19

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

A
B

<T_0 start>

<T_0, A, 1000, 950>

<T_0, B, 2000, 2050>

<T_0 commit>

<T_1 start>

<T_1, C, 700, 600>

<T_1 commit>

Log	Database
<T_0 start>	
<T_0, A, 1000, 950>	
<T_0, B, 2000, 2050>	
	$A = 950$
	$B = 2050$
<T_0 commit>	
<T_1 start>	
<T_1, C, 700, 600>	
	$C = 600$
<T_1 commit>	

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

Beginning of log

older

<T_0 start>
<T_0, B, 2000, 2050>
<T_1 start>
<checkpoint {T_0, T_1}>
<T_1, C, 700, 600>
<T_1 commit>
<T_2 start>
<T_2, A, 500, 400>
<T_0, B, 2000>
<T_0 abort>

<T_2, A, 500>
<T_2 abort>

newer

End of log at crash!

Log records added during recovery

T_0 rollback (during normal operation) begins

T_0 rollback complete

T_2 is incomplete at crash

T_2 rolled back in undo pass

Start log records found for all transactions in undo list

Redo Pass

Undo list: T_2

Undo Pass

If T_0 aborts before operation O_1 ends, undo of update to C will be physical

Beginning of log

<T_0 start>
<T_0, B, 2000, 2050>
<T_0, O_1, operation-begin>
<T_0, C, 700, 600>
<T_0, O_1, operation-end, (C, +100)>
<T_1 start>
<T_1, O_2, operation-begin>
<T_1, C, 600, 400>
<T_1, O_2, operation-end, (C, +200)>

<T_0 , C, 400, 500>
<T_0, O_1, operation-abort>
<T_0, B, 2000>
<T_0, abort>
<T_1, commit>

T_0 has completed operation O_1 on C, releases lower-level lock; physical undo cannot be done anymore, logical undo will add 100 to C

T_1 can update C since T_0 has released lower-level lock on C

T_1 releases lower-level lock on C

T0 decides to abort

Logical undo of O_1 adds 100 to C

O_1 undo complete

Database Buffer

PageID	PageLSN	RecLSN
4894	7567	7564
7200	7565	7565

Dirty Page Table

Log Buffer (PrevLSN and UndoNextLSN fields not shown)

Stable data

Stable log

newer
End of log at crash
PrevLSN pointers
7571: <T146 commit>
7570: <T146, 2390.4, 50, 90>
7569: <T146 begin>
7568: checkpoint
Txn | lastLSN
T145 | 7567
PageID | PageLSN | RecLSN
4894 | 7567 | 7564
7200 | 7565 | 7565
7567: <T145,4894.1, 40, 60>
7566: <T143 commit>
7565: <T143,7200.2, 60>
7564: <T145,4894.1, 20, 40>
7563: <T145 begin>
7562: <T143,7200.2, 60, 80>
older
Analysis pass
Redo pass
Undo pass
CLR
UndoNextLSN

imary

Extra

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Shadow Paging

- **Shadow paging** is an alternative to log-based recovery; this scheme is useful if transactions execute serially
- Idea: maintain *two* page tables during the lifetime of a transaction –the **current page table**, and the **shadow page table**
- Store the shadow page table in nonvolatile storage, such that state of the database prior to transaction execution may be recovered.
  - Shadow page table is never modified during execution
- To start with, both the page tables are identical. Only current page table is used for data item accesses during execution of the transaction.
- Whenever any page is about to be written for the first time
  - A copy of this page is made onto an unused page.
  - The current page table is then made to point to the copy
  - The update is performed on the copy

Sample Page Table

Example of Shadow Paging

Shadow and current page tables after write to page 4

Shadow Paging (Cont.)

- To commit a transaction :

1. Flush all modified pages in main memory to disk
2. Output current page table to disk
3. Make the current page table the new shadow page table, as follows:
   - keep a pointer to the shadow page table at a fixed (known) location on disk.
   - to make the current page table the new shadow page table, simply update the pointer to point to current page table on disk

- Once pointer to shadow page table has been written, transaction is committed.
- No recovery is needed after a crash — new transactions can start right away, using the shadow page table.
- Pages not pointed to from current/shadow page table should be freed (garbage collected).

Show Paging (Cont.)

- Advantages of shadow-paging over log-based schemes
  - no overhead of writing log records
  - recovery is trivial
- Disadvantages:
  - Copying the entire page table is very expensive
    - Can be reduced by using a page table structured like a B^+-tree
      - No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes
  - Commit overhead is high even with above extension
    - Need to flush every updated page, and page table
  - Data gets fragmented (related pages get separated on disk)
  - After every transaction completion, the database pages containing old versions of modified data need to be garbage collected
  - Hard to extend algorithm to allow transactions to run concurrently
    - Easier to extend log based schemes

Block Storage Operations

Portion of the Database Log Corresponding to T_0 and T_1

<T_0 start>
<T_0 , A, 950>
<T_0 , B, 2050>
<T_0 commit>
<T_1 start>
<T_1 , C, 600>
<T_1 commit>

State of the Log and Database Corresponding to T_0 and T_1

Log	Database
<T_0 start>	
<T_0, A, 950>	
<T_0, B, 2050>	
<T_0 commit>	
	$A = 950$
	$B = 2050$
<T_1 start>	
<T_1, C, 600>	
<T_1 commit>	
	$C = 600$

Portion of the System Log Corresponding to T_0 and T_1

<T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>
<T_1 commit>

State of System Log and Database Corresponding to T_0 and T_1

Log	Database
<T_0 start>	
<T_0, A, 1000, 950>	
<T_0, B, 2000, 2050>	
	$A = 950$
	$B = 2050$
<T_0 commit>	
<T_1 start>	
<T_1, C, 700, 600>	
	$C = 600$
<T_1 commit>	

Undo and Redo Operations

- **Undo** of a log record <T_i, X, V_1, V_2> writes the **old** value V_1 to X
- **Redo** of a log record <T_i, X, V_1, V_2> writes the **new** value V_2 to X
- **Undo and Redo of Transactions**
  - **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
    - Each time a data item X is restored to its old value V a special log record <T_i , X, V> is written out
    - When undo of a transaction is complete, a log record <T_i **abort**> is written out.
  - **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
    - No logging is done in this case

Recovering from Failure

- When recovering after failure:
  - Transaction T_i needs to be undone if the log
    - contains the record <T_i **start**>,
    - but does not contain either the record <T_i **commit**> *or* <T_i **abort**>.
  - Transaction T_i needs to be redone if the log
    - contains the records <T_i **start**>
    - and contains the record <T_i **commit**> *or* <T_i **abort**>
- Note that If transaction T_i was undone earlier and the <T_i **abort**> record written to the log, and then a failure occurs, on recovery from failure T_i is redone
  - **such a redo redoes all the original actions *including the steps that restored old values***
    - Known as **repeating history**
    - Seems wasteful, but simplifies recovery greatly

Example of Checkpoints

- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

Chapter 20: Database System Architectures

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Chapter 21: Database System Architectures

- Centralized Database Systems
- Server System Architectures
- Parallel Systems
- Distributed Systems
- Network Types

Centralized Database Systems

- Run on a single computer system
- **Single-user system**
  - Embedded databases
- **Multi-user systems** also known as **server systems**.
  - Service requests received from client systems
  - Multi-core systems with **coarse-grained parallelism**
    - Typically a few to tens of processor cores
    - In contrast, fine-grained parallelism uses very large number of computers

Server System Architecture

- Server systems can be broadly categorized into two kinds:
  - **transaction servers**
    - Widely used in relational database systems, and
  - **data servers**
    - Parallel data servers used to implement high-performance transaction processing systems

Transaction Servers

- Also called **query server** systems or SQL *server* systems
  - Clients send requests to the server
  - Transactions are executed at the server
  - Results are shipped back to the client.
- Requests are specified in SQL, and communicated to the server through a *remote procedure call* (RPC) mechanism.
- Transactional RPC allows many RPC calls to form a transaction.
- Applications typically use ODBC/JDBC APIs to communicate with transaction servers

Transaction System Processes (Cont.)

Transaction Server Process Structure

- A typical transaction server consists of multiple processes accessing data in shared memory
- Shared memory contains shared data
  - Buffer pool
  - Lock table
  - Log buffer
  - Cached query plans (reused if same query submitted again)
- All database processes can access shared memory
- Server processes
  - These receive user queries (transactions), execute them and send results back
  - Processes may be **multithreaded**, allowing a single process to execute several user queries concurrently
  - Typically multiple multithreaded server processes

Transaction Server Processes (Cont.)

- Database writer process
  - Output modified buffer blocks to disks continually
- Log writer process
  - Server processes simply add log records to log record buffer
  - Log writer process outputs log records to stable storage.
- Checkpoint process
  - Performs periodic checkpoints
- Process monitor process
  - Monitors other processes, and takes recovery actions if any of the other processes fail
    - E.g. aborting any transactions being executed by a server process and restarting it

Transaction System Processes (Cont.)

- Lock manager process
  - To avoid overhead of interprocess communication for lock request/grant, each database process operates directly on the lock table
    - instead of sending requests to lock manager process
  - Lock manager process still used for deadlock detection
- To ensure that no two processes are accessing the same data structure at the same time, databases systems implement **mutual exclusion** using either
  - Atomic instructions
    - Test-And-Set
    - Compare-And-Swap (CAS)
  - Operating system semaphores
    - Higher overhead than atomic instructions

Atomic Instructions

- **Test-And-Set**(M)
  - Memory location M, initially 0
  - Test-and-set(M) sets M to 1, and returns old value of M
    - Return value 0 indicates process has acquired the mutex
    - Return value 1 indicates someone is already holding the mutex
      - Must try again later
    - Release of mutex done by setting M = 0
- **Compare-and-swap**(M, V1, V2)
  - Atomically do following
    - If M = V1, set M = V2 and return success
    - Else return failure
  - With M = 0 initially, CAS(M, 0, 1) equivalent to test-and-set(M)
  - Can use CAS(M, 0, id) where id = thread-id or process-id to record who has the mutex

Data Servers/Data Storage Systems

- Data items are shipped to clients where processing is performed
- Updated data items written back to server
- Earlier generation of data servers would operated in units of data items, or pages containing multiple data items
- Current generation data servers (also called data storage systems) only work in units of data items
  - Commonly used data item formats include JSON, XML, or just uninterpreted binary strings

Data Servers/Storage Systems (Cont.)

- **Prefetching**
  - Prefetch items that may be used soon
- **Data caching**
  - Cache coherence
- **Lock caching**
  - Locks can be cached by client across transactions
  - Locks can be **called back** by the server
- **Adaptive lock granularity**
  - **Lock granularity escalation**
    - switch from finer granularity (e.g. tuple) lock to coarser
  - **Lock granularity de-escalation**
    - Start with coarse granularity to reduve overheads, switch to finer granularity in case of more concurrency conflict at server
    - Details in book

Data Servers (Cont.)

- **Data Caching**
  - Data can be cached at client even in between transactions
  - But check that data is up-to-date before it is used (**cache coherency**)
  - Check can be done when requesting lock on data item
- **Lock Caching**
  - Locks can be retained by client system even in between transactions
  - Transactions can acquire cached locks locally, without contacting server
  - Server **calls back** locks from clients when it receives conflicting lock request. Client returns lock once no local transaction is using it.
    - Similar to lock callback on prefetch, but across transactions.

Parallel Systems

- Parallel database systems consist of multiple processors and multiple disks connected by a fast interconnection network.
- Motivation: handle workloads beyond what a single computer system can handle
- High performance **transaction processing**
  - E.g. handling user requests at web-scale
- **Decision support** on very large amounts of data
  - E.g. data gathered by large web sites/apps

Parallel Systems (Cont.)

- A **coarse-grain parallel** machine consists of a small number of powerful processors
- A **massively parallel** or **fine grain parallel** machine utilizes thousands of smaller processors.
  - Typically hosted in a **data center**
- Two main performance measures:
  - **throughput** --- the number of tasks that can be completed in a given time interval
  - **response time** --- the amount of time it takes to complete a single task from the time it is submitted

Speed-Up and Scale-Up

- **Speedup**: a fixed-sized problem executing on a small system is given to a system which is *N*-times larger.
  - Measured by:
  - $speedup = \frac{small\ system\ elapsed\ time}{large\ system\ elapsed\ time}$
  - Speedup is **linear** if equation equals N.
- **Scaleup**: increase the size of both the problem and the system
  - *N*-times larger system used to perform *N*-times larger job
  - Measured by:
  - $scaleup = \frac{small\ system\ small\ problem\ elapsed\ time}{big\ system\ big\ problem\ elapsed\ time}$
  - Scale up is **linear** if equation equals 1.

Speedup

Scaleup

Batch and Transaction Scaleup

- **Batch scaleup**:
  - A single large job; typical of most decision support queries and scientific simulation.
  - Use an *N*-times larger computer on *N*-times larger problem.
- **Transaction scaleup**:
  - Numerous small queries submitted by independent users to a shared database; typical transaction processing and timesharing systems.
  - *N*-times as many users submitting requests (hence, *N*-times as many requests) to an *N*-times larger database, on an *N*-times larger computer.
  - Well-suited to parallel execution.

Factors Limiting Speedup and Scaleup

Speedup and scaleup are often sublinear due to:

- **Startup/sequential costs**: Cost of starting up multiple processes, and sequential computation before/after parallel computation
  - May dominate computation time, if the degree of parallelism is high
  - Suppose *p* fraction of computation is sequential
  - **Amdahl's law**: speedup limited to: $1 / [(1-p)+(p/n)]$
  - **Gustafson's law**: scaleup limited to: $1 / [n(1-p)+p]$
- **Interference**: Processes accessing shared resources (e.g.,system bus, disks, or locks) compete with each other, thus spending time waiting on other processes, rather than performing useful work.
- **Skew**: Increasing the degree of parallelism increases the variance in service times of parallely executing tasks. Overall execution time determined by **slowest** of parallely executing tasks.

Interconnection Network Architectures

- **Bus**. System components send data on and receive data from a single communication bus;
  - Does not scale well with increasing parallelism.
- **Mesh**. Components are arranged as nodes in a grid, and each component is connected to all adjacent components
  - Communication links grow with growing number of components, and so scales better.
  - But may require $2\sqrt{n}$ hops to send message to a node (or $\sqrt{n}$ with wraparound connections at edge of grid).
- **Hypercube**. Components are numbered in binary; components are connected to one another if their binary representations differ in exactly one bit.
  - *n* components are connected to *log(n)* other components and can reach each other via at most *log(n)* links; reduces communication delays.
- **Tree-like Topology**. Widely used in data centers today

Interconnection Architectures

(a) bus

(b) ring

(c) mesh

(d) hypercube

Interconnection Network Architectures

- **Tree-like or Fat-Tree Topology**: widely used in data centers today
  - Top of rack switch for approx 40 machines in rack
  - Each top of rack switch connected to multiple aggregation switches.
  - Aggregation switches connect to multiple core switches.
- **Data center fabric**

(e) tree-like topology

Network Technologies

- Ethernet
  - 1 Gbps and 10 Gbps common, 40 Gbps and 100 Gbps are available at higher cost
- Fiber Channel
  - 32-138 Gbps available
- **Infiniband**
  - a very-low-latency networking technology
    - 0.5 to 0.7 microseconds, compared to a few microseconds for optimized ethernet

Parallel Database Architectures

(a) shared memory

(b) shared disk

(c) shared nothing

(d) hierarchical

Shared Memory

- Processors (or processor cores) and disks have access to a common memory
  - Via a bus in earlier days, through an interconnection network today
- Extremely efficient communication between processors
- Downside: shared-memory architecture is not scalable beyond 64 to 128 processor cores
  - Memory interconnection network becomes a bottleneck

(a) shared memory

Modern Shared Memory Architecture

Cache Levels

- Cache line: typically 64 bytes in today's processors
- Cache levels within a single multi-core processor

Core 0	Core 1	Core 2	Core 3
L1 Cache	L1 Cache	L1 Cache	L1 Cache
L2 Cache	L2 Cache	L2 Cache	L2 Cache
Shared L3 Cache			

- Shared memory system can have multiple processors, each with its own cache levels

Cache Coherency

- Cache coherency:
  - Local cache may have out of date value
  - Strong vs weak consistency models
  - With weak consistency, need special instructions to ensure cache is up to date
- Memory barrier instructions
  - **Store barrier (sfence)**
    - Instruction returns after forcing cached data to be written to memory and invalidations sent to all caches
  - **Load barrier (lfence)**
    - Returns after ensuring all pending cache invalidations are processed
  - mfence instruction does both of above
- Locking code usually takes care of barrier instructions
  - Lfence done after lock acquisition and sfence done before lock release

Shared Disk

- All processors can directly access all disks via an interconnection network, but the processors have private memories.
  - Architecture provides a degree of **fault-tolerance** — if a processor fails, the other processors can take over its tasks
    - the data of the failed processor is resident on disks that are accessible from all processors.
- Downside: bottleneck now occurs at interconnection to the disk subsystem.

(b) shared disk

Modern Shared Disk Architectures: via Storage Area Network (SAN)

Shared Nothing

- Node consists of a processor, memory, and one or more disks
- All communication via interconnection network
- Can be scaled up to thousands of processors without interference.
- Main drawback: cost of communication and non-local disk access; sending data involves software interaction at both ends.

(c) shared nothing

Hierarchical

- Combines characteristics of shared-memory, shared-disk, and shared-nothing architectures.
  - Top level is a shared-nothing architecture
    - With each node of the system being a shared-memory system
  - Alternatively, top level could be a shared-disk system
    - With each node of the system being a shared-memory system

(d) hierarchical

Shared-Memory Vs Shared-Nothing

- Shared-memory internally looks like shared-nothing!
  - Each processor has direct access to its own memory, and indirect (hardware level) access to rest of memory
  - Also called **non-uniform memory architecture (NUMA)**
- Shared-nothing can be made to look like shared memory
  - Reduce the complexity of programming such systems by **distributed virtual-memory** abstraction
  - **Remote Direct Memory Access (RDMA)** provides very low-latency shared memory abstraction on shared-nothing systems
    - Often implemented on top of i**nfiniband** due it its very-low-latency
  - But careless programming can lead to performance issues

Distributed Systems

- Data spread over multiple machines (also referred to as **sites** or **nodes**).
- **Local-area networks** (**LANs**)
- **Wide-area networks** (**WAN**s)
  - Higher latency

Distributed Databases

- **Homogeneous distributed databases**
  - Same software/schema on all sites, data may be partitioned among sites
  - Goal: provide a view of a single database, hiding details of distribution
- **Heterogeneous distributed databases**
  - Different software/schema on different sites
  - Goal: integrate existing databases to provide useful functionality
- Differentiate between **local transactions** and **global transactions**
  - A local transaction accesses data in the *single* site at which the transaction was initiated.
  - A global transaction either accesses data in a site different from the one at which the transaction was initiated or accesses data in several different sites.

Data Integration and Distributed Databases

- Data integration between multiple distributed databases
- Benefits:
  - Sharing data – users at one site able to access the data residing at some other sites.
  - Autonomy – each site is able to retain a degree of control over data stored locally.

Availability

- **Network partitioning**
- **Availability** of system
  - If all nodes are required for system to function, failure of even one node stops system functioning.
  - Higher system availability through redundancy
    - data can be replicated at remote sites, and system can function even if a site fails.

Implementation Issues for Distributed Databases

- Atomicity needed even for transactions that update data at multiple sites
- The **two-phase commit protocol (2PC)** is used to ensure atomicity
  - Basic idea: each site executes transaction until just before commit, and the leaves final decision to a coordinator
  - Each site must follow decision of coordinator, even if there is a failure while waiting for coordinators decision
- 2PC is not always appropriate: other transaction models based on persistent messaging, and workflows, are also used
- Distributed concurrency control (and deadlock detection) required
- Data items may be replicated to improve data availability
- Details of all above in Chapter 24

Cloud Based Services

- Cloud computing widely adopted today
  - On-demand provisioning and **elasticity**
    - ability to scale up at short notice and to release of unused resources for use by others
- Infrastructure as a service
  - Virtual machines/real machines
- Platform as a service
  - Storage, databases, application server
- Software as a service
  - Enterprise applications, emails, shared documents, etc,
- Potential drawbacks
  - Security
  - Network bandwidth

Cloud Service Models

Application Deployment Alternatives

a) Multiple applications on a single machine

b) Each application running on its own VM, with multiple VMs running in a machine

c) Each application running in its own container, with multiple containers running in a machine

Individual Machines

Virtual Machines
(e.g. VMWare, KVM, ..)

Containers
(e.g. Docker)

Application Deployment Architectures

- Services
- Microservice Architecture
  - Application uses a variety of services
  - Service can add or remove instances as required
- Kubernetes supports containers, and microservices

End of Chapter 20

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Figure 17.11

storage array
storage array
tape library
SAN
server
server
LAN/WAN
client
client
client
data-processing center
Web content provider

Chapter 21: Parallel and Distributed Storage

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Introduction

- Parallel machines have become quite common and affordable
  - prices of microprocessors, memory and disks have dropped sharply
- Data storage needs are growing increasingly large
  - user data at web-scale
    - 100's of millions of users, petabytes of data
  - transaction data are collected and stored for analysis.
  - multimedia objects like images/videos
- Parallel storage system requirements
  - storing large volumes of data
  - processing time-consuming decision-support queries
  - providing high throughput for transaction processing
  - Very high demands on **scalability** and **availability**

Parallel/Distribted Data Storage History

- 1980/1990s
  - Distributed database systems with tens of nodes
- 2000s:
  - Distributed file systems with 1000s of nodes
    - Millions of Large objects (100's of megabytes)
    - Web logs, images, videos, …
    - Typically create/append only
  - Distributed data storage systems with 1000s of nodes
    - Billions to trillions of smaller (kilobyte to megabyte) objects
    - Social media posts, email, online purchases, …
    - Inserts, updates, deletes
  - **Key-value stores**
- 2010s: Distributed database systems with 1000s of nodes

I/O Parallelism

- Reduce the time required to retrieve relations from disk by partitioning the relations on *multiple disks*, on *multiple* ***nodes*** (computers)
  - Our description focuses on parallelism across nodes
  - Same techniques can be used across disks on a node
- **Horizontal partitioning** – tuples of a relation are divided among many nodes such that some subset of tuple resides on each node.
  - Contrast with **vertical partitioning**, e.g. *r*(*A,B,C,D*) with primary key *A* into r1(*A,B*) and r2(*A,C,D*)
  - By default, the word partitioning refers to horizontal partitioning

I/O Parallelism

- Partitioning techniques (number of nodes = *n*):

  Round-robin:

  Send the ith tuple inserted in the relation to node *i* mod *n*.

  Hash partitioning:

  - Choose one or more attributes as the partitioning attributes.
  - Choose hash function *h* with range 0…*n* - 1
  - Let *i* denote result of hash function *h* applied to the partitioning attribute value of a tuple. Send tuple to node *i*.

Range Partitioning

I/O Parallelism (Cont.)

Partitioning techniques (cont.):

- **Range partitioning**:
  - Choose an attribute as the partitioning attribute.
  - A partitioning vector $[v_o, v_1, ..., v_{n-2}]$ is chosen.
  - Let v be the partitioning attribute value of a tuple. Tuples such that $v_i \leq v_{i+1}$ go to node $I + 1$. Tuples with $v < v_0$ go to node 0 and tuples with $v \geq v_{n-2}$ go to node n-1.

  E.g., with a partitioning vector [5,11]

  - a tuple with partitioning attribute value of 2 will go to node 0,
  - a tuple with value 8 will go to node 1, while
  - a tuple with value 20 will go to node2.

Comparison of Partitioning Techniques

- Evaluate how well partitioning techniques support the following types of data access:
  1. Scanning the entire relation.
  2. Locating a tuple associatively – **point queries**.
     - E.g., $r.A = 25$.
  3. Locating all tuples such that the value of a given attribute lies within a specified range – **range queries**.
     - E.g., $10 \leq r.A < 25$.
- Do above evaluation for each of
  - Round robin
  - Hash partitioning
  - Range partitioning

Comparison of Partitioning Techniques (Cont.)

Round robin:

- Best suited for sequential scan of entire relation on each query.
  - All nodes have almost an equal number of tuples; retrieval work is thus well balanced between nodes.
- All queries must be processed at all nodes

Hash partitioning:

- Good for sequential access
  - Assuming hash function is good, and partitioning attributes form a key, tuples will be equally distributed between nodes
- Good for point queries on partitioning attribute
  - Can lookup single node, leaving others available for answering other queries.
- Range queries inefficient, must be processed at all nodes

©Silberschatz, Korth and Sudarshan

Comparison of Partitioning Techniques (Cont.)

Range partitioning:

- Provides data clustering by partitioning attribute value.
  - Good for sequential access
  - Good for point queries on partitioning attribute: only one node needs to be accessed.
- For range queries on partitioning attribute, one to a few nodes may need to be accessed
  - Remaining nodes are available for other queries.
  - Good if result tuples are from one to a few blocks.
  - But if many blocks are to be fetched, they are still fetched from one to a few nodes, and potential parallelism in disk access is wasted
    - Example of **execution skew**.

Handling Small Relations

- Partitioning not useful for small relations which fit into a single disk block or a small number of disk blocks
  - Instead, assign the relation to a single node, or
  - Replicate relation at all nodes
- For medium sized relations, choose how many nodes to partition across based on size of relation
- Large relations typically partitioned across all available nodes.

Types of Skew

- **Data-distribution skew:** some nodes have many tuples, while others may have fewer tuples. Could occur due to
  - **Attribute-value skew**.
    - Some partitioning-attribute values appear in many tuples
    - All the tuples with the same value for the partitioning attribute end up in the same partition.
    - Can occur with range-partitioning and hash-partitioning.
  - **Partition skew**.
    - Imbalance, even without attribute –value skew
    - Badly chosen range-partition vector may assign too many tuples to some partitions and too few to others.
    - Less likely with hash-partitioning

Types of Skew (Cont.)

- Note that **execution skew** can occur even without data distribution skew
  - E.g. relation range-partitioned on date, and most queries access tuples with recent dates
- Data-distribution skew can be avoided with range-partitioning by creating **balanced range-partitioning vectors**
- We assume for now that partitioning is **static**, that is partitioning vector is created once and not changed
  - Any change requires **repartitioning**
  - **Dynamic partitioning** once allows partition vector to be changed in a continuous manner
    - More on this later

Handling Skew in Range-Partitioning

- To create a balanced partitioning vector
  - Sort the relation on the partitioning attribute.
  - Construct the partition vector by scanning the relation in sorted order as follows.
    - After every $1/n^{th}$ of the relation has been read, the value of the partitioning attribute of the next tuple is added to the partition vector.
  - *n* denotes the number of partitions to be constructed.
  - Imbalances can result if duplicates are present in partitioning attributes.
- To reduce cost
  - Partitioning vector can be created using a random sample of tuples
  - Alternatively histograms can be used to create the partitioning vector

Histograms

- Histogram on attribute *age* of relation *person*

- **Equi-width** histograms
- **Equi-depth** histograms
  - break up range such that each range has (approximately) the same number of tuples
  - E.g. (4, 8, 14, 19)
- Assume uniform distribution within each range of the histogram
- Create partitioning vector for required number of partitions based on histogram

Virtual Node Partitioning

- Key idea: pretend there are several times (10x to 20x) as many **virtual nodes** as real nodes
  - Virtual nodes are mapped to real nodes
  - Tuples partitioned across virtual nodes using range-partitioning vector
    - Hash partitioning is also possible
- Mapping of virtual nodes to real nodes
  - **Round-robin**: virtual node *i* mapped to real node (*i* mod *n*)+1
  - **Mapping table**: mapping table *virtual_to_real_map*[] tracks which virtual node is on which real node
    - Allows skew to be handled by moving virtual nodes from more loaded nodes to less loaded nodes
    - Both data distribution skew and execution skew can be handled

Handling Skew Using Virtual Node Partitioning

- Basic idea:
  - If any normal partition would have been skewed, it is very likely the skew is spread over a number of virtual partitions
  - Skewed virtual partitions tend to get spread across a number of nodes, so work gets distributed evenly!
- Virtual node approach also allows **elasticity of storage**
  - If relation size grows, more nodes can be added and virtual nodes moved to new nodes

Dynamic Repartitioning

- Virtual node approach with a fixed partitioning vector cannot handle significant changes in data distribution over time
- Complete repartitioning is expensive and intrusive
- **Dynamic repartitioning** can be done incrementally using virtual node scheme
  - Virtual nodes that become too big can be split
    - Much like B+-tree node splits
  - Some virtual nodes can be moved from a heavily loaded node to a less loaded node
- Virtual nodes in such a scheme are often called **tablets**

Dynamic Repartitioning

- Virtual nodes in such a scheme are often called **tablets**
- Example of initial **partition table** and partition table after a split of tablet 6 and move of tablet 1

Value	Tablet ID	Node ID
2012-01-01	Tablet0	Node0
2013-01-01	Tablet1	Node1
2014-01-01	Tablet2	Node2
2015-01-01	Tablet3	Node2
2016-01-01	Tablet4	Node0
2017-01-01	Tablet5	Node1
MaxDate	Tablet6	Node1

Tablet move

Value	Tablet ID	Node ID
2012-01-01	Tablet0	Node0
2013-01-01	Tablet1	Node0
2014-01-01	Tablet2	Node2
2015-01-01	Tablet3	Node2
2016-01-01	Tablet4	Node0
2017-01-01	Tablet5	Node1
2018-01-01	Tablet6	Node1
MaxDate	Tablet7	Node1

Tablet split

Routing of Queries

- Partition table typically stored at a **master** node, and at multiple routers
- Queries are sent first to **routers**, which forward them to appropriate node
- **Consistent hashing** is an alternative fully-distributed scheme
  - without any master nodes, works in a completely peer-to-peer fashion
- **Distributed hash tables** are based on consistent hashing
  - work without master nodes or routers; each peer-node stores data and performs routing
  - See book for details of consistent hashing and distributed hash tables

Replication

- Goal: **availability** despite failures
- Data replicated at 2, often 3 nodes
- Unit of replication typically a partition (tablet)
- Requests for data at failed node automatically routed to a replica
- Partition table with each tablet replicated at two nodes

Value	Tablet ID	Node ID
2012-01-01	Tablet0	Node0,Node1
2013-01-01	Tablet1	Node0,Node2
2014-01-01	Tablet2	Node2,Node0
2015-01-01	Tablet3	Node2,Node1
2016-01-01	Tablet4	Node0,Node1
2017-01-01	Tablet5	Node1,Node0
2018-01-01	Tablet6	Node1,Node2
MaxDate	Tablet7	Node1,Node2

Basics: Data Replication

- Location of replicas
  - **Replication within a data center**
    - Handles machine failures
    - Reduces latency if copy available locally on a machine
    - Replication within/across racks
  - **Replication across data centers**
    - Handles data center failures (power, fire, earthquake, ..), and network partitioning of an entire data center
    - Provides lower latency for end users if copy is available on nearby data center

Updates and Consistency of Replicas

- Replicas must be kept consistent on update
  - Despite failures resulting in different replicas having different values (temporarily), reads must get the latest value.
  - Special concurrency control and atomic commit mechanisms to ensure consistency
- **Master replica** (**primary copy**) scheme
  - All updates are sent to master, and then replicated to other nodes
  - Reads are performed at master
  - But what if master fails? Who takes over? How do other nodes know who is the new master?
  - Details in Chapter 23

Protocols to Update Replicas

- *Two-phase commit*
  - Coming up in Chapter 23
  - Assumes all replicas are available
- *Persistent messaging*
  - Updates are sent as messages with guaranteed delivery
  - Replicas are updated asynchronously (after original transaction commits)
    - **Eventual consistency**
  - Can lead to inconsistency on reads from replicas
- *Consensus protocols*
  - Protocol followed by a set of replicas to agree on what updates to perform in what order
  - Can work even without a designated master

Parallel Indexing

- **Local index**
  - Index built only on local data
- **Global index**
  - Index built on all data, regardless of where it is stored
  - Index itself is usually partitioned across nodes
- **Global primary index**
  - Data partitioned on the index attribute
- **Global secondary index**
  - Data partitioned on the attribute other than the index attribute

Global Primary and Secondary Indices

Partition Table

231

543

765

Tablet 1

001	Zhang	Comp. Sci.	102
123	Shankar	Comp. Sci.	32
199	Brandt	History	80

Tablet 2

231	Chavez	Finance	110
445	Peltier	Physics	56
456	Levy	Physics	46

Tablet 3

543	Williams	Comp. Sci.	54
557	Sanchez	Music	38
705	Snow	Physics	0

Tablet 4

765	Brown	Comp. Sci.	58
766	Aoi	Elec. Eng.	6
987	Bourakis	Elec. Eng.	98
989	Tanaka	Biology	120

(a) Primary index on ID

Tablet 6

766	Aoi
987	Bourakis
199	Brandt
765	Brown

Partition Table

Chavez

Shankar

Tablet 7

231	Chavez
456	Levy
445	Peltier
557	Sanchez

Tablet 8

123	Shankar
705	Snow
989	Tanaka
543	Williams
001	Zhang

(b) Secondary index on name

Global Secondary Index

- Given relation r which is partitioned on K_p, to create global secondary index on attributes K_i,
  - create a relation
    - r_i^s (K_i, K_p) if K_p is unique, otherwise
    - $r_i^s(K_i, K_p, K_u)$ where (K_p, K_u) is a key for r
  - Partition r_i^s on K_i
  - At each node containing a partition of r, create index on (K_p) if K_p is a key, otherwise create index on (K_p, K_u)
  - Update the relation r_i^s on any updates to r on attributes in r_i^s

Distributed File Systems

- Google File System (GFS)
- Hadoop File System (HDFS)
- And older ones like CODA
- And more recent ones such as Google Colossus
- Basic architecture:
  - Master: responsible for metadata
  - Chunk servers: responsible for reading and writing large chunks of data
  - Chunks replicated on 3 machines, master responsible for managing replicas
  - Replication is in GFS/HDFS is within a single data center

Hadoop File System (HDFS)

- Client: sends filename to NameNode
- NameNode
  - Maps a filename to list of Block IDs
  - Maps each Block ID to DataNodes containing a replica of the block
  - Returns list of BlockIDs along with locations of their replicas
- DataNode:
  - Maps a Block ID to a physical location on disk
  - Sends data back to client

Hadoop Distributed File System

Hadoop Distributed File System (HDFS)

- Modeled after Google File System (GFS)
- Single Namespace for entire cluster
- Data Coherency
  - Write-once-read-many access model
  - Client can only append to existing files
- Files are broken up into blocks
  - Typically 64 MB block size
  - Each block replicated on multiple (e.g. 3) DataNodes
- Client
  - Finds location of blocks from NameNode
  - Accesses data directly from DataNode

Limitations of GFS/HDFS

- Central master becomes bottleneck
  - Keep directory/inode information in memory to avoid IO
  - Memory size limits number of files
  - Colossus file system supports distributed master
    - With smaller (1MB) block size
- File system directory overheads per file
  - Not appropriate for storing very large number of objects
- File systems do not provide consistency guarantees
  - File systems cache blocks locally
  - Ideal for write-once and and append only data
  - Can be used as underlying storage for a data storage system
    - E.g. BigTable uses GFS underneath

Sharding

Sharding (recall from Chapter 10)

- Divide data amongst many cheap databases (MySQL/PostgreSQL)
- Manage parallel access in the application
  - Partition tables map keys to nodes
  - Application decides where to route storage or lookup requests
- Scales well for both reads and writes
- Limitations
  - Not transparent
    - application needs to be partition-aware
    - AND application needs to deal with replication
  - (Not a true parallel database, since parallel queries and transactions spanning nodes are not supported)

Key Value Storage Systems

Recall from Chapter 10

- Key-value stores may store
  - **uninterpreted bytes**, with an associated key
    - E.g. Amazon S3, Amazon Dynamo
  - **Wide-column stores** (can have arbitrarily many attribute names) with associated key
      - Google BigTable, Apache Cassandra, Apache HBase, Amazon DynamoDB, Microsoft Azure Cloud store
      - Allows some operations (e.g. filtering) to execute on storage node
      - Google MegaStore and Spanner and Yahoo! PNUTS/Sherpa support relational schema
  - JSON
    - MongoDB, CouchDB (document model)
- **Document stores** store semi-structured data, typically JSON

Typical Data Storage Access API

- Basic API access:
  - get(key) -- Extract the value given a key
  - put(key, value) -- Create or update the value given its key
  - delete(key) -- Remove the key and its associated value
  - execute(key, operation, parameters) -- Invoke an operation to the value (given its key) which is a special data structure (e.g., List, Set, Map .... Etc.)
- Extensions to add range queries, version numbering, etc.

Data Storage Systems vs. Databases

Distributed data storage implementations:

- May have limited support for relational model (no schema, or flexible schema)
- But usually do provide flexible schema and other features
  - Structured objects e.g. using JSON
  - Multiple versions of data items
- Often do not support referential integrity constraints
- Often provide no support or limited support for transactions
  - But some do!
- Provide only lowest layer of database

Data Representation

- In wide-column stores like BigTable, records may be vertically partitioned by attribute (*columnar storage*)
  - (record-identifier, attribute-name) forms a key
- Multiple attributes may be stored in one file (**column family**)
  - In BigTable records are sorted by key, ensuring all attributes of a logical record in that file are contiguous
    - Attributes can be fetched by a prefix/range query
  - Record-identifiers can be structured hierarchically to exploit sorting
    - E.g. url: www.cs.yale.edu/people/silberschatz.html can be mapped to record identifier edu.yale.cs.www/people/silberschatz.html
    - Now all records for cs.yale.edu would be contiguous, as would all records for yale.edu

Storing and Retrieving Data

Architecture of BigTable key-value store

- Table split into multiple tablets
- Tablet servers manage tablets, multiple tablets per server. Each tablet is 100-200 MB
  - Each tablet controlled by only one server
  - Tablet server splits tablets that get too big
- Master responsible for load balancing and fault tolerance
- All data and logs stored in GFS
  - Leverage GFS replication/fault tolerance
  - Data can be accessed if required from any node to aid in recovery

Architecture of Key-Value Store (modelled after Yahoo! PNUTS)

Geographically Distributed Storage

- Many storage systems today support geographical distribution of storage
  - Motivations: Fault tolerance, latency (close to user), governmental regulations
- Latency of replication across geographically distributed data centers much higher than within data center
  - Some key-value stores support **synchronous replication**
    - Must wait for replicas to be updated before committing an update
  - Others support **asynchronous replication**
    - update is committed in one data center, but sent subsequently (in a fault-tolerant way) to remote data centers
    - Must deal with small risk of data loss if data center fails.

Index Structures in Key-Value Stores

- Storing data in each tablet in clustered on key benefits range queries
- B^+-tree file organization works well for range queries
- Write optimized trees, especially LSM trees (Section 24.2) work well for updates as well as for range queries
  - Used in BigTable, HBase and many other key-value stores
- Some key-value stores organize records on each node by hashing, or just build a hash index on the records

Transactions in Key-Value Stores

- Most key-value stores don't support full-fledged transactions
  - But treat each update as a transaction, to ensure integrity of internal data structure
- Some key-value stores allow multiple updates to one data item to be committed as a single transaction
- Without support for transactions, secondary indices cannot be maintained consistently
  - Some key-value stores do not support secondary indices at all
  - Some key-value stores support asynchronous maintenance of secondary indices
- Some key-value stores support ACID transactions across multiple data items along with two-phase commit across nodes
  - Google MegaStore and Spanner
- More details in Chapter 23

Transactions in Key-Value Stores

- Some key-value stores support concurrency control via locking and snapshots
- Some support *atomic test-and-set* and *increment* on data items
  - Others do not support concurrency control
- Key-value stores implement recovery protocols based on logging to ensure durability
  - Log must be replicated, to ensure availability in spite of failures
- Distributed file systems are used to store log and data files in some key-value stores such as BigTable, HBase
  - But distributed file systems do not support (atomic) updates of files except for appends
  - LSM trees provide a nice way to index data without requiring updates of files
- Some systems use persistent messaging to manage logs
- Details in Chapter 23

Querying and Performance Optimizations

- Many key-value stores do not provide a declarative query language
- Applications must manage joins, aggregates, etc on their own
- Some applications avoid computing joins at run-time by creating (what is in effect) materialized views
  - Application code maintains materialized views
  - E.g. If a user makes a post, the application may add a summary of the post to the data items representing all the friends of the user
- Many key-value stores allow related data items to be stored together
  - Related data items form an **entity-group**
  - e.g. user data item along with all posts of that user
  - Makes joining the related tuples very cheap
- Other functionality includes
  - Stored procedures executed at the nodes storing the data
  - Versioning of data, along with automated deletion of old versions

END OF CHAPTER

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Chapter 22: Parallel and Distributed Query Processing

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Chapter 22: Parallel And Distributed Query Processing

- Overview
- Parallel Sort
- Parallel Join
- Other Operations
- Parallel Evaluation of Query Plans
- Query Processing on Shared Memory
- Query Optimization
- Distributed Query Processing

 ©Silberschatz, Korth and Sudarshan

Parallel Query Processing

- Different queries/transactions can be run in parallel with each other.
  - **Interquery parallelism**
  - Concurrency control takes care of conflicts in case of updates
  - More on parallel transaction processing in Chapter 23
  - Focus in this chapter is on read-only queries
- Individual relational operations (e.g., sort, join, aggregation) can be executed in parallel
  - data can be partitioned and each processor can work independently on its own partition.
- Queries are expressed in high level language (SQL, translated to relational algebra)
  - makes parallelization easier.

Intraquery Parallelism

- **Intraquery parallelism**: execution of a single query in parallel on multiple processors/disks; important for speeding up long-running queries.
- Two complementary forms of intraquery parallelism:
  - **Intraoperation Parallelism** – parallelize the execution of each individual operation in the query
    - Supports high degree of parallelism
  - **Interoperation Parallelism** – execute the different operations in a query expression in parallel.
    - Limited degree of parallelism

Parallel Processing of Relational Operations

- Our discussion of parallel algorithms assumes:
  - *read-only* queries
  - shared-nothing architecture
  - *n* nodes, N_1, ..., N_n
    - *Each assumed to have disks and processors*.
  - Initial focus on parallelization to a shared-nothing node
    - Parallel processing within a shared memory/shared disk node discussed later
  - Shared-nothing architectures can be efficiently simulated on shared-memory and shared-disk systems.
    - Algorithms for shared-nothing systems can thus be run on shared-memory and shared-disk systems.
    - However, some optimizations may be possible.

INTRAOPERATION PARALLELISM

Range Partitioning

Redistribute using partitioning vector:
100, 175, 300, 500, 800

Range-Partitioning Parallel Sort

1) Redistribute using **partitioning vector**: 100, 175, 300, 500, 800

2) (External) sort locally at each node
3) Merge if output required at one node

Parallel Sort

(a) Range Partitioning Sort

(b) Parallel External Sort-Merge

Parallel Sort

Range-Partitioning Sort

- Choose nodes N_1, ..., N_m, where $m \leq n$ -1 to do sorting.
- Create range-partition vector with m-1 entries, on the sorting attributes
- Redistribute the relation using range partitioning
- Each node N_i sorts its partition of the relation locally.
  - Example of **data parallelism: e**ach node executes same operation in parallel with other nodes, without any interaction with the others.
- Final merge operation is trivial: range-partitioning ensures that, if i < j, all key values in node N_i are all less than all key values in N_j.

Parallel Sort (Cont.)

Parallel External Sort-Merge

- Assume the relation has already been partitioned among nodes N1, ..., Nn (in whatever manner).
- Each node Ni locally sorts the data (using local disk as required)
- The sorted runs on each node are then merged in parallel:
  - The sorted partitions at each node Ni are range-partitioned across the processors N1, ..., Nm.
  - Each node Ni performs a merge on the streams as they are received, to get a single sorted run.
  - The sorted runs on nodes N1,..., Nm are concatenated to get the final result.
- Algorithm as described vulnerable to execution skew
  - all nodes send to node 1, then all nodes send data to node 2, …
  - Can be modified so each node sends data to all other nodes in parallel (block at a time)

Partitioned Parallel Join

Step 1: Partition r Step 2: Partition s

Step 3: Each node N_i computes $r_i' \bowtie s_i'$

Partition using range or hash partitioning, on join attributes

Partitioned Parallel Join (Cont.)

- For equi-joins and natural joins, it is possible to *partition* the two input relations across the processors, and compute the join locally at each processor.
- Can use either *range partitioning* or *hash partitioning*.
- *r* and *s* must be partitioned on their join attributes *r*.A and *s*.B), using the same range-partitioning vector or hash function.
- Join can be computed at each site using any of
  - Hash join, leading to **partitioned parallel hash join**
  - Merge join, leading to **partitioned parallel merge join**
  - Nested loops join, leading to **partitioned parallel nested-loops join** or **partitioned parallel index nested-loops join**

Partitioned Parallel Hash-Join

Parallelizing partitioned hash join:

- A hash function h_1 takes the join attribute value of each tuple in s and maps this tuple to one of the n nodes.
- As tuples of relation s are received at the destination nodes, they are partitioned further using another hash function, h_2, which is used to compute the hash-join locally.
- Repeat above for each tupe in r.
- Each node N_i executes the build and probe phases of the hash-join algorithm on the local partitions r_i and s_i of r and s to produce a partition of the final result of the hash-join.
- Note: Hash-join optimizations can be applied to the parallel case
  - e.g., the hybrid hash-join algorithm can be used to cache some of the incoming tuples in memory and avoid the cost of writing them and reading them back in.

Fragment-and-Replicate Joins

Asymmetric and Symmetric Fragment-and-Replicate Joins

Fragment-and-Replicate Join

- Partitioning not possible for some join conditions
  - e.g., non-equijoin conditions, such as r.A > s.B.
- For joins were partitioning is not applicable, parallelization can be accomplished by **fragment and replicate technique**
- Special case – **asymmetric fragment-and-replicate**:
  - One of the relations, say r, is partitioned; any partitioning technique can be used.
  - The other relation, s, is replicated across all the processors.
  - Node Ni then locally computes the join of ri with all of s using any join technique.
  - Also referred to as **broadcast join**

Fragment-and-Replicate Join (Cont.)

- Both versions of fragment-and-replicate work with any join condition, since every tuple in *r* can be tested with every tuple in *s*.
- Usually has a higher cost than partitioning, since one of the relations (for asymmetric fragment-and-replicate) or both relations (for general fragment-and-replicate) have to be replicated.
- Sometimes asymmetric fragment-and-replicate is preferable even though partitioning could be used.
  - E.g., if *s* is small and *r* is large, and *r* is already partitioned, it may be cheaper to replicate *s* across all nodes, rather than repartition *r* and *s* on the join attributes.
- Question: how do you implement left outer join using above join techniques?

Handling Skew

- Skew can significantly slow down parallel join
- **Join skew avoidance**
  - Balanced partitioning vector
  - Virtual node partitioning
- **Dynamic handling of join skew**
  - Detect overloaded physical nodes
  - If a physical node has no remaining work, take on a waiting task (virtual node) currently assigned to a different physical node that is overloaded
  - Example of **work stealing**
    - Cheaper to implement in shared memory system, but can be used even in shared nothing/shared disk system

Other Relational Operations

Selection $\sigma_\theta(r)$

- If θ is of the form $a_i = v$, where a_i is an attribute and v a value.
  - If r is partitioned on a_i the selection is performed at a single node.
- If θ is of the form $l <= a_i <= u$ (i.e., θ is a range selection) and the relation has been range-partitioned on a_i
  - Selection is performed at each node whose partition overlaps with the specified range of values.
- In all other cases: the selection is performed in parallel at all the nodes.

Other Relational Operations (Cont.)

- **Duplicate elimination**
  - Perform by using either of the parallel sort techniques
    - eliminate duplicates as soon as they are found during sorting.
  - Can also partition the tuples (using either range- or hash-partitioning) and perform duplicate elimination locally at each node.

- **Projection**
  - Projection without duplicate elimination can be performed as tuples are read from disk, in parallel.
  - If duplicate elimination is required, any of the above duplicate elimination techniques can be used.

Grouping/Aggregation

- **Step 1**: Partition the relation on the grouping attributes
- **Step 2**: Compute the aggregate values locally at each node.
- **Optimization:** Can reduce cost of transferring tuples during partitioning by **partial aggregation** before partitioning
  - For distributive aggregate
  - Can be done as part of run generation
  - Consider the **sum** aggregation operation:
    - Perform aggregation operation at each node N_i on those tuples stored its local disk
      - results in tuples with partial sums at each node.
    - Result of the local aggregation is partitioned on the grouping attributes, and the aggregation performed again at each node N_i to get the final result.
  - Fewer tuples need to be sent to other nodes during partitioning.

Map and Reduce Operations

Map and Reduce Operations

- Map and reduce **workers**
  - Threads/processes that execute map and reduce functions
- Map and reduce **tasks**
  - Units of map and reduce work
  - Many more tasks than workers
    - Similar to virtual node partitioning
- Skew handling
  - **Straggler tasks**
    - Can be handled by initiating an extra copy of the task at another node
  - Partial aggregation (combiners) helps reduce skew at reduce nodes

PARALLEL EVALUATION OF QUERY PLANS

Interoperator Parallelism

- **Pipelined parallelism**
  - Consider a join of four relations
    - $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$
  - Set up a pipeline that computes the three joins in parallel

Each of these operations can execute in parallel, sending result tuples it computes to the next operation even as it is computing further results

- Provided a pipelineable join evaluation algorithm (e.g. indexed nested loops join) is used

Pipelined Parallelism

- Push model of computation appropriate for pipelining in parallel databases
- Buffer between consumer and producer
- Can batch tuples before sending to next operator
  - Reduce number of messages,
  - reduce contention on shared buffers

(a) Producer-consumer in shared memory

(b) Producer-consumer across a network

Utility of Pipeline Parallelism

- Limitations
  - Does not provide a high degree of parallelism since pipeline chains are not very long
  - Cannot pipeline operators which do not produce output until all inputs have been accessed (e.g. aggregate and sort)
  - Little speedup is obtained for the frequent cases of skew in which one operator's execution cost is much higher than the others.
- But pipeline parallelism is still very useful since it avoids writing intermediate results to disk

Independent Parallelism

- **Independent parallelism**
  - Consider a join of four relations

    $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$
    - Let N_1 be assigned the computation of
      temp1 = $r_1 \bowtie r_2$
    - And N_2 be assigned the computation of temp2 = $r_3 \bowtie r_4$
    - And N_3 be assigned the computation of temp1 $\bowtie$ $temp_2$
    - N_1 and N_2 can work **independently in parallel**
    - N_3 has to wait for input from N_1 and N_2
      - Can pipeline output of N_1 and N_2 to N_3, combining independent parallelism and pipelined parallelism
  - Does not provide a high degree of parallelism
    - useful with a lower degree of parallelism.
    - less useful in a highly parallel system,

Exchange Operator

- Repartitioning implemented using the **exchange operator**
  - **Partition** and **merge** steps

Exchange Operator Model

- Movement of data encapsulated in **exchange operator**
- Partitioning of data can be done by
  - Hash partitioning
  - Range partitioning
  - Replicating data to all nodes (called **broadcasting**)
  - Sending all data to a single node
- Destination nodes can receive data from multiple source nodes. Incoming data can be merged by:
  - **Random merge**
  - **Ordered merge**
- Other operators in a plan can be unaware of parallelism
  - **Data parallelism**: each operator works purely on local data
  - Not always best way, but works well in most cases

Parallel Plans Using Exchange Operator

- Range partitioning sort:
  1. Exchange operator using range partitioning, followed by
  2. Local sort
- Parallel external sort merge
  1. Local sort followed by
  2. Exchange operator with ordered merge
- Partitioned join
  1. Exchange operator with hash or range partitioning, followed by
  2. Local join
- Asymmetric fragment and replicate
  1. Exchange operator using broadcast replication, followed by
  2. Local join
- Exchange operator can also implement push model, with batching

Parallel Plans

Dashed boxes denote pipelined segment

(a) Logical Query

(b) Sequential Plan

(c) Parallel Plan

Parallel Plans (Cont.)

(c) Parallel Plan

(c) Parallel Plan with Partial Aggregation

Fault Tolerance in Query Plans

- **Alternative 1: Re-execute the query on failure**
  - Works well if mean time to failure >> query execution time
    - Good for medium scale parallelism with 100's of machines
  - Works badly on massively parallel execution of long queries
    - Where probability of some node failing during execution of a single query is high

Fault Tolerance in Query Plans

- **Alternative 2: Re-execute work of only failed nodes**
  - Works well if consumers process data only after it is completely generated
    - Just discard partial data
    - Used in map-reduce implementations
  - Problems arise if consumers process data as it is generated (pipelined execution)
    - Only new tuples must be consumed from re-execution
    - Re-execution must generate tuples in exactly same order to efficiently determine which tuples are new
- **Straggler nodes** (nodes that are running slow) can be treated similar to failed nodes

Fault Tolerance in Map-Reduce

Fault Tolerance in Map Reduce Implementation

- Map workers writes data to local disk
  - Cheaper than writing to distributed file system
- When task is complete, data is sent to reducers
- Reducers use data only after it is fully received
- On map worker failure:
  - Reexecute map tasks on a new node
  - Reducers get data from new node, discarding partially received data (if any) from failed node
- On reduce worker failure
  - Reexecute reduce task on new node
  - Re-fetch data from map nodes
- On completion of a map-reduce phase, result is written to distributed file system
  - Replication ensures result is safe from failures

Fault Tolerant Query Execution

- Overheads of fault-tolerant query execution
  - Materialization cost
  - Each step has to wait till the previous step finishes
    - Stragglers can cause significant delays
- Pipelined execution can avoid these costs
  - But harder to make pipelined execution fault-tolerant
    - E.g. duplication of tuples when failed task is reexecuted
- Apache Spark uses concept of Resilient Distributed Datasets (RDDs)
  - Data can be replicated in memory/on disk
  - But intermediate results are not materialized
    - Query nodes can be reexecuted to regenerate results

Query Processing in Shared Memory Systems

- Parallel query processing techniques discussed so far can be optimized if data is in shared memory
- Shared memory parallel processing usually based on **threads**, instead of processes
  - Usually number of threads = number of cores * number off hardware threads per core

Query Processing in Shared Memory Systems

Optimized algorithms for shared memory

- With asymmetric fragment-and-replicate join, the smaller relation can be in shared memory, instead of being replicated for each thread
- Hash join can be done by
  - Partitioning build relation to each thread, OR
  - **Shared build relation** with singe index, in shared memory
    - Probe relation can be partitioned into small pieces (a.k.a. morsels)
    - Each thread processes one piece of the probe at a time, in parallel with other threads
    - Shared index construction can be parallelized, but carefully
      - Multiple threads may try to write to same location in shared memory
      - Atomic instruction (test-and-set/compare-and-swap) can be used to add entries to hash table list

Query Processing in Shared Memory Systems

- Skew can be handled by **work stealing**
  - Idle processors can take up tasks allocated to other processors
  - Virtual node partitioning allows tasks to be broken into small pieces
    - Cost of reallocating a partition is low in shared memory
    - Even simpler if shared build relation is used
      - only probe relation partitioned need to be reassigned
- Query processing algorithms should be are of NUMA: Non-uniform memory access
  - Each thread scheduled as far as possible on same core every time it runs
  - Data structures used by only 1 thread allocated in memory local to the core on which the thread is running

©Silberschatz, Korth and Sudarshan

Query Processing in Shared Memory Systems

- Cache-conscious algorithms used in main-memory centralized query processing should also be used in shared-memory systems
- **Single Instruction Multiple Data** (**SIMD**) parallelism is increasingly used
  - In GPUs as well as Intel Xeon Phi co-processors
  - *Vector processing*
- Vector processing can be used for relational operations

QUERY OPTIMIZATION FOR PARALLEL QUERY EXECUTION

Query Optimization For Parallel Execution

- Query optimization in parallel databases is significantly more complex than query optimization in sequential databases.
  - Different options for partitioning inputs and intermediate results
  - Cost models are more complicated, since we must take into account partitioning costs and issues such as skew and resource contention.

Parallel Query Plan Space

A parallel query plan must specify

- How to parallelize each operation, including which algorithm to use, and how to partition inputs and intermediate results (using exchange operators)
- How the plan is to be **scheduled**
  - How many nodes to use for each operation
  - What operations to pipeline within same node or different nodes
  - What operations to execute independently in parallel, and
  - What operations to execute sequentially, one after the other.
- E.g. In query ${}_{r.A}\gamma_{\text{sum}(s.C)}(r \bowtie_{r.A=s.A \;\; r.B=s.B} s)$
  - Partitioning *r* and *s* on (*A,B*) for join will require repartitioning for aggregation
  - But partitioning *r* and *s* on (*A*) for join will allow aggregation with no further repartitioning
- Query optimizer has to choose best plan taking above issues into account

Cost of Parallel Query Execution

- **Resource consumption cost model**
  - used for centralized databases
- **Response time cost model**
  - attempts to better estimate the time to completion of a query
  - E.g. If an operation performs I/O operations in parallel with CPU execution, the response time
    $T = max$(CPU cost, I/O cost)
    - Resource consumption cost model uses (CPU cost + I/O cost).
  - E.g., if two operations o_1 and o_2 are in a pipeline, with CPU and I/O costs c_1; io_1 and c_2; io_2 respectively, then response time
    $T = max(c_1 + c_2, io_1 + io_2)$.
  - Operators in parallel: $T = max\ (T_1, T_1, \ldots, T_n)$
    - Skew is an issue

Cost of Parallel Query Execution (Cont.)

- Response time cost model would have to take into account
  - **Start-up costs** for initiating an operation on multiple nodes
  - **Skew** in distribution of work
- Response time cost model better suited for parallel databases
  - But not used much since it increases cost of query optimization

Choosing Query Plans

- The number of parallel evaluation plans from which to choose from is much larger than the number of sequential evaluation plans
  - Many alternative partitioning options
  - Choosing a good physical organization (partitioning technique) is important to speed up queries.
- Two alternatives often used for choosing parallel plans:
  - First choose most efficient sequential plan and then choose how best to parallelize the operations in that plan
    - Heuristic, since best sequential plan may not lead to best parallel plan
  - Parallelize every operation across all nodes
    - Use exchange operator to perform (re)partitioning
    - Use standard query optimizer with extended cost model

Physical Schema

- Partitioning scheme important for queries
- **Colocate data** that is accessed together
  - E.g., all records for a particular user
  - E.g., *student* record with all *takes* records of the student
- Store multiple copies of a relation, partitioned on different attributes
  - E.g., extra copy of *takes* partitioned on *(course id, year, semester, sec id)* for colocation with *section* record
- Materialized views to avoid joins at query time
  - Materialized view itself is stored partitioned across nodes
  - Speeds up queries, but extra cost for updates
  - Extra copy of materialized view may be stored partitioned on different attributes
- *See book for details of parallel maintenance of materialized views*

STREAMING DATA

Streaming Data

- Real-time analytics is increasingly important
- Online processing of incoming data
- But data must also be stored for later processing
- Architecture alternatives:
  - **Lambda architecture**: sends a copy of data to real time stream analytics system, and another copy to data storage system
  - Increasingly, same system provides storage and real time stream analytics

Logical Routing of Streams

(a) DAG representation of streaming data flow

(b) Publish-subscribe representation of streaming data flow

Parallel Processing of Streaming Data

- Logical routing of tuples is done by creating a Directed Acyclic Graph (DAG)
  - Can be done by creating a configuration file
    - Called a *topology* in the Apache Storm system
  - OR, can be done dynamically by using publish-subscribe system
    - E.g., Apache Kafka
    - Producers ***publish*** to **topic/stream**, consumers ***subscribe*** to topic/streams

Parallel Processing of Streaming Data

- Physical DAG reflects parallel execution of operator instances
- Parallel implementation of publish-subscribe system
  - Each topic/stream is partitioned (**topic-partition**)
- Multiple instances of each operator, running in parallel
  - Each subscribes to one or more partitions of a topic
- In Kafka, multiple instances of an operator register with an associated **consumer group**
  - Each topic-partition sent to a single consumer from the consumer group
- Streaming operators often need to store state
  - Can be stored locally (cheaper) or in a parallel data-store (better for fault tolerance)

Fault Tolerance

- Possible semantics when dealing with failures
  - **At-least once**
  - **At-most once**
  - **Exactly once**
- Can be implemented in the publish-subscribe/routing system
  - Need to store tuples, including intermediate results, persistently
  - Can lower overhead by **checkpointing** operator state, and replaying operator from checkpoint, instead of persisting operator output
    - But need to ensure duplicates are removed
- Replication of operators: Copy of operator running concurrently with original
  - Similar to hot-spare
  - Allows instant recovery from failure

Distributed Query Processing

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Data Integration From Multiple Sources

- Many database applications require data from multiple databases
- A **federated database system** is a software layer on top of existing database systems, which is designed to manipulate information in heterogeneous databases
  - Creates an illusion of logical database integration without any physical database integration
  - Each database has its **local schema**
  - **Global schema** integrates all the local schema
    - **Schema integration**
  - Queries can be issued against global schema, and translated to queries on local schemas
    - Databases that support common schema and queries, but not updates, are referred to as **mediator** systems

Data Integration From Multiple Sources

- **Data virtualization**
  - Allows data access from multiple databases, but without a common schema
- **External data** approach: allows database to treat external data as a database relation (**foreign tables**)
  - Many databases today allow a local table to be defined as a view on external data
  - SQL Management of External Data (SQL MED) standard
- **Wrapper** for a data source is a view that translates data from local to a global schema
  - Wrappers must also translate updates on global schema to updates on local schema

Data Warehouses and Data Lakes

- **Data warehouse** is an alternative to data integration
  - Migrates data to a common schema, avoiding run-time overhead
  - Cost of translating schema/data to a common warehouse schema can be significant
- **Data lake**: architecture where data is stored in multiple data storage systems, in different storage formats, but which can be queried from a single system.

Schema and Data Integration

- **Schema integration**: creating a unified conceptual schema
  - Requires creation of **global schema**, integrating a number of **local schema**
- **Global-as-view approach**
  - At each site, create a view of local data, mapping it to the global schema
  - Union of local views is the global view
  - Good for queries, but not for updates
    - E.g., which local database should an insert go to?
- **Local-as-view approach**
  - Create a view defining contents of local data as a view of global data
    - Site stores local data as before, the view is for update processing
  - Updates on global schema are mapped to updates to the local views
- *See book for more details with an example*

Unified View of Data

- Agreement on a common data model
  - Typically the relational model
- Agreement on a common conceptual schema
  - Different names for same relation/attribute
  - Same relation/attribute name means different things
- Agreement on a single representation of shared data
  - E.g., data types, precision,
  - Character sets
    - ASCII vs EBCDIC
    - Sort order variations
- Agreement on units of measure

Unified View of Data (Cont.)

- Variations in names
  - E.g., Köln vs Cologne, Mumbai vs Bombay
- One approach: globally unique naming system
  - E.g. GeoNames database (www.geonames.org)
- Another approach: specification of name equivalences
  - E.g. used in the Linked Data project supporting integration of a large number of databases storing data in RDF data

Query Processing Across Data Sources

- Several issues in query processing across multiple sources
- Limited query capabilities
  - Some data sources allow only restricted forms of selections
    - E.g., web forms, flat file data sources
  - Queries have to be broken up and processed partly at the source and partly at a different site
- Removal of duplicate information when sites have overlapping information
  - Decide which sites to execute query
- Global query optimization

Join Locations and Join Ordering

- Consider the following relational algebra expression in which the three relations are neither replicated nor fragmented

  $r1 \bowtie r2 \bowtie r3$

- *r1* is stored at site S_1
- *r2* at S_2
- *r3* at S_3
- For a query issued at site S_I, the system needs to produce the result at site S_I

Possible Query Processing Strategies

- Ship copies of all three relations to site S_I and choose a strategy for processing the entire locally at site S_I.
  - Ship a copy of the *r1* relation to site S_2 and compute $temp_1 = r1 \bowtie r2$ *at* S_2.
  - Ship $temp_1$ from S_2 to S_3, and compute $temp_2 = temp_1 \bowtie r3$ at S_3
  - Ship the result $temp_2$ to S_I.
- Devise similar strategies, exchanging the roles S_1, S_2, S_3
- Must consider following factors:
  - amount of data being shipped
  - cost of transmitting a data block between sites
  - relative processing speed at each site

Semijoin Strategy

- Let r_1 be a relation with schema R_1 stores at site S_1

  Let r_2 be a relation with schema R_2 stores at site S_2
- Evaluate the expression $r_1 \bowtie r_2$ and obtain the result at S_1.
  1. Compute $temp_1 \leftarrow \Pi_{R1 \cap R2}$ (r1) at S1.
  2. Ship $temp_1$ from S_1 to S_2.
  3. Compute $temp_2 \leftarrow r_2 \bowtie$ temp1 at S_2
  4. Ship $temp_2$ from S_2 to S_1.
  5. Compute $r_1 \bowtie temp_2$ at S_1. This is the same as $r_1 \bowtie r_2$.

Semijoin Reduction

- The **semijoin** of r_1 with r_2, is denoted by:

$$r_1 \ltimes r_2 \quad \Pi_{R1} (r_1 \bowtie r_2)$$

- Thus, $r_1 \ltimes r_2$ selects those tuples of r_1 that contributed to $r_1 \bowtie r_2$.
- In step 3 above, $temp_2 = r_2 \ltimes r_1$.
- For joins of several relations, the above strategy can be extended to a series of semijoin steps.
- Semijoin can be computed approximately by using a Bloom filter
  - For each tuple of r_2 compute hash value on join attribute; if hash value is i, and set bit i of the bitmap
  - Send bitmap to site containing r_1
  - Fetch only tuples of r_1 whose join attribute value hashes to a bit that is set to 1 in the bitmap
  - Bloom filter is an optimized bitmap filter structure (Section 24.1)

Distributed Query Optimization

- New physical property for each relation: location of data
- Operators also need to be annotated with the site where they are executed
  - Operators typically operate only on local data
  - Remote data is typically fetched locally before operator is executed
- Optimizer needs to find best plan taking data location and operator execution location into account.

Distributed Directory Systems

- Distributed directory systems are widely used
  - Internet Domain Name Service (**DNS**)
  - **Lightweight Directory Access Protocol** (**LDAP**)
    - Widely used to store organizational data, especially user profiles
- Data in a distributed directory system are stored and controlled in a distributed hierarchical manner
  - E.g. data about yale.edu is provided by a Yale, and about IIT Bombay by an IIT Bombay server
  - Data can be queries in a uniform manner regardless of where it is stored
    - Queries get forwarded to the site where the information is stored

End of Chapter

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Join Strategies that Exploit Parallelism

- Consider $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$ where relation ri is stored at site S_i. The result must be presented at site S_1.
- r_1 is shipped to S_2 and $r_1 \bowtie r_2$ is computed at S_2: simultaneously r_3 is shipped to S_4 and $r_3 \bowtie r_4$ is computed at S_4
- S_2 ships tuples of $(r_1 \bowtie r_2)$ to S_1 as they produced;
  S_4 ships tuples of $(r_3 \bowtie r_4)$ to S_1
- Once tuples of $(r_1 \bowtie r_2)$ and $(r_3 \bowtie r_4)$ arrive at S_1 $(r_1 \bowtie r_2) \bowtie (r_3 \quad r_4)$ is computed in parallel with the computation of $(r_1 \bowtie r_2)$ at S_2 and the computation of $(r_3 \bowtie r_4)$ at S_4.

Mediator Systems

- **Mediator** systems are systems that integrate multiple heterogeneous data sources by providing an integrated global view, and providing query facilities on global view
  - Unlike full fledged multidatabase systems, mediators generally do not bother about transaction processing
  - But the terms mediator and multidatabase are sometimes used interchangeably
  - The term **virtual database** is also used to refer to mediator/multidatabase systems

frequency
50
40
30
20
10
1–5
6–10
11–15
16–20
21–25
value

Distributed Query Processing

- For centralized systems, the primary criterion for measuring the cost of a particular strategy is the number of disk accesses.
- In a distributed system, other issues must be taken into account:
  - The cost of a data transmission over the network.
  - The potential gain in performance from having several sites process parts of the query in parallel.

Query Transformation

- Translating algebraic queries on fragments.
  - It must be possible to construct relation *r* from its fragments
  - Replace relation *r* by the expression to construct relation *r* from its fragments
- Consider the horizontal fragmentation of the *account* relation into

  $account_1 = \sigma_{branch_name = \text{“Hillside”}} (account)$

  $account_2 = \sigma_{branch_name = \text{“Valleyview”}} (account)$
- The query $\sigma_{branch_name = \text{“Hillside”}} (account)$ becomes

  $\sigma_{branch_name = \text{“Hillside”}} (account_1 \cup account_2)$

  which is optimized into

  $\sigma_{branch_name = \text{“Hillside”}} (account_1) \cup \sigma_{branch_name = \text{“Hillside”}} (account_2)$

Example Query (Cont.)

- Since $account_1$ has only tuples pertaining to the Hillside branch, we can eliminate the selection operation.
- Apply the definition of $account_2$ to obtain

  $\sigma_{branch_name = \text{“Hillside”}} (\sigma_{branch_name = \text{“Valleyview”}} (account))$

- This expression is the empty set regardless of the contents of the *account* relation.
- Final strategy is for the Hillside site to return $account_1$ as the result of the query.

Advantages

- Preservation of investment in existing
  - hardware
  - system software
  - Applications
- Local autonomy and administrative control
- Allows use of special-purpose DBMSs
- Step towards a unified homogeneous DBMS
  - Full integration into a homogeneous DBMS faces
    - Technical difficulties and cost of conversion
    - Organizational/political difficulties
      - Organizations do not want to give up control on their data
      - Local databases wish to retain a great deal of **autonomy**

End of Chapter 22

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Cost of Parallel Evaluation of Operations

- If there is no skew in the partitioning, and there is no overhead due to the parallel evaluation, expected speed-up will be 1/n
- If skew and overheads are also to be taken into account, the time taken by a parallel operation can be estimated as

$$T_{part} + T_{asm} + \max(T_1, T_1, \ldots, T_n)$$

  - T_{part} is the time for partitioning the relations
  - T_{asm} is the time for assembling the results
  - T_i is the time taken for the operation at node N_i
    - this needs to be estimated taking into account the skew, and the time wasted in contention.

Chapter 23: Parallel and Distributed Transaction Processing

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Distributed Transactions

- **Local transactions**
  - Access/update data at only one database
- **Global transactions**
  - Access/update data at more than one database
- Key issue: how to ensure ACID properties for transactions in a system with global transactions spanning multiple database

Distributed Transactions

- Transaction may access data at several sites.
  - Each site has a local **transaction manager**
  - Each site has a **transaction coordinator**
    - Global transactions submitted to any transaction coordinator

Distributed Transactions

- Each transaction coordinator is responsible for:
  - Starting the execution of transactions that originate at the site.
  - Distributing subtransactions at appropriate sites for execution.
  - Coordinating the termination of each transaction that originates at the site
    - transaction must be committed at all sites or aborted at all sites.
- Each local transaction manager responsible for:
  - Maintaining a log for recovery purposes
  - Coordinating the execution and commit/abort of the transactions executing at that site.

System Failure Modes

- Failures unique to distributed systems:
  - Failure of a site.
  - Loss of massages
    - Handled by network transmission control protocols such as TCP-IP
  - Failure of a communication link
    - Handled by network protocols, by routing messages via alternative links
  - **Network partition**
    - A network is said to be **partitioned** when it has been split into two or more subsystems that lack any connection between them
      - Note: a subsystem may consist of a single node
- Network partitioning and site failures are generally indistinguishable.

Commit Protocols

- Commit protocols are used to ensure atomicity across sites
  - a transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.
    - cannot have transaction committed at one site and aborted at another
- The *two-phase commit* (2PC) protocol is widely used
- *Three-phase commit* (3PC) protocol avoids some drawbacks of 2PC, but is more complex
- *Consensus protocols* solve a more general problem, but can be used for atomic commit
  - More on these later in the chapter
- The protocols we study all assume **fail-stop** model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.
  - Protocols that can tolerate some number of malicious sites discussed in bibliographic notes online

Two Phase Commit Protocol (2PC)

- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.
- The protocol involves all the local sites at which the transaction executed
- Protocol has two phases
- Let T be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i

Phase 1: Obtaining a Decision

- Coordinator asks all participants to *prepare* to commit transaction T_i.
  - C_i adds the records <**prepare** *T*> to the log and forces log to stable storage
  - sends **prepare** *T* messages to all sites at which *T* executed
- Upon receiving message, transaction manager at site determines if it can commit the transaction
  - if not, add a record <**no** *T*> to the log and send **abort** *T* message to C_i
  - if the transaction can be committed, then:
    - add the record <**ready** *T*> to the log
    - force *all records* for *T* to stable storage
    - send **ready** *T* message to C_i

    Transaction is now in ready state at the site

Phase 2: Recording the Decision

- *T* can be committed of C_i received a **ready** *T* message from all the participating sites: otherwise *T* must be aborted.
- Coordinator adds a decision record, <**commit** *T*> or <a**bort** *T*>, to the log and forces record onto stable storage. Once the record stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally.

Two-Phase Commit Protocol

Handling of Failures - Site Failure

When site S_k recovers, it examines its log to determine the fate of transactions active at the time of the failure.

- Log contain <**commit** *T*> record: site executes **redo** (*T*)
- Log contains <**abort** *T*> record: site executes **undo** (*T*)
- Log contains <**ready** *T*> record: site must consult C_i to determine the fate of *T*.
  - If *T* committed, **redo** (*T*)
  - If *T* aborted, **undo** (*T*)
- The log contains no control records concerning *T* implies that S_k failed before responding to the **prepare** *T* message from C_i
  - since the failure of S_k precludes the sending of such a response C_i must abort *T*
  - S_k must execute **undo** (*T*)

Handling of Failures- Coordinator Failure

- If coordinator fails while the commit protocol for *T* is executing then participating sites must decide on *T*' s fate:
  1. If an active site contains a <**commit** *T*> record in its log, then *T* must be committed.
  2. If an active site contains an <**abort** *T*> record in its log, then *T* must be aborted.
  3. If some active participating site does not contain a <**ready** *T*> record in its log, then the failed coordinator C_i cannot have decided to commit *T*. Can therefore abort *T*.
  4. If none of the above cases holds, then all active sites must have a <**ready** *T*> record in their logs, but no additional control records (such as <**abort** *T*> of <**commit** *T*>). In this case active sites must wait for C_i to recover, to find decision.
- **Blocking problem**: active sites may have to wait for failed coordinator to recover.

Handling of Failures - Network Partition

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol.
- If the coordinator and its participants belong to several partitions:
  - Sites that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.
    - No harm results, but sites may still have to wait for decision from coordinator.
- The coordinator and the sites are in the same partition as the coordinator think that the sites in the other partition have failed, and follow the usual commit protocol.
    - Again, no harm results

Recovery and Concurrency Control

- **In-doubt transactions** have a <**ready** *T*>, but neither a <**commit** *T*>, nor an <**abort** *T*> log record.
- The recovering site must determine the commit-abort status of such transactions by contacting other sites; this can slow and potentially block recovery.
- Recovery algorithms can note lock information in the log.
  - Instead of <**ready** *T*>, write out <**ready** *T*, *L*> *L* = list of locks held by *T* when the log is written (read locks can be omitted).
  - For every in-doubt transaction *T*, all the locks noted in the <**ready** *T*, *L*> log record are reacquired.
- After lock reacquisition, transaction processing can resume; the commit or rollback of in-doubt transactions is performed concurrently with the execution of new transactions.

Avoiding Blocking During Consensus

- Blocking problem of 2PC is a serious concern
- Idea: involve multiple nodes in decision process, so failure of a few nodes does not cause blocking as long as majority don't fail
- More general form: **distributed consensus problem**
  - A set of *n* nodes need to agree on a decision
  - Inputs to make the decision are provided to all the nodes, and then each node votes on the decision
  - The decision should be made in such a way that all nodes will "learn" the same value for the even if some nodes fail during the execution of the
    protocol, or there are network partitions.
  - Further, the distributed consensus protocol should not block, as long as a majority of the nodes participating remain alive and can communicate with each other
- Several consensus protocols, Paxos and Raft are popular
  - More later in this chapter

Using Consensus to Avoid Blocking

- After getting response from 2PC participants, coordinator can initiate distributed consensus protocol by sending its decision to a set of participants who then use consensus protocol to commit the decision
  - If coordinator fails before informing all consensus participants
    - Choose a new coordinator, which follows 2PC protocol for failed coordinator
    - If a commit/abort decision was made as long as a majority of consensus participants are accessible, decision can be found without blocking
  - If consensus process fails (e.g., split vote), restart the consensus
    - Split vote can happen if a coordinator send decision to some participants and then fails, and new coordinator send a different decision
- The **three phase commit** protocol is an extension of 3PC which avoids blocking under certain assumptions
  - Ideas are similar to distributed consensus.
- Consensus is also used to ensure consistency of replicas of a data item
  - Details later in the chapter

Distributed Transactions via Persistent Messaging

- Notion of a single transaction spanning multiple sites is inappropriate for many applications
  - E.g., transaction crossing an organizational boundary
  - Latency of waiting for commit from remote site
- Alternative models carry out transactions by sending messages
  - Code to handle messages must be carefully designed to ensure atomicity and durability properties for updates
    - Isolation cannot be guaranteed, in that intermediate stages are visible, but code must ensure no inconsistent states result due to concurrency
  - Persistent messaging systems are systems that provide transactional properties to messages
    - **Persistent messages** are guaranteed to be delivered exactly once

Persistent Messaging

- Example: funds transfer between two banks
  - Two phase commit would have the potential to block updates on the accounts involved in funds transfer
  - Alternative solution:
    - Debit money from source account and send a message to other site
    - Site receives message and credits destination account
  - Messaging has long been used for distributed transactions (even before computers were invented!)
- Atomicity issue
  - once transaction sending a message is committed, message must guaranteed to be delivered
    - Guarantee as long as destination site is up and reachable, code to handle undeliverable messages must also be available
      - e.g., credit money back to source account.
  - If sending transaction aborts, message must not be sent

Error Conditions with Persistent Messaging

- Code to handle messages has to take care of variety of failure situations (even assuming guaranteed message delivery)
  - E.g., if destination account does not exist, failure message must be sent back to source site
  - When failure message is received from destination site, or destination site itself does not exist, money must be deposited back in source account
    - Problem if source account has been closed
      - get humans to take care of problem
- User code executing transaction processing using 2PC does not have to deal with such failures
- There are many situations where extra effort of error handling is worth the benefit of absence of blocking
  - E.g., pretty much all transactions across organizations

Persistent Messaging Implementation

Persistent Messaging (Cont.)

- Receiving site may get duplicate messages after a very long delay
  - To avoid keeping processed messages indefinitely
    - Messages are given a timestamp
    - Received messages older than some cutoff are ignored
    - Stored messages older than the cutoff can be deleted at receiving site
- **Workflows** provide a general model of transactional processing involving multiple sites and possibly human processing of certain steps
  - E.g., when a bank receives a loan application, it may need to
    - Contact external credit-checking agencies
    - Get approvals of one or more managers

    and then respond to the loan application
  - Persistent messaging forms the underlying infrastructure for workflows in a distributed environment

Concurrency Control in Distributed Databases

Concurrency Control

- Modify concurrency control schemes for use in distributed environment.
- We assume that each site participates in the execution of a commit protocol to ensure global transaction atomicity.
- We assume all replicas of any item are updated
  - Will see how to relax this in case of site failures later

Single-Lock-Manager Approach

- In the **single lock-manager** approach, lock manager runs on a *single* chosen site, say S_i
  - All lock requests sent to central lock manager
- The transaction can read the data item from *any* one of the sites at which a replica of the data item resides.
- Writes must be performed on all replicas of a data item
- Advantages of scheme:
  - Simple implementation
  - Simple deadlock handling
- Disadvantages of scheme are:
  - Bottleneck: lock manager site becomes a bottleneck
  - Vulnerability: system is vulnerable to lock manager site failure.

Distributed Lock Manager

- In the **distributed lock-manager** approach, functionality of locking is implemented by lock managers at each site
  - Lock managers control access to local data items
  - Locking is performed separately on each site accessed by transaction
    - Every replica must be locked and updated
    - But special protocols may be used for replicas (more on this later)
- Advantage: work is distributed and can be made robust to failures
- Disadvantage:
  - Possibility of a global deadlock without local deadlock at any single site
  - Lock managers must cooperate for deadlock detection

Deadlock Handling

Consider the following two transactions and history, with item X and transaction T_1 at site 1, and item Y and transaction T_2 at site 2:

T_1: write (X) write (Y)	T_2: write (X) write (Y)
X-lock on X write (X)	
	X-lock on Y write (Y) wait for X-lock on X
Wait for X-lock on Y	

Result: deadlock which cannot be detected locally at either site

Deadlock Detection

- In the **centralized deadlock-detection** approach, a global wait-for graph is constructed and maintained in a *single* site; the deadlock-detection coordinator
  - *Real graph*: Real, but unknown, state of the system.
  - *Constructed graph*: Approximation generated by the controller during the execution of its algorithm .
- the global wait-for graph can be constructed when:
  - a new edge is inserted in or removed from one of the local wait-for graphs.
  - a number of changes have occurred in a local wait-for graph.
  - the coordinator needs to invoke cycle-detection.
- If the coordinator finds a cycle, it selects a victim and notifies all sites. The sites roll back the victim transaction.

Local and Global Wait-For Graphs

site S_1 site S_2

Local

Global

Example Wait-For Graph for False Cycles

Initial state:

False Cycles (Cont.)

- Suppose that starting from the state shown in figure,
  1. T_2 releases resources at S_1
     - resulting in a message remove $T_1 \rightarrow T_2$ message from the Transaction Manager at site S_1 to the coordinator)
  2. And then T_2 requests a resource held by T_3 at site S_2
     - resulting in a message insert $T_2 \rightarrow T_3$ from S_2 to the coordinator
- Suppose further that the insert message reaches before the **delete** message
  - this can happen due to network delays
- The coordinator would then find a false cycle

$$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$$

- The false cycle above never existed in reality.
- False cycles cannot occur if two-phase locking is used.

Distributed Deadlocks

- Unnecessary rollbacks may result
  - When deadlock has indeed occurred and a victim has been picked, and meanwhile one of the transactions was aborted for reasons unrelated to the deadlock.
  - Due to false cycles in the global wait-for graph; however, likelihood of false cycles is low.
- In the **distributed deadlock-detection** approach, sites exchange wait-for information and check for deadlocks
  - Expensive and not used in practice

Leases

- A **lease** is a lock that is granted for a specific period of time
- If a process needs a lock even after expiry of lease, process can **renew** the lease
- But if renewal is not done before end time of lease, the lease **expires,** and lock is released
- Leases can be used to that there is only one coordinator for a protocol at any given time
  - Coordinator gets a lease and renews it periodically before expire
  - If coordinator dies, lease will not be renewed and can be acquired by backup coordinator

Leases (Cont.)

- Coordinator must check that it still has lease when performing action
  - Due to delay between check and action, must check that expiry is at least some time *t'* into the future
    - *t'* includes delay in processing and maximum network delay
    - Old messages must be ignored
- Leases depend on clock synchronization

Distributed Timestamp-Based Protocols

- Timestamp based concurrency-control protocols can be used in distributed systems
- Each transaction must be given a *unique* timestamp
- Main problem: how to generate a timestamp in a distributed fashion
  - Each site generates a unique local timestamp using either a logical counter or the local clock.
  - Global unique timestamp is obtained by concatenating the unique local timestamp with the unique identifier.

Distributed Timestamps

- A node with a slow clock will assign smaller timestamps
  - Still logically correct: serializability not affected
  - But: “disadvantages” transactions
- To fix this problem
  - Keep clocks synchronized using network time protocol
  - Or, define within each node N_i a **logical clock** (LC_i), which generates the unique local timestamp
    - Require that N_i advance its logical clock whenever a request is received from a transaction Ti with timestamp < *x,y*> and x is greater that the current value of LC_i.
    - In this case, site N_i advances its logical clock to the value $x + 1$

Distributed Timestamp Ordering

- Centralized TSO and multiversion TSO easily extended to distributed setting
  - Transactions use a globally unique timestamp
  - Each site that performs a read or write performs same checks as in centralized case
- Clocks at sites should be synchronized
  - Otherwise a transaction initiated at a site with a slower clock may get restarted repeatedly.

Distributed Validation

- The validation protocol used in centralized systems can be extended to distributed systems
- Start/validation/finish timestamp for a transaction T_i may be issued by any of the participating nodes
  - Must ensure StartTS(T_i) < TS(T_i) < FinishTS(T_i)
- Validation for T_i is done at each node that performed read/write
  - Validation checks for transaction T_i are same as in centralized case
    - Ensure that no transaction that committed after T_i started has updated any data item read by T_i.
  - A key difference from centralized case is that may T_i start validation after a transaction with a higher validation timestamp has already finished validation
    - In that case T_i is rolled back

Distributed Validation (Cont.)

- Two-phase commit (2PC) needed to ensure atomic commit across sites
  - Transaction is validated, then enters prepared state
  - Writes can be performed (and transaction finishes) only after 2PC makes a commit decision
  - If transaction T_i is in prepared state, and another transaction T_k reads old value of data item written by T_i, T_k will fail if T_i commits
    - Can make the read by T_k wait, or create a commit dependency for T_k on T_i.

Distributed Validation (Cont.)

- Distributed validation is not widely used, but optimistic concurrency control without read-validation is widely used in distributed settings
  - Version numbers are stored with data items
  - Writes performed at commit time ensure that the version number of a data item is same as when data item was read
  - Hbase supports atomic checkAndPut() as well as checkAndMutate() operations; see book for details

Replication

Replication

- **High availability** is a key goal in a distributed database
  - **Robustness**: the ability to continue function despite failures
- Replication is key to robustness
- Replication decisions can be made at level of data items, or at the level of partitions

Consistency of Replicas

- Consistency of replicas
  - Ideally: all replicas should have the same value ➔ updates performed at all replicas
    - But what if a replica is not available (disconnected, or failed)?
  - Suffices if reads get correct value, even if some replica is out of date
  - Above idea formalized by **linearizability**: given a set of read and write operations on a (replicated) data item
    - There must be a linear ordering of operations such that each read sees the value written by the most recent preceding write
    - If o_1 finishes before o_2 begins (based on external time), then o_1 must precede o_2 in the linear order
- Note that linearizability only addresses a single (replicated) data item; serializability is orthogonal

Consistency of Replicas

- Cannot differentiate node failure from network partition in general
  - Backup coordinator should takeover if primary has failed
  - Use multiple independent links, so single link failure does not result in partition, but it is possible all links have failed
- Protocols that require all copies to be updated are not robust to failure
- Will see techniques that can allow continued processing during failures, whether node failure or network partition
  - Key idea: decisions made based on successfully writing/reading majority
- Alternative: **asynchronous replication:** commit after performing update on a *primary copy* of the data item, and update replicas *asynchronously*
  - Lower overheads, but risk of reading stale data, or lost updates on primary failure

Concurrency Control With Replicas

- Focus here on concurrency control with locking
  - Failures addressed later
  - Ideas described here can be extended to other protocols
- **Primary copy**
  - one replica is chosen as primary copy for each data item
    - Node containing primary replica is called **primary node**
  - concurrency control decisions made at the primary copy only
- Benefit: Low overhead
- Drawback: primary copy failure results in loss of lock information and non-availability of data item, even if other replicas are available
  - Extensions to allow backup server to take over possible, but vulnerable to problems on network partition

Concurrency Control With Replicas (Cont.)

- **Majority protocol**:
  - Transaction requests locks at multiple/all replicas
  - Lock is successfully acquired on the data item only if lock obtained at a majority of replicas
- Benefit: resilient to node failures and node failures
  - Processing can continue as long as at least a majority of replicas are accessible
- Overheads
  - Higher cost due to multiple messages
  - Possibility of deadlock even when locking single item
    - How can you avoid such deadlocks?

Concurrency Control With Replicas (Cont.)

- **Biased protocol**
  - Shared lock can be obtained on any replica
    - Reduces overhead on reads
  - Exclusive lock must be obtained on *all* replicas
    - Blocking if any replica is unavailable

©Silberschatz, Korth and Sudarshan

Quorum Consensus Protocol

Quorum consensus protocol for locking

- Each site is assigned a weight; let S be the total of all site weights
- Choose two values **read quorum** Q_R and **write quorum** Q_W
  - Such that $Q_r + Q_w > S$ and $2 * Q_w > S$
- Each read must lock enough replicas that the sum of the site weights is $\geq Q_r$
- Each write must lock enough replicas that the sum of the site weights is $\geq Q_w$
- Can choose Q_r and Q_w to tune relative overheads on reads and writes
  - Suitable choices result in majority and biased protocols.
    - What are they?

Dealing with Failures

- Read one write all copies protocol assumes all copies are available
  - Will block if any site is not available
- *Read one write all available* (ignoring failed sites) is attractive, but incorrect
  - Failed link may come back up, without a disconnected site ever being aware that it was disconnected
  - The site then has old values, and a read from that site would return an incorrect value
  - With network partitioning, sites in each partition may update same item concurrently
    - believing sites in other partitions have all failed

Handling Failures with Majority Protocol

- The majority protocol with version numbers
  - Each replica of each item has a **version number**
  - Locking is done using majority protocol, as before, and version numbers are returned along with lock allocation
  - Read operations read the value from the replica with largest version number
  - Write operations
    - Find highest version number like reads, and set new version number to old highest version + 1
    - Writes are then performed on all locked replicas and version number on these replicas is set to new version number
- Read operations that find out-of-date replicas may optionally write the latest value and version number to replicas with lower version numbers
  - no need to obtain locks on all replicas for this task

Handling Failures with Majority Protocol

- Atomic commit of updated replicas must be ensured using either
  - *2 phase commit on all locked replicas*, or
  - distributed consensus protocol such as Paxos (more on this later)
- Failure of nodes during 2PC can be ignored as long as majority of sites enter prepared state
- Failure of coordinator can cause blocking
  - Consensus protocols can avoid blocking

Handling Failures with Majority Protocol

- Benefits of majority protocol
  - Failures (network and site) do not affect consistency
    - Reads are guaranteed to see latest successfully written version of a data item
  - Protocol can proceed as long as
    - Sites available at commit time contain a majority of replicas of any updated data items
    - During reads a majority of replicas are available to find version numbers
  - No need for any special reintegration protocol: nothing needs to be done if nodes fail and subsequently recover
- Drawback of majority protocol
  - Higher overhead, especially for reads

Reducing Read Cost

- Quorum consensus can be used to reduce read cost
  - But at increased risk of blocking of writes due to failures
- Use primary copy scheme:
  - perform all updates at primary copy
  - reads only need to be done at primary copy
  - But what if primary copy fails
    - Need to ensure new primary copy is chosen
      - Leases can ensure there is only 1 primary copy at a time
    - New primary copy needs to have latest committed version of data item
      - Can use consensus protocol to avoid blocking

Reducing Read Cost

- **Chain replication**:
  - Variant of primary copy scheme
  - Replicas are organized into a chain
  - Writes are done at head of chain, and passed on to subsequent replicas
  - Reads performed at tail
    - Ensures that read will get only fully replicated copy
  - Any node failure requires reconfiguration of chain

Reconfiguration and Reintegration

- To be robust, a distributed system must either
  - Follow protocols like the majority protocol that work in spite of failures or
  - Use other protocols like primary copy protocol, but
    - Detect failures (failed/non-reachable nodes)
    - Reconfigure the system to remove failed nodes, and assign their tasks to other sites, so computation may continue
    - Recover/reintegrate nodes a node or link is repaired
- Failure detection: distinguishing link failure from site failure is hard
  - (partial) solution: have multiple links, multiple link failure is likely a site failure

Reconfiguration

- Reconfiguration:
  - Abort all transactions that were active at a failed site
  - If replicated data items were at failed site, update system catalog to remove them from the list of replicas.
    - This should be reversed when failed site recovers, but additional care needs to be taken to bring values up to date
  - If a failed site was a central server for some subsystem, an **election** must be held to determine the new server
    - E.g., name server, concurrency coordinator, global deadlock detector

Reconfiguration (Cont.)

- Since network partition may not be distinguishable from site failure, the following situations must be avoided
  - Two or more central servers elected in distinct partitions
  - More than one partition updates a replicated data item
- Updates must be able to continue even if some sites are down
- Solution: majority based approach

Site Reintegration

- When failed site recovers, it must catch up with all updates that it missed while it was down
  - Problem: updates may be happening to items whose replica is stored at the site while the site is recovering
  - Solution 1: halt all updates on system while reintegrating a site
    - Unacceptable disruption
  - Solution 2: lock all replicas of all data items at the site, update to latest version, then release locks
    - Can do this for one partition at a time

Comparison with Remote Backup

- Remote backup systems (Section 19.7) are also designed to provide high availability
- Remote backup systems are simpler and have lower overhead
  - All actions performed at a single site, and only log records shipped
  - No need for distributed concurrency control, or 2 phase commit
- Using distributed databases with replicas of data items can provide higher availability by having multiple (> 2) replicas and using the majority protocol
  - Also avoid failure detection and switchover time associated with remote backup systems

Extended Concurrency Control Protocols

Multiversion 2PL and Globally Consistent Timestamps

- Recall multiversion 2PL protocol:
  - Read only transactions get timestamp at start
    - T_i reads latest committed version of data items with TS < startTS(T_i)
  - Update transactions perform 2PL, and also get timestamp at commit
  - Serialization order defined by timestamp
- Question: can we use MV2PL in a distributed system
- Answer: yes, *but a lot of conditions apply*
  - If commits are serialized at central coordinator, timestamps can be given based on counter
  - But if commits are distributed, how to give timestamps in a consistent manner?
    - Clocks may not be in sync, later commit may get lower timestamp
    - Out of order timestamp issual may result in serialization order not matching timestamp order

Multiversion 2PL and Globally Consistent Timestamps

- Centralized coordinator to assign consistent timestamps
  - Can be done, but becomes bottleneck
- Google Spanner ideas:
  - In an ideal world, clocks are synchronized, and can be used to assign commit timestamps to transactions
  - In reality, clocks are out of sync
  - Key ideas
    - Use atomic clocks, GPS etc to periodically get precise time
    - Derive bound on how out-of-sync a node's clock *t'* can be w.r.t. to actual time *t*
      - $t' - \varepsilon \leq t \leq t' + \varepsilon$
    - Introduce **commit wait**: hold locks for some period and assign timestamp *ts* such that locks were definitely held at actual time *ts*

Multiversion 2PL and Globally Consistent Timestamps

- Google Spanner ideas (cont):
  - If version of *x* has timestamp ts, then *x* definitely had that value at time *ts*
  - System can generate transactionally consistent snapshop as of actual time *ts* (**external consistency**)
  - Commit processing can still take time
    - With 2PC status of transaction may not be known for a while
      - Reads may have to wait till status of transaction is known
    - But read-only transactions can use a snapshot timestamp *ts* such that all transactions before that timestamp have been committed or aborted
      - Read can proceed without waiting
      - But perhaps with older versions of data

Other Concurrency Control Techniques

- **Distributed snapshot isolation**
  - Running Snapshot Isolation separately on each node may result in different serialization orders at different nodes
  - Extensions to SI to ensure consistent ordering have been proposed
- **Concurrency control in federated databases**
  - Local transactions
  - Global transactions
  - Local serializability may not guarantee global serializability unless all nodes use 2PL
  - Use idea of **tickets** to create conflicts that will ensure serializability

Replication With Weak Degrees of Consistency

Consistency

- Recall: Consistency in Databases (ACID):
  - Database has a set of integrity constraints
  - A consistent database state is one where all integrity constraints are satisfied
  - Each transaction run individually on a consistent database state must leave the database in a consistent state
- Recall: Consistency in distributed systems with replication
  - **Strong consistency**: a schedule with read and write operations on an object should give results and final state equivalent to some schedule on a single copy of the object, with order of operations from a single site preserved
  - Weak consistency (several forms)

Availability

- Traditionally, availability of centralized server
- For distributed systems, availability of system to process requests
  - For large system, at almost any point in time there's a good chance that
    - a node is down or even
    - Network partitioning
- Availability: ability to continue operations despite node and network failures.

CAP "Theorem"

- Three properties of a system
  - Consistency
    - an execution of a set of operations (reads and writes) on replicated data is said to be **consistent** if its result is the same as if the operations were executed on a single node, in a sequential order that is consistent with the ordering of operations issued by each process (transaction)
  - Availability (system can run even if parts have failed)
    - Via replication
  - Partitions (network can break into two or more parts, each with active systems that can't talk to other parts)
- Brewer's CAP "Theorem": You can have at most two of these three properties for any system

CAP "Theorem" (Cont.)

- Very large systems will partition at some point
- Choose one of consistency or availability
  - Traditional database choose consistency
  - Many web applications choose availability
    - Except for specific parts such as order processing
- Latency is another factor
  - Many applications choose to serve potentially stale data to reduce latency

Replication with Weak Consistency

- Many systems support replication of data with weak degrees of consistency (I.e., without a guarantee of serializabiliy)
  - In quorum consistency notation: allow Q_R and Q_W to be set such that $Q_R + Q_W \leq S$ or $2*Q_W \leq S$
    - E.g., can be set in MongoDB and Cassandra
  - Usually only when not enough sites are available to ensure quorum
    - But sometimes to allow fast local reads
  - Tradeoff of consistency versus availability or latency
- Key issues:
  - Reads may get old versions
  - Some replicas may not get updated
  - **Different updates may be applied to different replicas**
    - Question: how to detect, and how to resolve
    - Will see in detail later

Eventual Consistency

- When no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent
- For a given accepted update and a given node, eventually either the update reaches the node or the node is removed from service
- Known as **BASE** (**B**asically **A**vailable, **S**oft state, **E**ventual consistency), as opposed to ACID
  - **Soft state**: copies of a data item may be inconsistent
  - **Eventually Consistent :** Copies may be allowed to become inconsistent, but (once partitioning is resolved) eventually all copies become consistent with each other
    - at some later time, if there are no more updates to that data item

Asynchronous Replication

- With **asynchronous replication,** updates are done at the primary node (also known as master node), and then propagated to replicas
  - Transaction can commit once update is done at primary node
  - Propagation after commit is also referred to as **lazy propagation**
  - Allows updates to occur even if some sites are disconnected from the network, but at the cost of consistency
- Replicas may not be up-to-date
  - Transactions that can live with old data can read from replicas
  - Snapshot reads at a point in time can also be served from replicas that are sufficiently up-to-date
    - E.g., in Google Spanner
      - each replica maintains a timestamp t_{safe} such that all updates with timestamp $t < t_{safe}$ have already been received
      - Reads of a transaction can be satisfied by a replicate if transaction timestamp $t < t_{safe}$

Asynchronous Replication

- **Master-slave replication**: updates performed only at master, and asynchronously propagated to replicas
  - replicas can only satisfy reads
- **Multimaster replication** (or **update-anywhere replication**): updates can be performed at any replica, and propagated synchronously or asynchronously to other replicas
- Updates must be propagated to replicas even if there are failures, and processed in the correct order at the replicas
  - Persistent messaging systems can be used for this, with minor extensions to ensure in-order delivery
  - Publish-subscribe systems such as Kafka can also be used for this task
    - More flexible, support parallelism by having multiple topics and partitions of topics
  - Fault-tolerance is important
    - Can use log-replication with two-safe protocol (Section 19.7)

Asynchronous View Maintenance

- Materialized views can be useful in distributed systems
  - Secondary indices can be considered as a simple form of materialized view in a parallel database
    - E.g., given relation *r(A,B,C)* where *A* is the primary key on which *r* is partitioned, a secondary index on *B* is simply a projection of *r* on *(B,A),* partitioned on *B*.
  - Materialized aggregate views are also very useful in many contexts
- Performing view maintenance as part of the original transaction may not be possible (if the underlying database does not support distributed transactions), or may be expensive
- Asynchronous maintenance of materialized views, after the original transaction commits, is a good option in such a case
  - Applications using the view/index must then be aware that it may be a little out-of-date

Requirements for Asynchronous View Maintenance

Requirements:

1. Updates must be delivered and processed exactly once despite failures
2. Derived data (such as materialized views/indices) must be updated in such a way that it will be consistent with the underlying data
   - Formalized as **eventual consistency**: if there are no updates for a while, eventually the derived data will be consistent with the underlying data
3. Queries should get a transactionally consistent view of derived data
   - Potentially a problem with long queries that span multiple nodes
   - E.g., without transactional consistency, a scan of relation may miss some older updates and see some later updates
   - Not supported by many systems, supported via snapshots in some systems

©Silberschatz, Korth and Sudarshan

Detecting Inconsistency

- Data items are versioned
- Each update creates a new immutable version
- In absence of failure, there is a single latest version
- But with failures and weak consistency, versions can diverge
  - Different nodes may perform different updates on same data
  - Need to detect, and fix such situations
- Key idea: **vector-vector** identifies each data version
  - Set of (node, counter) pairs
    - E.g., with two nodes N1 and N2, ([N1,2],[N2,1])
    - Represented as a vector [2, 1]
  - An update to a data item at a node increments the counter for that node
  - Define a partial order across versions

Vector Vectors

- Examples of vector vectors
  - ([Sx,1]): data item created by site Sx
  - ([Sx,2]): data item created by site Sx, and later updated
  - ([Sx,2],[Sy,1]): data item updated twice by site Sx and once by Sy
    - Update by a site Sx increments the counter for Sx, but leaves counters from other sites unchanged
  - ([Sx,4],[Sy,1]) **newer than** ([Sx,3],[Sy,1])
  - But ([Sx,2],[Sy,1]) **incomparable with** ([Sx,1],[Sy,2])
  - Read operation may find **incomparable versions**
    - Such versions indicate inconsistent concurrent updates
    - All such versions returned by read operation
    - Up to application to reconcile multiple versions

Example of Vector Clock in action

- Item D1 created by Node N1
- D1 updated by Node N1
- D1 concurrently updated by node N2 and N3 (usually due to network partitioning)
- Subsequent read from N2 and N3 returns two incomparable versions
- Application merges versions and writes new version

Extensions for Detecting Inconsistency

- Two replicas may diverge, and divergence is not detected until the replicas is read
  - To detect divergence early, one approach is to scan all replicas of all items periodically
    - But requires a lot of network, CPU and I/O load
    - Alternative approach based on Merkle trees covered shortly

How to Reconcile Inconsistent Versions?

- Reconciliation is application specific
  - E.g., two sites concurrent insert items to cart
    - Merge adds both items to the final cart state
  - E.g., S1 adds item A, S2 deletes item B
    - Merge adds item A, but deleted item B resurfaces
    - Cannot distinguish S2 deletes B from S1 add B
    - Problem: operations are inferred from states of divergent versions
  - Better alternative:
    - Keep track of history of operations
    - Merge operation histories

Order Independent Operations

- Basic idea: updates are performed as logical operations
  - Data store is aware of the set of operations that can be carried out
  - Operation performed at place where data (replica) is stored
- If result of a sequence of operations is independent of the operation ordering
  - Independent update operations can be merged in different orders at different replicas, but will lead to same result
  - Eventual consistency can be ensured relatively easily

Detecting Differences Using Merkle Trees

- **Merkle Tree**: A data structure that can
  - Efficiently sign contents of a tree
  - Efficiently find differences (if any) between two replicas
- Example

Hash values of data items
$h_1(i_1)=00$
$h_1(i_2)=01$
$h_1(i_3)=11$
$h_1(i_4)=00$
$h_1(i_5)=10$
$h_1(i_6)=11$
$h_1(i_7)=10$

Node identifier shown above node, and has value shown inside node, v_i denotes stored hash value in node i

Detecting Differences Using Merkle Trees (Cont.)

- Overall cost of finding differences with Merkle tree
  - $O(m \log_2 N)$ with N data items and m differences using binary tree
  - Each operation requires communication between the two trees (nodes)
  - Use wider trees to reduce height/cost
    - Cost is $O(m \log_K N)$ if each node has K children instead of 2 children
    - Particularly important due to high network latency
- Merkle trees originally used for **verification of contents** of a collection
  - Include digital signature at root in this case.

Weak Consistency Models for Applications

- **Read-your-writes**
  - if a process has performed a write, a subsequent read will reflect the earlier write operation
- **Session consistency**
  - Read-your-writes in the context of a session, where application connects to storage system
- **Monotonic consistency**
  - For reads: later reads never return older version than earlier reads
  - For writes: serializes writes by a single process
    - Minimum requirement
- **Sticky sessions**: all operations from a session on a data item go to the same node
- Can be implemented by specifying a version vector in get() operations
  - Result of get guaranteed to be at least as new as specified version vector

Coordinator Selection

Coordinator Selection

- **Backup coordinators**
  - Backup coordinator maintains enough information locally to assume the role of coordinator if the actual coordinator fails
    - executes the same algorithms and maintains the same internal state information as the actual coordinator
  - allows fast recovery from coordinator failure but involves overhead during normal processing.
- Backup coordinator approach vulnerable to two-site failure
  - Failure of coordinator and backup leads to non-availability
  - Key question: how to choose a new coordinator from a set of candidates
    - Choice done by a master: vulnerable to master failure
    - Election algorithms are key

Coordinator Selection

- **Coordinator selection using a fault-tolerant lock manager**
  - Coordinator gets a lease on a coordinator lock, and renews the lease as long as it is alive
  - If coordinator dies or gets disconnected, lease is losk
  - Other nodes can detect coordinator failure using heart-beat messages
  - Nodes request coordinator lock lease from lock manager; only 1 node gets the lease, and becomes new coordinator
- Fault-tolerant coordination services such as **ZooKeeper, Chubby**
  - Provide fault-tolerant lock management services
  - And are widely used for coordinator section
  - Store (small amounts) of data in files
  - Create and delete files
    - Which can be used as locks/leases
      - Coordinator releases lease if it is not renewed in time
  - Can watch for changes on a file
  - But these services themselves need a coordinator……..

Election of Coordinator

- **Election algorithms**
  - Used to elect a new coordinator in case of failures
    - Heartbeat messages used to detect failure of coordinator
  - One-time election protocol
    - **Proposers**: Nodes that propose themselves as coordinator and send vote requests to other nodes
    - **Acceptors**: Nodes that can vote for candidate proposers
    - **Learners**: Nodes that ask acceptors who they voted for, to find winner
      - A node can perform all above roles
    - Problems with this protocol
      - What if no one won the election due to split vote?
      - If election is rerun, need to identify which election a request is for
  - General approach
    - Candidates make a proposal with a **term number**
      - Term number is 1 more than term number of previous election known to candidate

Election of Coordinator

- **Election algorithms (Cont.)**
    - **Stale messages** corresponding to old terms can be ignored
    - If a candidate wins majority vote it becomes coordinator
    - Otherwise election is rerun with term number incremented
    - Minimizing chances of split elections:
        - Use node IDs to decide who to vote for
            - e.g., max node ID (**Bully algorithm**)
            - Candidates withdraw if they find another candidate with higher ID
        - **Randomized retry**: candidates wait for random time intervals before retrying
            - High probability that only one node is asking to be elected at a time
- Special case of distributed consensus

Issues with Multiple Coordinators

- Coordinator may get disconnected, and new coordinator elected, without old coordinator ever knowing about the election
  - Multiple nodes may thus believe they are coordinators
    - Called a **split-brain** situation
- Solutions:
  - **Term numbers** can be used to identify coordinator
    - Majority of node will know of latest coordinator term since they voted for it
    - Messages with old term number (**stale messages**) can be ignored
  - **Leases** can be used to ensure only one coordinator at a time
    - Delayed messages may still be received from old coordinator
      - Term numbers can be used to ignore such delayed messages

Distributed Consensus

Distributed Consensus

- Motivating example: commit decision in two-phase commit (2PC)
  - Decision made by coordinator alone: vulnerable to blocking problem
    - If coordinator fails/gets disconnected at certain key points, rest of system does not know if the decision was to commit/abort, and must block till coordinator recovers
  - Multiple nodes must participate in decision process to ensure fault tolerance
    - Although initial proposal for decision may be made by a single node
  - Goal: A decision making protocol that is non-blocking as long as a majority of participating nodes are up and reachable
- 2PC is a special case of a more general class of decision problems that must be made by a collection of nodes in a fault-tolerant, non-blocking manner

Distributed Consensus

- **Distributed consensus problem**: A set of *n* nodes (called **participants**) need to agree on a decision by executing a protocol such that
  - All participants "learn" the same value for the decision
    - even if some nodes fail during the execution of the protocol, messages are lost, or there are network partitions
  - The protocol should not block, and must terminate, as long as some majority of nodes are alive and can communicate with each other

Distributed Consensus: Overview

- An real system needs to make a series of decisions: **multiple consensus protocol**
- Problem can be abstracted as adding a record to a log
  - Each node has a copy of the log, and log records are appended at each node
  - Potential for conflicts between the nodes on what record is appended at what point in the log
  - The multiple consensus protocol must ensure that the log is uniquely defined
    - Copies of the log may temporarily differ, but must be made consistent subsequently
      - May require deleting parts of the log on a node
    - Actions can be taken on a log record only after consensus has been reached for that position in the log

Distributed Consensus: Overview (Cont.)

- Several protocols proposed
  - We outline key ideas behind **Paxos** and **Raft**
  - The **Zab** protocol used in ZooKeeper is another widely used consensus protocol
- Key idea: voting to make a decision
  - A particular decision succeeds only if a majority of the participating nodes have voted for it
    - Prevents more than one decision being chosen in a round
    - If majority of nodes are up and agree on a decision voting will not block
      - But devil in the details!

Paxos Consensus Protocol

- Assume a collection of processes that can **propose** values
  - Different processes may propose different values
  - Proposals are sent to **acceptors** which collectively choose from among the proposals
- A single execution of a **distributed consensus protocol** must ensure that:
  - At most a single value from amongst those proposed is chosen collectively by the acceptors
  - If a value has been chosen, then **learner** processes should be able to learn the chosen value
    - In case no value is chosen (split-voting), protocol reexecutes
  - Protocol should not block, and must terminate, as long as some majority of the nodes participating remain alive and can communicate with each other

Paxos Consensus Protocol: Overview

- Key idea: Consensus is reached when a *majority* of acceptors have accepted a particular proposal
  - Learner finds what value (if any) was accepted by a majority of acceptors
- If a majority vote for a particular value, all is fine, BUT
  - Vote may get split, requiring further rounds to reach a majority
  - Worse, even if a majority accept a value (and even if a learner learns of the majority), some of the acceptors (and the learner) may die or get disconnected
    - Remaining nodes may not be a majority
    - If this is treated as failure and another round is run, a different proposal may get accepted, with different learners learning different values!
  - Once acceptor has voted for a particular proposal in a round, it cannot change its mind for that round
    - Decision must be logged to ensure no change in decision if acceptor dies and comes back up

Paxos: Overview

- To deal with split vote Paxos uses a coordinator
  - Proposals serialized through coordinator, so only one value is typically proposed in a round
  - Paxos works correctly (but less efficiently) even if there are multiple coordinators
  - Coordinator can be elected
- Different values getting majorities in different nodes is a more serious problem. To solve it further rounds should give same result.
- Key idea:
  - Each proposal in Paxos has a unique number
  - Acceptors accept highest numbered proposal received in a round
  - Proposers will not create new proposals with a different number
  - Two phase protocol

Paxos Made Simple

- Phase 1
  - **Phase 1a**: A proposer selects a proposal number *n* and sends a prepare request with number *n* to a majority of acceptors
    - Number has to be chosen in some unique way
  - **Phase 1b**: If an acceptor receives a prepare request with number *n*
    - If *n* is less than that of any prepare request to which it has already responded then it ignores the request
    - Else it remembers *n* and responds to the request
      - If it has already accepted a proposal with number *m* and value *v,* it sends (*m, v*) with the response
      - Otherwise it indicates to the proposer that it has not accepted any value earlier
      - *NOTE: responding is NOT the same as accepting*

Paxos Made Simple

- Phase 2
  - **Phase 2a:** *Proposer Algorithm*: If the proposer receives a response to its prepare requests (numbered *n*) from a majority of acceptors
    - then it sends an *accept* request to each of those acceptors for a proposal numbered *n* with a value *v*, where v is
      - the value selected by the proposer if none of the acceptors indicated it had already accepted a value.
      - Otherwise *v* is the value of the highest-numbered proposal among the responses
        - i.e., proposer backsoff from its own proposal and votes for highest numbered proposal already accepted by at least one acceptor
    - If proposer does not hear from a majority it takes no further action in this round

Paxos Made Simple

- Phase 2
  - **Phase 2b:** *Acceptor Algorithm*: If an acceptor receives an accept request for a proposal numbered *n*,
    - If it has earlier responded to a prepare message with number *n1* > *n* it ignores the message
    - Otherwise it *accepts* the proposed value *v* with number *n*.
      - Note: acceptor may accept different values with increasing *n*

Paxos Details

- Key idea: if a majority of acceptors accept a value *v* (with number *n*), then even if there are further proposals with number *n1* > *n*, the value proposed will be value *v*
  - *Why?:*
    - A value can be accepted with number *n* only if a majority of nodes (say *P*) respond to a prepare message with number *n*
    - Any subsequent majority (say *A*) will have nodes in common with the first majority *P*, and at least one of those nodes would have responded with value *v* and number *n*
      - If a higher numbered proposal *p* was accepted earlier by even one node majority would have responded to *p*, and will ignore *n*
    - Further rounds will use this value *v* (since highest accepted value is used in Phase 2a)

Paxos Details (Cont.)

- At end of phase 2, it is possible that there is no majority have agreed on a value
  - Learners that believe majority was not reached can initiate a fresh proposal
  - If majority had actually been reached, same value will be chosen again
- Many more details under cover
- Above is for a single decision. **Multi-Paxos**: extension which deals with a series of decisions
- Many variants of Paxos optimized for different scenarios

The Raft Consensus Protocol

The Log-Based Consensus Protocols

- Fault-tolerant log, to which records are appended
- Each participating node maintains a replica of a log
- Key goal: keep the log replicas in sync
  - Logical view of atomically appending records to all copies of the log
  - Can’t actually be done atomically; logs may diverge
- Consensus protocols must ensure
  - Even if a log replica is temporarily inconsistent with another, it will be brought back to sync
    - May require log deletion and replacement
  - A log entry will not be treated as committed until the algorithm guarantees that it will never be deleted

The Raft Consensus Algorithm

- Raft is based on having a coordinator, called a **leader**
  - Essential in Raft, unlike Paxos, where coordinator is an optimization
- Other nodes are called **followers**
- Leaders may die and get replaced
  - Time divided into **terms,** each term has a unique leader
  - Terms have increasing numbers

©Silberschatz, Korth and Sudarshan

The Raft Leader Election

- Leaders are elected using randomized retry algorithm outlined in Section 23.7.2
  - Recall that algorithm already uses notion of term
  - Voting is done for a specific term
    - Can change in another term
  - Nodes track *currentTerm* based on messages received
- Leader *N1* may get disconnected and get reconnected after new leader *N2* is elected
  - *N1* may not even know it was disconnected and may continue leader actions

Example of Raft Logs

- Number in each entry indicates term
- Example log entries are assignments to variables

log index	1	2	3	4	5	6	7
leader	1 $x \leftarrow 2$	1 $z \leftarrow 2$	1 $x \leftarrow 3$	2 $x \leftarrow 4$	3 $x \leftarrow 1$	3 $y \leftarrow 6$	3 $z \leftarrow 4$
follower 1	1 $x \leftarrow 2$	1 $z \leftarrow 2$	1 $x \leftarrow 3$	2 $x \leftarrow 4$	3 $x \leftarrow 1$		
follower 2	1 $x \leftarrow 2$	1 $z \leftarrow 2$	1 $x \leftarrow 3$	2 $x \leftarrow 4$	3 $x \leftarrow 1$	3 $y \leftarrow 6$	3 $z \leftarrow 4$
follower 3	1 $x \leftarrow 2$	1 $z \leftarrow 2$	1 $x \leftarrow 3$				
follower 4	1 $x \leftarrow 2$	1 $z \leftarrow 2$	1 $x \leftarrow 3$	2 $x \leftarrow 4$	3 $x \leftarrow 1$	3 $y \leftarrow 6$	

committed entries

Raft Log Replication

- Appending a log entry done by sending log append request to leader
- Leader sends *AppendEntries* request to all followers, with these parameters
  - *term*
  - *previousLogEntryPosition*
  - *previousLogEntryTerm*
  - *logEntries*: array allowing multiple log records to be appended
  - *leaderCommitIndex*: an index such that all log records before the index are committed
- Followers carry out checks and respond (next slide)
- If majority of nodes respond with true, leader can report successful log append to initiating node
  - Otherwise more work is needed, explained later

Raft AppendEntries Procedure

- Follower that receives *AppendEntries* message does the following
  1. If term in message is less than followers *currentTerm*, Return false
  2. If log does not have an entry at *previousLogEntryPosition* with term matching *previousLogEntryTerm*, Return false
  3. If entry at *previousLogEntryPosition* is different from first log record in *AppendEntries* message, delete existing entry and all subsequent entries in log
  4. Any log records in *logEntries* that are not already in log are appended to log
  5. Follower maintains local *commitIndex*
     - if *leaderCommitIndex* > *commitIndex*, set *commitIndex*=min(*leaderCommitIndex*, last log entry index)
  6. Return true

©Silberschatz, Korth and Sudarshan

Raft AppendEntries Procedure (Cont.)

- If leader *N1* receives a false message from follower with a higher currentTerm, *N1* realizes it is no longer a leader and becomes a follower
- Different followers may have different log states
- If leader receives false from a node, log in that node is out of date and needs updating
  - Leader retries AppendEntries for that node, starting from an earlier point in its own log
  - May get false several times, until it goes far enough back in log to find a matching log entry
- Key remaining issue: if a leader dies, and another one takes over, the log must be brought to consistent state
  - New leader may have an older log

Raft Leader Replacement

- Raft protocol ensures any node elected as leader has all committed log entries
  - Candidate must send information about its own log state when seeking votes
  - Node votes for candidate only if candidates log state is at least as up-to-date as its own (we omit details)
  - Since majority have voted for new leader, any committed log entry will be in new leaders log
- Raft forces all other nodes to replicate leaders log
  - Log records at new leader may get committed when log gets replicated
  - Leader *cannot* count number of replicas with a record from an earlier term and declare it committed if it is at majority
    - Details are subtle, and omitted
    - Instead, leader must replicate a new log record in its current term

Raft Protocol

- There are many more subtle details that need to be taken care of
  - Consistency even in face of multiple failures and restarts
  - Maintaining cluster membership, cluster membership changes
- Raft has been proven formally correct
- See bibliographic notes for more details of above

Fault-Tolerant Services using Replicated State Machines

- Key requirement: make a service fault tolerant
  - E.g., lock manager, key-value storage system, ….
- State machines are a powerful approach to creating such services
- A **state machine**
  - Has a stored state, and receives inputs
  - Makes state transitions on each input, and may output some results
    - Transitions and output must be deterministic
- A **replicated state machine** is a state machine that is replicated on multiple nodes
  - All replicas must get exactly the same inputs
    - Replicated log! State machine processes only committed inputs!
  - Even if some of the nodes fail, state and output can be obtained from other nodes

Replicated State Machine

- Replicated state machine based on replicated log
- Example commands assign values to variables

Leader declares log record committed after it is replicated at a majority of nodes. Update of state machine at each replica happens only after log record has been committed.

Uses of Replicated State Machines

- Replicated state machines can be used to implement wide variety of services
  - Inputs can specify operations with parameters
  - But operations must be deterministic
  - Result of operation can be sent from any replica
    - Gets executed only when log record is committed in replicated log
    - Usually sent from leader, which knows which part of log is committed
- Example: **Fault-tolerant lock manager**
  - State: lock table
  - Operations: lock requests and lock releases
  - Output: grant, or rollback requests on deadlock
  - Centralized implementation is made fault tolerant by simply running it on a replicated state machine

Uses of Replicated State Machines

- **Fault tolerant key-value store**
  - State: key-value storage state
  - Operations: get() and put() are first logged
    - Operations executed when the log record is in committed state
    - Note: even get() operations need to be processed via log
- Google Spanner uses replicated state machine to implement key-value store
  - Data is partitioned, and each partition is replicated across multiple nodes
  - Replicas of a partition form a *Paxos group* with one node as leader
  - Operations initiated at leader, and replicated to other nodes

Two-Phase Commit Using Consensus

- Basic two-phase commit can result in blocking
- **Non-blocking two-phase commit** can be implemented using consensus
  - Key idea: Record commit decisions using consensus protocol instead of logging it at coordinator
  - As long as majority of sites are up and reachable, decision will be known
    - Blocking is then avoided
- Used e.g. in Google spanner, for transactions that span partitions
  - 2PC is coordinated by Paxos group leader at any 1 partition
  - Lock table is implemented using replicated state machine
    - Even if leader fails, new leader can see up-to-date lock state

End of Chapter 23

Extra Slides – Material Not in Text

- Weak Consistency
- Miscellaneous

Dynamo: Basics

- Provides a key-value store with basic get/put interface
  - Data values entirely uninterpreted by system
    - Unlike Bigtable, PNUTS, Megastore, etc.
- Underlying storage based on DHTs using consistent hashing with virtual processors
- Replication (N-ary)
  - Data stored in node to which key is mapped, as well as N-1 consecutive successors in ring
  - Replication at level of key range (virtual node)
  - Put call may return before data has been stored on all replicas
    - Reduces latency, at risk of consistency
    - Programmer can control degree of consistency (Q_R, Q_W and S) per instance (relation)

Performing Put/Get Operations

- Get/put requests handled by a coordinator (one of the nodes containing a replica of the item)
- Upon receiving a put() request for a key
  - the coordinator generates the vector clock for the new version and writes the new version locally
  - The coordinator then sends the new version (along with the new vector clock) to the N highest-ranked reachable nodes.
  - If at least Q_W-1 nodes respond then the write is considered successful.
- For a get() request
  - the coordinator requests all existing versions of data for that key from the N highest-ranked reachable nodes in the preference list for that key,
  - Waits for Q_R responses before returning the result to the client.
  - Returns all causally unrelated (incomparable) versions
  - Application should reconcile divergent versions and write back a reconciled version superseding the current versions

How to Reconcile Inconsistent Versions?

- Reconciliation is application specific
  - E.g., two sites concurrent insert items to cart
    - Merge adds both items to the final cart state
  - E.g., S1 adds item A, S2 deletes item B
    - Merge adds item A, but deleted item B resurfaces
    - Cannot distinguish S2 deletes B from S1 add B
    - Problem: operations are inferred from states of divergent versions
    - Better solution (not supported in Dynamo) keep track of history of operations

Availability vs Latency

- Abadi's classification system: **PACELC**
  - CAP theorem only matters when there is a partition
  - Even if partitions are rare, applications may trade off consistency for latency
    - E.g. PNUTS allows inconsistent reads to reduce latency
      - Critical for many applications
    - But update protocol (via master) ensures consistency over availability
  - Thus Abadi asks two questions:
    - If there is **P**artitioning, how does system tradeoff **A**vailability for **C**onsistency
    - **E**lse how does system trade off **L**atency for **C**onsistency
  - E.g., Megastore: PC/EC
    PNUTS: PC/EL
    Dynamo (by default): PA/EL

Amazon Dynamo

- Distributed data storage system supporting very high availability
  - Even at cost of consistency
  - E.g., motivation from Amazon: Web users should always be able to add items to their cart
    - Even if they are connected to an app server which is now in a minority partition
    - Data should be synchronized with majority partition eventually
    - Inconsistency may be visible (briefly) to users
      - preferable to losing a customer!
- DynamoDB: part of Amazon Web Service, can subscribe and use over the Web

Bully Algorithm Details

- If site S_i sends a request that is not answered by the coordinator within a time interval *T*, assume that the coordinator has failed S_i tries to elect itself as the new coordinator.
- S_i sends an election message to every site with a higher identification number, S_i then waits for any of these processes to answer within *T*.
- If no response within *T*, assume that all sites with number greater than *i* have failed, S_i elects itself the new coordinator.
- If answer is received S_i begins time interval *T'* , waiting to receive a message that a site with a higher identification number has been elected.

Bully Algorithm (Cont.)

- If no message is sent within *T'* , assume the site with a higher number has failed; S_i restarts the algorithm.
- After a failed site recovers, it immediately begins execution of the same algorithm.
- If there are no active sites with higher numbers, the recovered site forces all processes with lower numbers to let it become the coordinator site, even if there is a currently active coordinator with a lower number.

Chapter 24: Advanced Indexing

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Bloom Filters

- A **bloom filter** is a probabilistic data structure used to check membership of a value in a set
  - May return true (with low probability) even if an element is not present
  - But never returns false if an element is present
  - Used to filter out irrelevant sets
- Key data structure is a single bitmap
  - For a set with *n* elements, typical bitmap size is 10n
- Uses multiple independent hash functions
- With a single hash function h() with range=number of bits in bitmap:
  - For each element *s* in set *S* compute *h*(*s*) and set bit *h*(*s*)
  - To query an element *v* compute *h*(*v*), and check if bit *h*(*v*) is set
- Problem with single hash function: significant chance of false positive due to hash collision
  - 10% chance with 10n bits

Bloom Filters (Cont.)

- Key idea of Bloom filter: reduce false positives by use multiple hash functions $h_i()$ for $i = 1..k$
  - For each element s in set S *for each* i compute $h_i(s)$ and set bit $h_i(s)$
  - To query an element v *for each* i compute $h_i(v)$, and check if bit $h_i(v)$ is set
    - If bit $h_i(v)$ is set for every i then report v as present in set
    - Else report v as absent
  - With 10n bits, and $k = 7$, false positive rate reduces to 1% instead of 10% with $k = 1$

Write Optimized Indices

- Performance of
- B^+-trees can be poor for write-intensive workloads
  - One I/O per leaf, assuming all internal nodes are in memory
  - With magnetic disks, < 100 inserts per second per disk
  - With flash memory, one page overwrite per insert
- Two approaches to reducing cost of writes
  - **Log-structured merge tree**
  - **Buffer tree**

Log Structured Merge (LSM) Tree

- Consider only inserts/queries for now
- Records inserted first into in-memory tree (L_0 tree)
- When in-memory tree is full, records moved to disk (L_1 tree)
  - B^+-tree constructed using bottom-up build by merging existing L_1 tree with records from L_0 tree
- When L_1 tree exceeds some threshold, merge into L_2 tree
  - And so on for more levels
  - Size threshold for L_{i+1} tree is *k* times size threshold for L_i tree
  - Merge creates a new B^+-tree using bottom-up build

LSM Tree (Cont.)

- Benefits of LSM approach
  - Inserts are done using only sequential I/O operations
  - Leaves are full, avoiding space wastage
  - Reduced number of I/O operations per record inserted as compared to normal B^+-tree (up to some size)
    - If each leaf has *m* entries, *m/k* entries merged in using 1 IO
    - Total I/O operations: $k/m \log_k(I/M)$ where I = total number of entries, and M is the size of L_0 tree.
- Drawback of LSM approach
  - Queries have to search multiple trees
  - Entire content of each level copied multiple times

Optimizations of LSM

- **Rolling merge**
- LSM/Stepped Merge often implemented on a partitioned relation
  - Each partition size set to some max, split if over-sized
  - Spread partitions over multiple machines

Stepped Merge Index

- Stepped-merge index: variant of LSM tree with *k* trees at each level on disk
  - When all *k* indices exist at a level, merge them into one index of next level.
  - Reduces write cost compared to LSM tree
- But queries are even more expensive since many trees need to be queries
- Optimization for point lookups
  - Compute Bloom filter for each tree and store in-memory
  - Query a tree only if Bloom filter returns a positive result

LSM Trees (Cont.)

- Deletion handled by adding special “delete” entries
  - Lookups will find both original entry and the delete entry, and must return only those entries that do not have matching delete entry
  - When trees are merged, if we find a delete entry matching an original entry, both are dropped.
- Update handled using insert+delete
- LSM trees were introduced for disk-based indices
  - But useful to minimize erases with flash-based indices
  - The stepped-merge variant of LSM trees is used in many BigData storage systems
    - Google BigTable, Apache Cassandra, MongoDB
    - And more recently in SQLite4, LevelDB, and MyRocks storage engine of MySQL

Buffer Tree

- Alternative to LSM tree
- Key idea: each internal node of B^+-tree has a buffer to store inserts
  - Inserts are moved to lower levels when buffer is full
  - With a large buffer, many records are moved to lower level each time
  - Per record I/O decreases correspondingly
- Benefits
  - Less overhead on queries
  - Can be used with any tree index structure
  - Used in PostgreSQL Generalized Search Tree (GiST) indices
- Drawback: more random I/O than LSM tree

Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially from, say, 0
  - Given a number *n* it must be easy to retrieve record *n*
    - Particularly easy if records are of fixed size
- Applicable on attributes that take on a relatively small number of distinct values
  - E.g. gender, country, state, …
  - E.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000-infinity)
- A bitmap is simply an array of bits

Bitmap Indices (Cont.)

- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute
  - Bitmap has as many bits as records
  - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

record number	*ID*	*gender*	*income_level*
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Bitmap Indices (Cont.)

- Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
  - Intersection (and)
  - Union (or)
  - Complementation (not)
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
  - E.g. 100110 AND 110011 = 100010

    100110 OR 110011 = 110111

    NOT 100110 = 011001
  - Males with income level L1: 10010 AND 10100 = 10000
    - Can then retrieve required tuples.
    - Counting number of matching tuples is even faster

Bitmap Indices (Cont.)

- Bitmap indices generally very small compared with relation size
  - E.g. if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation.
    - If number of distinct attribute values is 8, bitmap is only 1% of relation size
- Deletion needs to be handled properly
  - **Existence bitmap** to note if there is a valid record at a record location
  - Needed for complementation
    - not(*A*=*v*): *(NOT bitmap-A-v) AND ExistenceBitmap*
- Should keep bitmaps for all values, even null value
  - To correctly handle SQL null semantics for NOT(*A*=*v*):
    - intersect above result with (NOT *bitmap-A-Null*)

Efficient Implementation of Bitmap Operations

- Bitmaps are packed into words; a single word and (a basic CPU instruction) computes and of 32 or 64 bits at once
  - E.g. 1-million-bit maps can be and-ed with just 31,250 instruction
- Counting number of 1s can be done fast by a trick:
  - Use each byte to index into a precomputed array of 256 elements each storing the count of 1s in the binary representation
    - Can use pairs of bytes to speed up further at a higher memory cost
  - Add up the retrieved counts
- Bitmaps can be used instead of Tuple-ID lists at leaf levels of B^+-trees, for values that have a large number of matching records
  - Worthwhile if > 1/64 of the records have that value, assuming a tuple-id is 64 bits
  - Above technique merges benefits of bitmap and B^+-tree indices

SPATIAL AND TEMPORAL INDICES

Spatial Data

- Databases can store data types such as lines, polygons, in addition to raster images
  - allows relational databases to store and retrieve spatial information
  - Queries can use spatial conditions (e.g. contains or overlaps).
  - queries can mix spatial and nonspatial conditions
- **Nearest neighbor queries**, given a point or an object, find the nearest object that satisfies given conditions.
- **Range queries** deal with spatial regions. e.g., ask for objects that lie partially or fully inside a specified region.
- Queries that compute intersections or unions of regions.
- **Spatial join** of two spatial relations with the location playing the role of join attribute.

Indexing of Spatial Data

- **k-d tree** - early structure used for indexing in multiple dimensions.
- Each level of a *k-d* tree partitions the space into two.
  - choose one dimension for partitioning at the root level of the tree.
  - choose another dimensions for partitioning in nodes at the next level and so on, cycling through the dimensions.
- In each node, approximately half of the points stored in the sub-tree fall on one side and half on the other.
- Partitioning stops when a node has less than a given number of points.

- The **k-d-B tree** extends the *k-d* tree to allow multiple child nodes for each internal node; well-suited for secondary storage.

Division of Space by Quadtrees

Quadtrees

- Each node of a quadtree is associated with a rectangular region of space; the top node is associated with the entire target space.
- Each non-leaf nodes divides its region into four equal sized quadrants
  - correspondingly each such node has four child nodes corresponding to the four quadrants and so on
- Leaf nodes have between zero and some fixed maximum number of points (set to 1 in example).

Quadtrees (Cont.)

- **PR quadtree**: stores points; space is divided based on regions, rather than on the actual set of points stored.
- **Region quadtrees** store array (raster) information.
  - A node is a leaf node is all the array values in the region that it covers are the same. Otherwise, it is subdivided further into four children of equal area, and is therefore an internal node.
  - Each node corresponds to a sub-array of values.
  - The sub-arrays corresponding to leaves either contain just a single array element, or have multiple array elements, all of which have the same value.
- Extensions of *k-d* trees and PR quadtrees have been proposed to index line segments and polygons
  - Require splitting segments/polygons into pieces at partitioning boundaries
    - Same segment/polygon may be represented at several leaf nodes

R-Trees

- **R-trees** are a N-dimensional extension of B^+-trees, useful for indexing sets of rectangles and other polygons.
- Supported in many modern database systems, along with variants like R^+ -trees and R*-trees.
- Basic idea: generalize the notion of a one-dimensional interval associated with each B+ -tree node to an N-dimensional interval, that is, an N-dimensional rectangle.
- Will consider only the two-dimensional case ($N = 2$)
  - generalization for $N > 2$ is straightforward, although R-trees work well only for relatively small N
- A polygon is stored only in one node, and the bounding box of the node must contain the polygon
  - The storage efficiency or R-trees is better than that of k-d trees or quadtrees since a polygon is stored only once

Example R-Tree

- The **bounding box** of a node is a minimum sized rectangle that contains all the rectangles/polygons associated with the node
  - *Bounding boxes of children of a node are allowed to overlap*

Rectangles being Indexed

R-Tree

Search in R-Trees

- To find data items intersecting a given query point/region, do the following, starting from the root node:
  - If the node is a leaf node, output the data items whose keys intersect the given query point/region.
  - Else, for each child of the current node whose bounding box intersects the query point/region, recursively search the child

Search in R-Trees (Cont.)

- Can be very inefficient in worst case since multiple paths may need to be searched
  - but works acceptably in practice.
- Simple extensions of search procedure to handle predicates *contained-in* and *contains*

Insertion in R-Trees

- To insert a data item:
  - Find a leaf to store it, and add it to the leaf
    - To find leaf, follow a child (if any) whose bounding box contains bounding box of data item, else child whose overlap with data item bounding box is maximum
  - Handle overflows by splits (as in B+ -trees)
    - Split procedure is different though (see below)
  - Adjust bounding boxes starting from the leaf upwards
- Split procedure:
  - Goal: divide entries of an overfull node into two sets such that the bounding boxes have minimum total area
    - This is a heuristic. Alternatives like minimum overlap are possible
  - Finding the "best" split is expensive, use heuristics instead
    - See next slide

Splitting an R-Tree Node

- **Quadratic split** divides the entries in a node into two new nodes as follows
  1. Find pair of entries with "maximum separation"
     - that is, the pair such that the bounding box of the two would has the maximum wasted space (area of bounding box – sum of areas of two entries)
  2. Place these entries in two new nodes
  3. Repeatedly find the entry with "maximum preference" for one of the two new nodes, and assign the entry to that node
     - Preference of an entry to a node is the increase in area of bounding box if the entry is added to the *other* node
  4. Stop when half the entries have been added to one node
     - Then assign remaining entries to the other node
- Cheaper **linear split** heuristic works in time linear in number of entries,
  - Cheaper but generates slightly worse splits.

Deleting in R-Trees

- Deletion of an entry in an R-tree done much like a B^+-tree deletion.
  - In case of underfull node, borrow entries from a sibling if possible, else merging sibling nodes
  - Alternative approach removes all entries from the underfull node, deletes the node, then reinserts all entries

Indexing Temporal Data

- Temporal data refers to data that has an associated time period (interval)
- Time interval has a start and end time
  - End time set to infinity (or large date such as 9999-12-31) if a tuple is currently valid and its validity end time is not currently known
- Query may ask for all tuples that are valid at a point in time or during a time interval
  - Index on valid time period speeds up this task

course_id	*title*	*dept_name*	*credits*	*start*	*end*
BIO-101	Intro. to Biology	Biology	4	1985-01-01	9999-12-31
CS-201	Intro. to C	Comp. Sci.	4	1985-01-01	1999-01-01
CS-201	Intro. to Java	Comp. Sci.	4	1999-01-01	2010-01-01
CS-201	Intro. to Python	Comp. Sci.	4	2010-01-01	9999-12-31

Figure 7.17 A temporal version of the *course* relation

Indexing Temporal Data (Cont.)

- To create a temporal index on attribute *a*:
  - Use spatial index, such as R-tree, with attribute *a* as one dimension, and time as another dimension
    - Valid time forms an interval in the time dimension
  - Tuples that are currently valid cause problems, since value is infinite or very large
    - Solution: store all current tuples (with end time as infinity) in a separate index, indexed on (*a, start-time*)
      - To find tuples valid at a point in time *t* in the current tuple index, search for tuples in the range (*a, 0*) to (*a,t*)
- Temporal index on primary key can help enforce temporal primary key constraint

course_id	*title*	*dept_name*	*credits*	*start*	*end*
BIO-101	Intro. to Biology	Biology	4	1985-01-01	9999-12-31
CS-201	Intro. to C	Comp. Sci.	4	1985-01-01	1999-01-01
CS-201	Intro. to Java	Comp. Sci.	4	1999-01-01	2010-01-01
CS-201	Intro. to Python	Comp. Sci.	4	2010-01-01	9999-12-31

Hashing

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Static Hashing

- A **bucket** is a unit of storage containing one or more entries (a bucket is typically a disk block).
  - we obtain the bucket of an entry from its search-key value using a **hash function**
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B.*
- Hash function is used to locate entries for access, insertion as well as deletion.
- Entries with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate an entry.
- In a **hash index**, buckets store entries with pointers to records
- In a **hash file-organization** buckets store records

Hash Functions

- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.
- Typical hash functions perform computation on the internal binary representation of the search-key.
  - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned.

Example of Hash File Organization

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Hash file organization of *instructor* file, using *dept_name* as key.

Handling of Bucket Overflows

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using ***overflow buckets***.

Handling of Bucket Overflows (Cont.)

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed addressing (**also called **closed hashing or open hashing** depending on the book you use**)**
  - An alternative, called **open addressing (**also called **open hashing** or **closed hashing** depending on the book you use) which does not use overflow buckets, is not suitable for database applications.

Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses. Databases grow or shrink with time.
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
  - If database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically.

Dynamic Hashing

- Periodic rehashing
  - If number of entries in a hash table becomes (say) 1.5 times size of hash table,
    - create new hash table of size (say) 2 times the size of the previous hash table
    - Rehash all entries to new table
- Linear Hashing
  - Do rehashing in an incremental manner
- Extendable Hashing
  - Tailored to disk based hashing, with buckets shared by multiple hash values
  - Doubling of # of entries in hash table, without doubling # of buckets

Extendable Hashing

- **Extendable hashing** – one form of dynamic hashing
  - Hash function generates values over a large range — typically *b*-bit integers, with $b = 32$.
  - At any time use only a prefix of the hash function to index into a table of bucket addresses.
  - Let the length of the prefix be *i* bits, $0 \leq i \leq 32$.
    - Bucket address table size = 2^{i}. Initially $i = 0$
    - Value of *i* grows and shrinks as the size of the database grows and shrinks.
  - Multiple entries in the bucket address table may point to a bucket (why?)
  - Thus, actual number of buckets is $< 2^{i}$
    - The number of buckets also changes dynamically due to coalescing and splitting of buckets.

General Extendable Hash Structure

In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$ (see next slide for details)

Use of Extendable Hash Structure

- Each bucket j stores a value i_j
  - All the entries that point to the same bucket have the same values on the first i_j bits.
- To locate the bucket containing search-key K_j:
  1. Compute $h(K_j) = X$
  2. Use the first i high order bits of X as a displacement into bucket address table, and follow the pointer to appropriate bucket
- To insert a record with search-key value K_j
  - follow same procedure as look-up and locate the bucket, say j.
  - If there is room in the bucket j insert record in the bucket.
  - Else the bucket must be split and insertion re-attempted (next slide.)
    - Overflow buckets used instead in some cases (will see shortly)

Insertion in Extendable Hash Structure (Cont)

To split a bucket j when inserting record with search-key value K_j:

- If $i > i_j$ (more than one pointer to bucket j)
  - allocate a new bucket z, and set $i_j = i_z = (i_j + 1)$
  - Update the second half of the bucket address table entries originally pointing to j, to point to z
  - remove each record in bucket j and reinsert (in j or z)
  - recompute new bucket for K_j and insert record in the bucket (further splitting is required if the bucket is still full)
- If $i = i_j$ (only one pointer to bucket j)
  - If i reaches some limit b, or too many splits have happened in this insertion, create an overflow bucket
  - Else
    - increment i and double the size of the bucket address table.
    - replace each entry in the table by two entries that point to the same bucket.
    - recompute new bucket address table entry for K_j
      Now $i > i_j$ so use the first case above.

Deletion in Extendable Hash Structure

- To delete a key value,
  - locate it in its bucket and remove it.
  - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
  - Coalescing of buckets can be done (can coalesce only with a "*buddy*" bucket having same value of i_j and same $i_j - 1$ prefix, if it is present)
  - Decreasing bucket address table size is also possible
    - Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

Example (Cont.)

- Initial Hash structure; bucket size = 2

Example (Cont.)

- Hash structure after insertion of “Mozart”, “Srinivasan”, and “Wu” records

Example (Cont.)

- Hash structure after insertion of Einstein record

Example (Cont.)

- Hash structure after insertion of Gold and El Said records

Example (Cont.)

- Hash structure after insertion of Katz record

Example (Cont.)

And after insertion of eleven records

Example (Cont.)

And after insertion of Kim record in previous hash structure

Extendable Hashing vs. Other Schemes

- Benefits of extendable hashing:
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
- Disadvantages of extendable hashing
  - Extra level of indirection to find desired record
  - Bucket address table may itself become very big (larger than memory)
    - Cannot allocate very large contiguous areas on disk either
    - Solution: B^+-tree structure to locate desired record in bucket address table
  - Changing size of bucket address table is an expensive operation
- **Linear hashing** is an alternative mechanism
  - Allows incremental growth of its directory (equivalent to bucket address table)
  - At the cost of more bucket overflows

Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indices are preferred
- In practice:
  - Hash-indices are extensively used in-memory
  - But not used much on fisk
    - Oracle supports static hash organization, but not hash indices
    - SQL Server and PostgreSQL do not support hashing on disk

End of Chapter 24

Partitioned Hashing

- Hash values are split into segments that depend on each attribute of the search-key.

  $(A_1, A_2, \ldots, A_n)$ for n attribute search-key

- Example: $n = 2$, for *customer,* search-key being (*customer-street, customer-city*)

search-key value	*hash value*
(Main, Harrison)	101 111
(Main, Brooklyn)	101 001
(Park, Palo Alto)	010 010
(Spring, Brooklyn)	001 001
(Alma, Palo Alto)	110 010

- To answer equality query on single attribute, need to look up multiple buckets. Similar in effect to grid files.

Chapter 25: Advanced Application Development

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Hardware Tuning: Choice of RAID Level

- To use RAID 1 or RAID 5?
  - Depends on ratio of reads and writes
    - RAID 5 requires 2 block reads and 2 block writes to write out one data block
- If an application requires *r* reads and *w* writes per second
  - RAID 1 requires $r + 2w$ I/O operations per second
  - RAID 5 requires: $r + 4w$ I/O operations per second
- For reasonably large r and w, this requires lots of disks to handle workload
  - RAID 5 may require more disks than RAID 1 to handle load!
  - Apparent saving of number of disks by RAID 5 (by using parity, as opposed to the mirroring done by RAID 1) may be illusory!
- Thumb rule: RAID 5 is fine when writes are rare and data is very large, but RAID 1 is preferable otherwise
  - If you need more disks to handle I/O load, just mirror them since disk capacities these days are enormous!

Tuning the Database Design (Cont.)

Materialized Views

- Materialized views can help speed up certain queries
  - Particularly aggregate queries
- Overheads
  - Space
  - Time for view maintenance
    - Immediate view maintenance: done as part of update txn
      - time overhead paid by update transaction
    - Deferred view maintenance: done only when required
      - update transaction is not affected, but system time is spent on view maintenance
        - until updated, the view may be out-of-date
- Preferable to denormalized schema since view maintenance is systems responsibility, not programmers
  - Avoids inconsistencies caused by errors in update programs

Tuning the Database Design (Cont.)

- How to choose set of materialized views
  - Helping one transaction type by introducing a materialized view may hurt others
  - Choice of materialized views depends on costs
    - Users often have no idea of actual cost of operations
  - Overall, manual selection of materialized views is tedious
- Some database systems provide tools to help DBA choose views to materialize
  - “Materialized view selection wizards”

Tuning of Transactions (Cont.)

- Reducing lock contention
- Long transactions (typically read-only) that examine large parts of a relation result in lock contention with update transactions
  - E.g., large query to compute bank statistics and regular bank transactions
- To reduce contention
  - Use multi-version concurrency control
    - E.g., Oracle “snapshots” which support multi-version 2PL
  - Use degree-two consistency (cursor-stability) for long transactions
    - Drawback: result may be approximate

Tuning of Transactions (Cont.)

- Long update transactions cause several problems
  - Exhaust lock space
  - Exhaust log space
    - and also greatly increase recovery time after a crash, and may even exhaust log space during recovery if recovery algorithm is badly designed!
- Use **mini-batch** transactions to limit number of updates that a single transaction can carry out. E.g., if a single large transaction updates every record of a very large relation, log may grow too big.
  - Split large transaction into batch of “mini-transactions," each performing part of the updates
  - Hold locks across transactions in a mini-batch to ensure serializability
    - If lock table size is a problem can release locks, but at the cost of serializability
  - In case of failure during a mini-batch, must complete its remaining portion on recovery, to ensure atomicity.

Performance Simulation

- **Performance simulation** using queuing model useful to predict bottlenecks as well as the effects of tuning changes, even without access to real system
- Queuing model as we saw earlier
  - Models activities that go on in parallel
- Simulation model is quite detailed, but usually omits some low level details
  - Model **service time**, but disregard details of service
  - E.g., approximate disk read time by using an average disk read time
- Experiments can be run on model, and provide an estimate of measures such as average throughput/response time
- Parameters can be tuned in model and then replicated in real system
  - E.g., number of disks, memory, algorithms, etc.

Database Application Classes

- **Online transaction processing (OLTP)**
  - requires high concurrency and clever techniques to speed up commit processing, to support a high rate of update transactions.
- **Decision support applications**
  - including **online analytical processing, or OLAP** applications
  - require good query evaluation algorithms and query optimization.
- Architecture of some database systems tuned to one of the two classes
  - E.g., Teradata is tuned to decision support
- Others try to balance the two requirements
  - E.g., Oracle, with snapshot support for long read-only transaction

Benchmarks Suites (Cont.)

- TPC benchmarks (cont.)
  - **TPC-D**: complex decision support application
    - Superceded by TPC-H and TPC-R
  - **TPC-H:** (H for ad hoc) based on TPC-D with some extra queries
    - Models ad hoc queries which are not known beforehand
      - Total of 22 queries with emphasis on aggregation
    - prohibits materialized views
    - permits indices only on primary and foreign keys
  - **TPC-R:** (R for reporting) same as TPC-H, but without any restrictions on materialized views and indices
  - **TPC-W**: (W for Web) End-to-end Web service benchmark modeling a Web bookstore, with combination of static and dynamically generated pages

TPC Performance Measures

- Two types of tests for TPC-H and TPC-R
  - **Power test**: runs queries and updates sequentially, then takes mean to find queries per hour
  - **Throughput test**: runs queries and updates concurrently
    - multiple streams running in parallel each generates queries, with one parallel update stream
  - **Composite query per hour metric**: square root of product of power and throughput metrics
  - **Composite price/performance metric**

Other Benchmarks

- OODB transactions require a different set of benchmarks.
  - OO7 benchmark has several different operations, and provides a separate benchmark number for each kind of operation
  - Reason: hard to define what is a typical OODB application
- Benchmarks for XML being discussed

SQL Standards History (Cont.)

- SQL:1999
  - Adds variety of new features --- extended data types, object orientation, procedures, triggers, etc.
  - Broken into several parts
    - SQL/Framework (Part 1): overview
    - SQL/Foundation (Part 2): types, schemas, tables, query/update statements, security, etc.
    - SQL/CLI (Call Level Interface) (Part 3): API interface
    - SQL/PSM (Persistent Stored Modules) (Part 4): procedural extensions
    - SQL/Bindings (Part 5): embedded SQL for different embedding languages

SQL Standards History (Cont.)

- More parts undergoing standardization process
  - Part 7: SQL/Temporal: temporal data
  - Part 9: SQL/MED (Management of External Data)
    - Interfacing of database to external data sources
      - Allows other databases, even files, can be viewed as part of the database
  - Part 10 SQL/OLB (Object Language Bindings): embedding SQL in Java
  - Missing part numbers 6 and 8 cover features that are not near standardization yet

XML-Based Standards

- Several XML based Standards for E-commerce
  - E.g., RosettaNet (supply chain), BizTalk
  - Define catalogs, service descriptions, invoices, purchase orders, etc.
  - XML wrappers are used to export information from relational databases to XML
- Simple Object Access Protocol (SOAP): XML based remote procedure call standard
  - Uses XML to encode data, HTTP as transport protocol
  - Standards based on SOAP for specific applications
    - E.g., OLAP and Data Mining standards from Microsoft

E-Commerce

- E-commerce is the process of carrying out various activities related to commerce through electronic means
- Activities include:
  - Presale activities: catalogs, advertisements, etc.
  - Sale process: negotiations on price/quality of service
  - Marketplace: e.g., stock exchange, auctions, reverse auctions
  - Payment for sale
  - Delivery related activities: electronic shipping, or electronic tracking of order processing/shipping
  - Customer support and post-sale service

E-Catalogs

- Product catalogs must provide searching and browsing facilities
  - Organize products into intuitive hierarchy
  - Keyword search
  - Help customer with comparison of products
- Customization of catalog
  - Negotiated pricing for specific organizations
  - Special discounts for customers based on past history
    - E.g., loyalty discount
  - Legal restrictions on sales
    - Certain items not exposed to under-age customers
- Customization requires extensive customer-specific information

Marketplaces

- Marketplaces help in negotiating the price of a product when there are multiple sellers and buyers
- Several types of marketplaces
  - Reverse auction
  - Auction
  - Exchange
- Real world marketplaces can be quite complicated due to product differentiation
- Database issues:
  - Authenticate bidders
  - Record buy/sell bids securely
  - Communicate bids quickly to participants
    - Delays can lead to financial loss to some participants
  - Need to handle very large volumes of trade at times
    - E.g., at the end of an auction

Types of Marketplace

- **Reverse auction system:** single buyer, multiple sellers.
  - Buyer states requirements, sellers bid for supplying items. Lowest bidder wins. (also known as tender system)
  - **Open bidding** vs. **closed bidding**
- **Auction:** Multiple buyers, single seller
  - Simplest case: only one instance of each item is being sold
  - Highest bidder for an item wins
  - More complicated with multiple copies, and buyers bid for specific number of copies
- **Exchange:** multiple buyers, multiple sellers
  - E.g., stock exchange
  - Buyers specify maximum price, sellers specify minimum price
  - exchange matches buy and sell bids, deciding on price for the trade
    - e.g., average of buy/sell bids

Order Settlement

- Order settlement: payment for goods and delivery
- Insecure means for electronic payment: send credit card number
  - Buyers may present some one else's credit card numbers
  - Seller has to be trusted to bill only for agreed-on item
  - Seller has to be trusted not to pass on the credit card number to unauthorized people
- Need secure payment systems
  - Avoid above-mentioned problems
  - Provide greater degree of privacy
    - E.g., not reveal buyers identity to seller
  - Ensure that anyone monitoring the electronic transmissions cannot access critical information

Secure Payment Systems

- All information must be encrypted to prevent eavesdropping
  - Public/private key encryption widely used
- Must prevent **person-in-the-middle attacks**
  - E.g., someone impersonates seller or bank/credit card company and fools buyer into revealing information
    - Encrypting messages alone doesn't solve this problem
    - More on this in next slide
- Three-way communication between seller, buyer and credit-card company to make payment
  - Credit card company credits amount to seller
  - Credit card company consolidates all payments from a buyer and collects them together
    - E.g., via buyer's bank through physical/electronic check payment

Secure Payment Systems (Cont.)

- **Digital certificates** are used to prevent impersonation/man-in-the middle attack
  - Certification agency creates digital certificate by encrypting, e.g., seller's public key using its own private key
    - Verifies sellers identity by external means first!
  - Seller sends certificate to buyer
  - Customer uses public key of certification agency to decrypt certificate and find sellers public key
    - Man-in-the-middle cannot send fake public key
  - Sellers public key used for setting up secure communication
- Several secure payment protocols
  - E.g., **Secure Electronic Transaction (SET)**

Digital Cash

- Credit-card payment does not provide anonymity
  - The SET protocol hides buyers identity from seller
  - But even with SET, buyer can be traced with help of credit card company
- Digital cash systems provide anonymity similar to that provided by physical cash
  - E.g., **Dig Cash**
  - Based on encryption techniques that make it impossible to find out who purchased digital cash from the bank
  - Digital cash can be spent by purchaser in parts
    - much like writing a check on an account whose owner is anonymous

Legacy Systems

- Legacy systems are older-generation systems that are incompatible with current generation standards and systems but still in production use
  - E.g., applications written in Cobol that run on mainframes
    - Today's hot new system is tomorrows legacy system!
- Porting legacy system applications to a more modern environment is problematic
  - Very expensive, since legacy system may involve millions of lines of code, written over decades
    - Original programmers usually no longer available
  - Switching over from old system to new system is a problem
    - more on this later
- One approach: build a **wrapper** layer on top of legacy application to allow interoperation between newer systems and legacy application
  - E.g., use ODBC or OLE-DB as wrapper

Legacy Systems (Cont.)

- Rewriting legacy application requires a first phase of understanding what it does
  - Often legacy code has no documentation or outdated documentation
  - **reverse engineering**: process of going over legacy code to
    - Come up with schema designs in ER or OO model
    - Find out what procedures and processes are implemented, to get a high level view of system
- **Re-engineering**: reverse engineering followed by design of new system
  - Improvements are made on existing system design in this process

Legacy Systems (Cont.)

- Switching over from old to new system is a major problem
  - Production systems are in every day, generating new data
  - Stopping the system may bring all of a company's activities to a halt, causing enormous losses
- **Big-bang approach**:
  1. Implement complete new system
  2. Populate it with data from old system
     1. No transactions while this step is executed
     2. scripts are created to do this quickly
  3. Shut down old system and start using new system
  - Danger with this approach: what if new code has bugs or performance problems, or missing features
    - Company may be brought to a halt

Legacy Systems (Cont.)

- **Chicken-little approach**:
  - Replace legacy system one piece at a time
  - Use wrappers to interoperate between legacy and new code
    - E.g., replace front end first, with wrappers on legacy backend
      - Old front end can continue working in this phase in case of problems with new front end
    - Replace back end, one functional unit at a time
      - All parts that share a database may have to be replaced together, or wrapper is needed on database also
  - Drawback: significant extra development effort to build wrappers and ensure smooth interoperation
    - Still worth it if company's life depends on system

End of Chapter 25

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Chapter 26: Blockchain Databases

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Outline

- Overview
- Blockchain Properties
- Achieving Blockchain Properties via Cryptographic Hash Functions
- Consensus
- Data Management in a Blockchain
- Smart Contracts
- Performance Enhancement
- Emerging Applications

History

- Blockchain technology's development was driven initially by **cryptocurrencies**.
  - But cryptocurrency is just one application of blockchain
- Cryptocurrencies:
  - Purely online
  - Maintained by a decentralized distributed ledger
- **Bitcoin**
  - One of the first successful cryptocurrencies, and the best known.
  - Published under the pseudonym Satoshi Nakomoto
  - Aims to be an anonymous, fully distributed and decentralized currency
  - There are other blockchains that are more appropriate for blockchain-based enterprise data

Types of Blockchain

- **Public**
  - Anyone can download the needed software and create a blockchain node
  - No trust assumed among participating nodes
- **Permissioned**
  - Permission to run a blockchain node is granted by a permissioning authority
  - Some degree of relaxation of the assumptions of trustlessness and autonomy
- The type of blockchain influences the choice of algorithm used by which nodes agree on the next block to be added to the blockchain

Blockchain Properties and Structure

- Linked list of blocks
  - Each block contains a pointer to the previous block plus a hash of the previous block
    - Except, of course, for the first block, called the **genesis block**
- **Tamper resistance**
  - The inclusion of the hash of the previous block makes tampering difficult
    - Changing the contents of a block means changes all newer blocks as well
  - Specific mathematical requirements for the hash function (see later)
  - Replication prevents replacement of the entire blockchain without gaining majority control

Blockchain Properties and Structure

- Node types
  - **Full** node – maintains copy of blockchain and participates in the consensus process
  - **Light** node – submits updates to the blockchain but does not participate in the consensus process
- Consensus algorithms to choose node to add next block:
  - **Proof of work** – first node to solve a certain hard math problem
  - **Proof of stake** – node selected based on amount of currency owned or held in reserve
  - **Byzantine consensus** – message-based protocol to reach consensus on next block
  - Other approaches – proof of activity, proof of burn, proof of capacity, proof of elapsed time

Blockchain Properties and Structure

- **Forks**
  - A fork occurs if a block is added to a block that is not the most recent one
  - Accidental if consequence of the consensus algorithm
    - One fork survives; others are eventually **orphaned**
  - Malicious if due to an attempt to damage the blockchain
  - Consensus-based if agreed to by majority of users
- Consensus-based fork types
  - **Hard** – old version of blockchain software does not recognize new blocks as valid
  - **Soft** – old version of blockchain software recognizes new blocks as valid

Blockchain Properties and Structure

- Digital signature
  - Public-key encryption is used to allow users to sign their transactions.
  - Ensures that users cannot deny submitting the transaction, a property called **irrefutability**.
- Anonymity
  - Users can remain anonymous unless there is a way to tie the user's ID (public key) to a real-world entity
- Summary of blockchain properties:
  - **Decentralization** – majority consensus with no central authority.
  - **Tamper resistance** – infeasibility of changing the contents of blocks on the blockchain.
  - **Irrefutability** – user cannot deny having submitted a transaction.
  - **Anonymity** – IDs not directly tied to any real-world entity

Cryptographic Hash Functions

- Let *h* denote a cryptographic hash function. Then *h* must satisfy the following properties:
  - **Collision resistant** – It is infeasible to find two distinct values *x* and *y* such that h(x) = h(y)
  - **Irreversible** – Given h(x), it is infeasible to find x.
- By ***infeasible***, we mean that there is strong mathematical evidence, if not an actual proof, that there is no approach to obtaining an answer that is better than guessing from the set of all possibilities.

Blockchain Transactions

- Exact transaction model is specific to each blockchain.
- Bitcoin
  - No account balances stored directly.
  - A transaction specifies:
    - Input transactions (whose output are spent by this transaction)
    - A set of outputs, each specifying the recipient and the amount
    - A digital signature from the user submitting the transaction
  - Additionally a Bitcoin transaction may:
    - Store a small amount of data on the blockchain
    - Specify a slightly more complex transaction using the Bitcoin scripting language
- Ethereum
  - Maintains account balances, which are modified by transactions
  - Has a more sophisticated, Turing-complete scripting language

Consensus

- All nodes must agree on additions to the blockchain
- In a decentralized system like a blockchain system, there is no central coordinator (unlike the case for 2PC and 3PC)
- Categorization of consensus algorithms:
  - **Proof of Work**
    - Node needs to solve a cryptographic puzzle in order to add a block (more on next slide)
  - **Proof of Stake**
    - Node is chosen to add next block based on amount of currency held, with probability proportionate to stake
  - **Byzantine Consensus**
    - Node is chosen to add next block based on a message-passing-based consensus algorithm (more later)

Proof of Work

- To add a block B, a node needs to find a **nonce**, *n*, such that the value of the hash function *h* applied to the concatenation of *n* and *B* (n || B) is less that some specified value.
- The function *h* must have the **puzzle-friendliness** property: Given *k* and an *n*-bit value *y*, it is infeasible to find *x* such that h(x || k) = y in time significantly less than 2^n.
- Forks
  - If more than one node solves the puzzle around the same time, two blocks could be added after the most recent block
  - The result is a fork
  - Since nodes attempt to add to the most recent block of the longest chain, eventually blocks on shorter forks are orphaned (slide 7)

Byzantine Consensus

- **Byzantine failure**: A failed node can behave in an arbitrarily bad manner, including taking the exactly correct set of steps to sabotage the system
  - Contrast with 2PC, 3PC, Paxos, and Raft (Chapter 23), where the only assumed type of failure is a crash. This is called the **fail-stop** model of failure.
- Achieving consensus with Byzantine failure that at most (n-1)/3 nodes fail, where *n* is the total number of nodes.
- Traditional Byzantine consensus algorithms assume that *n* does not change.
  - In a blockchain system, however, nodes can join and leave.
  - Modern blockchain Byzantine consensus algorithms are robust to the number of nodes changing

Sybil Attacks

- A **Sybil attack** is an attempt to overwhelm the consensus algorithm by adding a large number of nodes.
- Protection against Sybil attack:
  - Proof of work: Hard for an attacker to control a majority of the computing power in the network, thus making it hard to dominate success in solving the cryptographic puzzle.
  - Proof of stake: Costly to acquire a majority of all outstanding currency.
  - Byzantine consensus: Vulnerable to attack unless there is a permissioning mechanism for new nodes:
    - Trusted permission-granting agent.
    - A decentralized trust-based feature in the protocol itself.

Data Management in a Blockchain

- Validating a transaction requires data look-up in the blockchain:
  - Bitcoin: Look up input transactions to ensure that their output has not already been spend ("**double-spend**").
  - Ethereum: Look up account balances.
- Blockchains use the **Merkle-tree** data structure (Chapter 23):
  - Allows a node to store just the root-hash of the Merkle tree for verification purposes, rather than the entire blockchain.
  - But blockchain immutability means the tree structure can't be updated in place.
- **Merkle-Patricia-tree** structure:
  - Patricia-tree structure allows key-based search.
  - Updates performed by creating a new root that points to unchanged parts of the data structure.

Smart Contracts

- Transaction execution is specified by code
  - Bitcoin: a relatively simple stack-based scripting language with instructions designed for funds-transfer, including the **multisig** instruction: *m* of *n* users must approve to enable escrow transactions.
    - Infinite loops not possible due to the limited power of the language.
  - Ethereum: a scripting Turing-complete language:
    - Based on the **Ethereum virtual machine** (EVM)
    - **Solidity**: A high-level language compiled to EVM code
    - Greater expressive power but risk of infinite loops
      - Need mechanism for terminating such loops despite undecidability of the halting problem
      - A cost-per-instruction framework serves both to avoid infinite loops and to incentivize miners (next slide)

Concept of “Gas” in Ethereum

- Each Ethereum instruction has a fixed cost, denominated in **gas**.
- Each smart contract sets its own gas price in Ether defining the amount paid to the miner for one unit of gas.
- How to set price wisely:
  - Gas price too low: miners disincented to include transaction.
  - Gas price too high: user overpaid.
- Limits
  - Transaction gas limit: upper bound on transaction gas usage, set by user.
  - Block gas limit: upper bound on gas usage in a block, set by the system.
- Infinite loops impossible since the transaction gas limit is reached eventually, at which point the transaction fails and terminates, (though the miner keeps the payment received).

©Silberschatz, Korth and Sudarshan

Smart Contracts

- External input:
  - Messages from other contracts
  - Input pertaining to the outside world from trusted sources called **oracles**.
- Autonomy:
  - An Ethereum smart contract may run indefinitely by receiving Ether from external sources to fund its continued operation. Such contracts are called **distributed autonomous organizations** (DAOs).
- A smart contract may be used to create a separate currency or token on top of the Ethereum blockchain.
  - The **ERC-20** standard is the most widely used
- Smart contracts may be used in the implementation **of cross-chain transactions**, allowing transactions between separate blockchain systems

Performance Enhancement

- The consensus mechanism is an important factor in blockchain performance, as discussed earlier.
- Other ways to enhance performance include
  - **Sharding**: parallelizing the mining of new blocks
  - **Off-chain transaction processing**: Creation of a separate channel for users who process transactions among themselves frequently.
    - Channel funded with funds from the underlying blockchain.
    - Routine transactions avoid mining overhead.
    - Users can terminate the agreement to process transactions off-chain at any point, with current channel funds balances refunded on the underlying blockchain.
  - Database-style blockchain data structures.

Emerging Applications

- Academic transcript distribution
- Accounting and audit
- Asset management
- E-Government
- Foreign-currency exchange
- Health care
- Insurance claims
- Internet of Things
- Loyalty programs
- Supply chain
- Ticket sales and re-sales
- Trade finance
- and many more

End of Chapter 26

Chapter 27: Formal-Relational Query Languages

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Outline

- Tuple Relational Calculus
- Domain Relational Calculus
- Datalog

Tuple Relational Calculus

Tuple Relational Calculus

- A nonprocedural query language, where each query is of the form

$$\{t \mid P(t)\}$$

- It is the set of all tuples t such that predicate P is true for t
- t is a *tuple variable*, $t[A]$ denotes the value of tuple t on attribute A
- $t \in r$ denotes that tuple t is in relation r
- P is a *formula* similar to that of the predicate calculus

Predicate Calculus Formula

1. Set of attributes and constants
2. Set of comparison operators: (e.g., $<, \leq, =, \neq, >, \geq$)
3. Set of connectives: and ($\wedge$), or (v), not ($\neg$)
4. Implication ($\Rightarrow$): x $\Rightarrow$ y, if x if true, then y is true

$$x \Rightarrow y \equiv \neg x \vee y$$

5. Set of quantifiers:
   - $\exists\, t \in r\, (Q\, (t\,)) \equiv$ ”there exists” a tuple in t in relation r such that predicate $Q\, (t\,)$ is true
   - $\forall t \in r\, (Q\, (t\,)) \equiv Q$ is true “for all” tuples t in relation r

Example Queries

- Find the *ID, name, dept_name, salary* for instructors whose salary is greater than $80,000

$$\{t \mid t \in instructor \wedge t\,[salary\,] > 80000\}$$

Notice that a relation on schema (*ID, name, dept_name, salary*) is implicitly defined by the query

- As in the previous query, but output only the *ID* attribute value

$$\{t \mid \exists\, s \in \text{instructor}\ (t\,[ID\,] = s\,[ID\,] \wedge s\,[salary\,] > 80000)\}$$

Notice that a relation on schema (*ID*) is implicitly defined by the query

Example Queries

- Find the names of all instructors whose department is in the Watson building

$$\{t \mid \exists s \in instructor\ (t\,[name\,] = s\,[name\,]$$
$$\wedge\ \exists u \in department\ (u\,[dept_name\,] = s[dept_name]\ \text{“}$$
$$\wedge\ \ u\,[building] = \text{“Watson”}\ ))\}$$

- Find the set of all courses taught in the Fall 2009 semester, or in the Spring 2010 semester, or both

$$\{t \mid \exists s \in section\ (t\,[course_id\,] = s\,[course_id\,] \wedge$$
$$s\,[semester] = \text{“Fall”} \wedge s\,[year] = 2009$$
$$\vee\ \exists u \in section\ (t\ \ [course_id\,] = u\,[course_id\,] \wedge$$
$$u\,[semester] = \text{“Spring”} \wedge u\,[year] = 2010\ )\}$$

Example Queries

- Find the set of all courses taught in the Fall 2009 semester, and in the Spring 2010 semester

$$\{t \mid \exists s \in section\ (t\,[course_id\,] = s\,[course_id\,] \wedge$$
$$s\,[semester] = \text{“Fall”} \wedge s\,[year] = 2009$$
$$\wedge\ \exists u \in section\ (t\,[course_id\,] = u\,[course_id\,] \wedge$$
$$u\,[semester] = \text{“Spring”} \wedge u\,[year] = 2010\ )\}$$

- Find the set of all courses taught in the Fall 2009 semester, but not in the Spring 2010 semester

$$\{t \mid \exists s \in section\ (t\,[course_id\,] = s\,[course_id\,] \wedge$$
$$s\,[semester] = \text{“Fall”} \wedge s\,[year] = 2009$$
$$\wedge\ \neg\ \exists u \in section\ (t\,[course_id\,] = u\,[course_id\,] \wedge$$
$$u\,[semester] = \text{“Spring”} \wedge u\,[year] = 2010\ )\}$$

Universal Quantification

- Find all students who have taken all courses offered in the Biology department
  - $\{t \mid \exists\, r \in student\ (t\,[ID] = r\,[ID]) \wedge$
    $(\forall\, u \in course\ (u\,[dept_name]=\text{“Biology”} \Rightarrow$
    $\exists\, s \in takes\ (t\,[ID] = s\,[ID\,] \wedge$
    $s\,[course_id] = u\,[course_id]))\}$

Safety of Expressions

- It is possible to write tuple calculus expressions that generate infinite relations.
- For example, { t | $\neg$ $t \in r$ } results in an infinite relation if the domain of any attribute of relation *r* is infinite
- To guard against the problem, we restrict the set of allowable expressions to safe expressions.
- An expression {*t* | *P* (*t*)} in the tuple relational calculus is *safe* if every component of *t* appears in one of the relations, tuples, or constants that appear in *P*
  - NOTE: this is more than just a syntax condition.
    - E.g. { *t* | *t* [*A*] = 5 $\vee$ **true** } is not safe --- it defines an infinite set with attribute values that do not appear in any relation or tuples or constants in *P*.

Safety of Expressions (Cont.)

- Consider again that query to find all students who have taken all courses offered in the Biology department
  - $\{t \mid \exists\, r \in student\ (t\,[ID] = r\,[ID]) \wedge$
    $(\forall\, u \in course\ (u\,[dept_name]=\text{“Biology”} \Rightarrow$
    $\exists\, s \in takes\ (t\,[ID] = s\,[ID\,] \wedge$
    $s\,[course_id] = u\,[course_id]))\}$
- Without the existential quantification on student, the above query would be unsafe if the Biology department has not offered any courses.

Domain Relational Calculus

Domain Relational Calculus

- A nonprocedural query language equivalent in power to the tuple relational calculus
- Each query is an expression of the form:

$$\{ < x_1, x_2, \ldots, x_n > \mid P(x_1, x_2, \ldots, x_n)\}$$

  - $x_1, x_2, \ldots, x_n$ represent domain variables
  - P represents a formula similar to that of the predicate calculus

Example Queries

- Find the *ID, name, dept_name, salary* for instructors whose salary is greater than $80,000
  - $\{< i, n, d, s> \mid < i, n, d, s> \in instructor \wedge s > 80000\}$
- As in the previous query, but output only the *ID* attribute value
  - $\{< i> \mid < i, n, d, s> \in instructor \wedge s > 80000\}$
- Find the names of all instructors whose department is in the Watson building

$$\{< n > \mid \exists\, i, d, s\, (< i, n, d, s > \in instructor \wedge \exists\, b, a\, (< d, b, a> \in department \wedge b = \text{“Watson”}))\}$$

Example Queries

- Find the set of all courses taught in the Fall 2009 semester, or in the Spring 2010 semester, or both

$$\{\langle c\rangle \mid \exists\, a, s, y, b, r, t\ (\langle c, a, s, y, b, r, t\rangle \in section \wedge s = \text{“Fall”} \wedge y = 2009)$$
$$\vee\ \exists\, a, s, y, b, r, t\ (\langle c, a, s, y, b, r, t\rangle \in section\ ] \wedge s = \text{“Spring”} \wedge y = 2010)\}$$

This case can also be written as

$$\{\langle c\rangle \mid \exists\, a, s, y, b, r, t\ (\langle c, a, s, y, b, r, t\rangle \in section \wedge ((s = \text{“Fall”} \wedge y = 2009) \vee (s = \text{“Spring”} \wedge y = 2010))\}$$

- Find the set of all courses taught in the Fall 2009 semester, and in the Spring 2010 semester

$$\{\langle c\rangle \mid \exists\, a, s, y, b, r, t\ (\langle c, a, s, y, b, r, t\rangle \in section \wedge s = \text{“Fall”} \wedge y = 2009)$$
$$\wedge\ \exists\, a, s, y, b, r, t\ (\langle c, a, s, y, b, r, t\rangle \in section\ ] \wedge s = \text{“Spring”} \wedge y = 2010)\}$$

Safety of Expressions

The expression:

$$\{ < x_1, x_2, \ldots, x_n > \mid P(x_1, x_2, \ldots, x_n)\}$$

is safe if all of the following hold:

1. All values that appear in tuples of the expression are values from *dom* (P) (that is, the values appear either in P or in a tuple of a relation mentioned in P).
2. For every "there exists" subformula of the form $\exists\, x\, (P_1(x))$, the subformula is true if and only if there is a value of x in *dom* (P_1) such that $P_1(x)$ is true.
3. For every "for all" subformula of the form $\forall_x\, (P_1\, (x))$, the subformula is true if and only if $P_1(x)$ is true for all values x from *dom* (P_1).

Universal Quantification

- Find all students who have taken all courses offered in the Biology department
  - $\{< i > \mid \exists\, n, d, tc\, (< i, n, d, tc > \in student \;\wedge$
    $(\forall\, ci, ti, dn, cr\, (< ci, ti, dn, cr > \in course \wedge dn = \text{“Biology”}$
    $\Rightarrow \exists\, si, se, y, g\, (< i, ci, si, se, y, g > \in takes\,))\}$
  - Note that without the existential quantification on student, the above query would be unsafe if the Biology department has not offered any courses.

Datalog

End of Chapter 27

Chapter 28: Advanced Relational Database Design

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Outline

- Reasoning with MVDs
- Higher normal forms
  - Join dependencies and PJNF
  - DKNF

Theory of Multivalued Dependencies

- Let *D* denote a set of functional and multivalued dependencies. The closure D^+ of *D* is the set of all functional and multivalued dependencies logically implied by *D*.
- Sound and complete inference rules for functional and multivalued dependencies:

1. **Reflexivity rule**. If α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.
2. **Augmentation rule**. If $\alpha \rightarrow \beta$ holds and γ is a set of attributes, then $\gamma\ \alpha \rightarrow \gamma\ \beta$ holds.
3. **Transitivity rule**. If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.

Theory of Multivalued Dependencies (Cont.)

4. **Complementation rule.** If $\alpha \rightarrow\rightarrow \beta$ holds, then $\alpha \rightarrow\rightarrow R - \beta - \alpha$ holds.
5. **Multivalued augmentation rule.** If $\alpha \rightarrow\rightarrow \beta$ holds and $\gamma \subseteq R$ and $\delta \subseteq \gamma$, then $\gamma\ \alpha \rightarrow\rightarrow \delta\ \beta$ holds.
6. **Multivalued transitivity rule**. If $\alpha \rightarrow\rightarrow \beta$ holds and $\beta \rightarrow\rightarrow \gamma$ holds, then $\alpha \rightarrow\rightarrow \gamma - \beta$ holds.
7. **Replication rule.** If $\alpha \rightarrow \beta$ holds, then $\alpha \rightarrow\rightarrow \beta$.
8. **Coalescence rule.** If $\alpha \rightarrow\rightarrow \beta$ holds and $\gamma \subseteq \beta$ and there is a δ such that $\delta \subseteq R$ and $\delta \cap \beta = \varnothing$ and $\delta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ holds.

Simplification of the Computation of D^+

- We can simplify the computation of the closure of *D* by using the following rules (proved using rules 1-8).
  - **Multivalued union rule.** If $\alpha \rightarrow\rightarrow \beta$ holds and $\alpha \rightarrow\rightarrow \gamma$ holds, then $\alpha \rightarrow\rightarrow \beta\gamma$ holds.
  - **Intersection rule.** If $\alpha \rightarrow\rightarrow \beta$ holds and $\alpha \rightarrow\rightarrow \gamma$ holds, then $\alpha \rightarrow\rightarrow \beta \cap \gamma$ holds.
  - **Difference rule.** If If $\alpha \rightarrow\rightarrow \beta$ holds and $\alpha \rightarrow\rightarrow \gamma$ holds, then $\alpha \rightarrow\rightarrow \beta - \gamma$ holds and $\alpha \rightarrow\rightarrow \gamma - \beta$ holds.

Example

- $R = (A, B, C, G, H, I)$
  $D = \{A \rightarrow\rightarrow B$
  $B \rightarrow\rightarrow HI$
  $CG \rightarrow H\}$

- Some members of D^+:
  - $A \rightarrow\rightarrow CGHI.$
    Since $A \rightarrow\rightarrow B$, the complementation rule (4) implies that $A \rightarrow\rightarrow R - B - A.$
    Since $R - B - A = CGHI$, so $A \rightarrow\rightarrow CGHI.$
  - $A \rightarrow\rightarrow HI.$
    Since $A \rightarrow\rightarrow B$ and $B \rightarrow\rightarrow HI$, the multivalued transitivity rule (6) implies that $B \rightarrow\rightarrow HI - B.$
    Since $HI - B = HI$, $A \rightarrow\rightarrow HI.$

Example (Cont.)

- Some members of D^+ (cont.):
  - $B \rightarrow H$.
    Apply the coalescence rule (8); $B \rightarrow\rightarrow HI$ holds.
    Since $H \subseteq HI$ and $CG \rightarrow H$ and $CG \cap HI = \emptyset$, the coalescence rule is satisfied with α being B, β being HI, δ being CG, and γ being H. We conclude that $B \rightarrow H$.
  - $A \rightarrow\rightarrow CG$.
    $A \rightarrow\rightarrow CGHI$ and $A \rightarrow\rightarrow HI$.
    By the difference rule, $A \rightarrow\rightarrow CGHI - HI$.
    Since $CGHI - HI = CG$, $A \rightarrow\rightarrow CG$.

Normalization Using Join Dependencies

- Join dependencies constrain the set of legal relations over a schema R to those relations for which a given decomposition is a lossless-join decomposition.
- Let R be a relation schema and R_1 , R_2 ,..., R_n be a decomposition of R. If $R = R_1 \cup R_2 \cup \ldots \cup R_n$, we say that a relation $r(R)$ satisfies the *join dependency* $*(R_1 , R_2 ,..., R_n)$ if:

  $$r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r) \bowtie \ldots\ldots \bowtie \Pi_{Rn}(r)$$

  A join dependency is *trivia*l if one of the R_i is R itself.
- A join dependency $*(R_1, R_2)$ is equivalent to the multivalued dependency $R_1 \cap R_2 \twoheadrightarrow R_2$. Conversely, $\alpha \twoheadrightarrow \beta$ is equivalent to $*(\alpha \cup (R - \beta), \alpha \cup \beta)$
- However, there are join dependencies that are not equivalent to any multivalued dependency.

Project-Join Normal Form (PJNF)

- A relation schema R is in PJNF with respect to a set D of functional, multivalued, and join dependencies if for all join dependencies in D^+ of the form

  $*(R_1, R_2, ..., R_n)$ where each $R_i \subseteq R$

  and $R = R_1 \cup R_2 \cup ... \cup R_n$

  at least one of the following holds:

  - $*(R_1, R_2, ..., R_n)$ is a trivial join dependency.
  - Every R_i is a superkey for R.
- Since every multivalued dependency is also a join dependency, every PJNF schema is also in 4NF.

Example

- Consider *Loan-info-schema* = (*branch-name, customer-name, loan-number, amount*).
- Each loan has one or more customers, is in one or more branches and has a loan amount; these relationships are independent, hence we have the join dependency
- *(=(*loan-number, branch-name*), (*loan-number, customer-name*), (*loan-number, amount*))
- *Loan-info-schema* is not in PJNF with respect to the set of dependencies containing the above join dependency. To put *Loan-info-schema* into PJNF, we must decompose it into the three schemas specified by the join dependency:
  - (*loan-number, branch-name*)
  - (*loan-number, customer-name*)
  - (*loan-number, amount*)

Domain-Key Normal Form (DKNY)

- **Domain declaration**. Let A be an attribute, and let **dom** be a set of values. The domain declaration $A \subseteq$ **dom** requires that the A value of all tuples be values in **dom**.
- **Key declaration**. Let R be a relation schema with $K \subseteq R$. The key declaration **key** (K) requires that K be a superkey for schema R ($K \rightarrow R$). All key declarations are functional dependencies but not all functional dependencies are key declarations.
- **General constraint**. A general constraint is a predicate on the set of all relations on a given schema.
- Let **D** be a set of domain constraints and let **K** be a set of key constraints for a relation schema R. Let **G** denote the general constraints for R. Schema R is in DKNF if **D** $\cup$ **K** logically imply **G**.

Example

- Accounts whose *account-number* begins with the digit 9 are special high-interest accounts with a minimum balance of 2500.
- General constraint: ``If the first digit of *t* [*account-number*] is 9, then *t* [*balance*] $\geq$ 2500."
- DKNF design:

  Regular-acct-schema = (*branch-name, account-number, balance*)

  Special-acct-schema = (*branch-name, account-number, balance*)
- Domain constraints for {*Special-acct-schema*} require that for each account:
  - The account number begins with 9.
  - The balance is greater than 2500.

DKNF rephrasing of PJNF Definition

- Let $R = (A_1, A_2, ..., A_n)$ be a relation schema. Let dom(A_i) denote the domain of attribute A_i, and let all these domains be infinite. Then all domain constraints **D** are of the form $A_i \subseteq$ **dom** (A_i).
- Let the general constraints be a set **G** of functional, multivalued, or join dependencies. If F is the set of functional dependencies in **G**, let the set **K** of key constraints be those nontrivial functional dependencies in F^+ of the form $\alpha \rightarrow R$.
- Schema R is in PJNF if and only if it is in DKNF with respect to **D**, **K**, and **G**.

End of Chapter 28

Chapter 29: Object-Based Databases

Outline

- Complex Data Types and Object Orientation
- Structured Data Types and Inheritance in SQL
- Table Inheritance
- Array and Multiset Types in SQL
- Object Identity and Reference Types in SQL
- Implementing O-R Features
- Persistent Programming Languages
- Comparison of Object-Oriented and Object-Relational Databases

Object-Relational Data Models

- Extend the relational data model by including object orientation and constructs to deal with added data types.
- Allow attributes of tuples to have complex types, including non-atomic values such as nested relations.
- Preserve relational foundations, in particular the declarative access to data, while extending modeling power.
- Upward compatibility with existing relational languages.

Complex Data Types

- Motivation:
  - Permit non-atomic domains (atomic ≡ indivisible)
  - Example of non-atomic domain: set of integers,or set of tuples
  - Allows more intuitive modeling for applications with complex data
- Intuitive definition:
  - allow relations whenever we allow atomic (scalar) values — relations within relations
  - Retains mathematical foundation of relational model
  - Violates first normal form.

Example of a Nested Relation

- Example: library information system
- Each book has
  - title,
  - a list (array) of authors,
  - Publisher, with subfields *name* and *branch*, and
  - a set of keywords
- Non-1NF relation *books*

title	*author_array*	*publisher*	*keyword_set*
		(*name*, *branch*)	
Compilers	[Smith, Jones]	(McGraw-Hill, NewYork)	{parsing, analysis}
Networks	[Jones, Frick]	(Oxford, London)	{Internet, Web}

4NF Decomposition of Nested Relation

- Suppose for simplicity that title uniquely identifies a book
  - In real world ISBN is a unique identifier
- Decompose *books* into 4NF using the schemas:
  - (*title, author, position*)
  - (*title, keyword*)
  - (*title, pub-name, pub-branch*)
- 4NF design requires users to include joins in their queries.

title	*author*	*position*
Compilers	Smith	1
Compilers	Jones	2
Networks	Jones	1
Networks	Frick	2

authors

title	*keyword*
Compilers	parsing
Compilers	analysis
Networks	Internet
Networks	Web

keywords

title	*pub_name*	*pub_branch*
Compilers	McGraw-Hill	New York
Networks	Oxford	London

books4

Complex Types and SQL

- Extensions introduced in SQL:1999 to support complex types:
  - Collection and large object types
    - Nested relations are an example of collection types
  - Structured types
    - Nested record structures like composite attributes
  - Inheritance
  - Object orientation
    - Including object identifiers and references
- Not fully implemented in any database system currently
  - But some features are present in each of the major commercial database systems
    - Read the manual of your database system to see what it supports

Structured Types and Inheritance in SQL

- **Structured types** (a.k.a. **user-defined types**) can be declared and used in SQL

```
create type Name as
    (firstname      varchar(20),
     lastname       varchar(20))
     final

create type Address as
    (street         varchar(20),
     city           varchar(20),
     zipcode        varchar(20))
     not final
```

  - Note: **final** and **not final** indicate whether subtypes can be created

- Structured types can be used to create tables with composite attributes

```
create table person (
    name        Name,
    address     Address,
    dateOfBirth date)
```

- Dot notation used to reference components: *name.firstname*

Structured Types (cont.)

- **User-defined row types**

```
create type PersonType as (
    name Name,
    address Address,
    dateOfBirth date)
    not final
```

- Can then create a table whose rows are a user-defined type

```
create table customer of CustomerType
```

- Alternative using **unnamed row types**.

```
create table person_r(
        name      row(firstname  varchar(20),
                      lastname   varchar(20)),
        address   row(street     varchar(20),
                      city       varchar(20),
                      zipcode    varchar(20)),
        dateOfBirth date)
```

Methods

- Can add a method declaration with a structured type.

  method *ageOnDate* (*onDate* **date**)

  returns interval year

- Method body is given separately.

  create instance method *ageOnDate* (*onDate* **date**)

  returns interval year

  for *CustomerType*

  begin

  return *onDate* - **self**.*dateOfBirth*;

  end

- We can now find the age of each customer:

  select *name.lastname, ageOnDate* (**current_date**)

  from *customer*

Constructor Functions

- **Constructor functions** are used to create values of structured types
- E.g.
  create function *Name*(*firstname* **varchar**(20), *lastname* **varchar**(20))
  returns *Name*
  begin
  set self.*firstname* = *firstname;*
  set self.*lastname* = *lastname;*
  end
- To create a value of type *Name,* we use
  new *Name*(‘John’, ‘Smith’)
- Normally used in insert statements
  insert into *Person* **values**
  (**new** *Name*(‘John’, ‘Smith),
  new *Address*(’20 Main St’, ‘New York’, ‘11001’),
  date ‘1960-8-22’);

Type Inheritance

- Suppose that we have the following type definition for people:

```
create type Person
    (name varchar(20),
    address varchar(20))
```

- Using inheritance to define the student and teacher types

```
create type Student
  under Person
  (degree       varchar(20),
   department   varchar(20))
create type Teacher
  under Person
  (salary       integer,
   department   varchar(20))
```

- Subtypes can redefine methods by using **overriding method** in place of **method** in the method declaration

Multiple Type Inheritance

- SQL:1999 and SQL:2003 do not support multiple inheritance
- If our type system supports multiple inheritance, we can define a type for teaching assistant as follows:

  create type *Teaching Assistant*
  under *Student, Teacher*

- To avoid a conflict between the two occurrences of *department* we can rename them

  create type *Teaching Assistant*
  under
  Student **with** (*department* **as** *student_dept*),
  Teacher **with** (*department* **as** *teacher_dept*)

- Each value must have a **most-specific type**

Table Inheritance

- Tables created from subtypes can further be specified as **subtables**
- E.g. **create table** *people* **of** *Person;*
  create table *students* **of** *Student* **under** *people;*
  create table *teachers* **of** *Teacher* **under** *people;*
- Tuples added to a subtable are automatically visible to queries on the supertable
  - E.g. query on *people* also sees *students* and *teachers*.
  - Similarly updates/deletes on *people* also result in updates/deletes on subtables
  - To override this behaviour, use "**only** *people"* in query
- Conceptually, multiple inheritance is possible with tables
  - e.g. *teaching_assistants* under *students* and *teachers*
  - *But is not supported in SQL currently*
    - So we cannot create a person (tuple in *people*) who is both a student and a teacher

Consistency Requirements for Subtables

- Consistency requirements on subtables and supertables.
  - Each tuple of the supertable (e.g. *people)* can correspond to at most one tuple in each of the subtables (e.g. *students* and *teachers)*
  - Additional constraint in SQL:1999:

    All tuples corresponding to each other (that is, with the same values for inherited attributes) must be derived from one tuple (inserted into one table).
    - That is, each entity must have a most specific type
    - We cannot have a tuple in *people* corresponding to a tuple each in *students* and *teachers*

Array and Multiset Types in SQL

- Example of array and multiset declaration:

```
create type Publisher as
    (name          varchar(20),
     branch        varchar(20));
create type Book as
    (title         varchar(20),
     author_array  varchar(20) array [10],
     pub_date      date,
     publisher     Publisher,
     keyword-set   varchar(20) multiset);

create table books of Book;
```

Creation of Collection Values

- Array construction

  array [‘Silberschatz’,`Korth’,`Sudarshan’]

- Multisets

  multiset [‘computer’, ‘database’, ‘SQL’]

- To create a tuple of the type defined by the books relation:

  (‘Compilers’, **array**[`Smith’,`Jones’],
  new *Publisher* (`McGraw-Hill’,`New York’),
  multiset [`parsing’,`analysis’])

- To insert the preceding tuple into the relation books

  insert into *books*
  values
  (‘Compilers’, **array**[`Smith’,`Jones’],
  new *Publisher* (`McGraw-Hill’,`New York’),
  multiset [`parsing’,`analysis’]);

Querying Collection-Valued Attributes

- To find all books that have the word “database” as a keyword,

  select *title*
  from *books*
  where ‘database’ **in** (**unnest**(*keyword-set*))

- We can access individual elements of an array by using indices
  - E.g.: If we know that a particular book has three authors, we could write:

    select *author_array*[1], *author_array*[2], *author_array*[3]
    from *books*
    where *title* = `Database System Concepts’

- To get a relation containing pairs of the form “title, author_name” for each book and each author of the book

  select *B.title, A.author*
  from *books* **as** *B*, **unnest** (*B.author_array*) **as** *A* (*author*)

- To retain ordering information we add a **with ordinality** clause

  select *B.title, A.author, A.position*
  from *books* **as** *B*, **unnest** (*B.author_array*) **with ordinality as**
  A (*author, position*)

Unnesting

- The transformation of a nested relation into a form with fewer (or no) relation-valued attributes us called **unnesting**.
- E.g.

  select *title*, *A* **as** *author*, *publisher.name* **as** *pub_name*,
  publisher.branch **as** *pub_branch*, *K.keyword*
  from *books* **as** *B*, **unnest**(*B.author_array*) **as** *A* (*author*),
  unnest (*B.keyword_set*) **as** *K* (*keyword*)
- Result relation *flat_books*

title	*author*	*pub_name*	*pub_branch*	*keyword*
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

Nesting

- **Nesting** is the opposite of unnesting, creating a collection-valued attribute
- Nesting can be done in a manner similar to aggregation, but using the function **colect**() in place of an aggregation operation, to create a multiset
- To nest the *flat_books* relation on the attribute *keyword*:

  select *title*, *author*, *Publisher* (*pub_name, pub_branch*) **as** *publisher*,
  collect (*keyword*) **as** *keyword_set*
  from *flat_books*
  groupby *title, author, publisher*
- To nest on both authors and keywords:

  select *title*, **collect** (*author*) **as** *author_set*,
  Publisher (*pub_name, pub_branch*) **as** *publisher*,
  collect (*keyword*) **as** *keyword_set*
  from *flat_books*
  group by *title*, *publisher*

Nesting (Cont.)

- Another approach to creating nested relations is to use subqueries in the **select** clause, starting from the 4NF relation *books4*

```
select title,
        array (select author
                from authors as A
                where A.title = B.title
                        order by A.position) as author_array,
        Publisher (pub-name, pub-branch) as publisher,
        multiset (select keyword
                from keywords as K
                where K.title = B.title) as keyword_set
from books4 as B
```

Object-Identity and Reference Types

- Define a type *Department* with a field *name* and a field *head* which is a reference to the type *Person,* with table *people* as scope:

  create type *Department* (
  name **varchar** (20),
  head **ref** (*Person*) **scope** *people*)

- We can then create a table *departments* as follows

  create table *departments* **of** *Department*

- We can omit the declaration **scope** people from the type declaration and instead make an addition to the **create table** statement:

  create table *departments* **of** *Department*
  (*head* **with options scope** *people*)

- Referenced table must have an attribute that stores the identifier, called the **self-referential attribute**

  create table *people* **of** *Person*
  ref is *person_id* **system generated;**

Initializing Reference-Typed Values

- To create a tuple with a reference value, we can first create the tuple with a null reference and then set the reference separately:

```
insert into departments
      values (`CS', null)
update departments
    set head = (select p.person_id
                  from people as p
                    where name = `John')
    where name = `CS'
```

User Generated Identifiers

- The type of the object-identifier must be specified as part of the type definition of the referenced table, and
- The table definition must specify that the reference is user generated

```
create type Person
    (name varchar(20)
     address varchar(20))
   ref using varchar(20)
create table people of Person
  ref is person_id user generated
```

- When creating a tuple, we must provide a unique value for the identifier:

```
insert into people (person_id, name, address ) values
    ('01284567', 'John', `23 Coyote Run')
```

- We can then use the identifier value when inserting a tuple into *departments*
  - Avoids need for a separate query to retrieve the identifier:

```
insert into departments
values(`CS', `02184567')
```

User Generated Identifiers

- Can use an existing primary key value as the identifier:

  create type *Person*
  (*name* **varchar** (20) **primary key**,
  address **varchar**(20))
  ref from (*name*)
  create table *people* **of** *Person*
  ref is *person_id* **derived**

- When inserting a tuple for *departments*, we can then use

  insert into *departments*
  values(`CS',`John')

Path Expressions

- Find the names and addresses of the heads of all departments:

  select *head –>name*, *head –>address*
  from *departments*

- An expression such as “head–>name” is called a **path expression**
- Path expressions help avoid explicit joins
  - If department head were not a reference, a join of *departments* with *people* would be required to get at the address
  - Makes expressing the query much easier for the user

Implementing O-R Features

- Similar to how E-R features are mapped onto relation schemas
- Subtable implementation
  - Each table stores primary key and those attributes defined in that table

  or,

  - Each table stores both locally defined and inherited attributes

Persistent Programming Languages

- Languages extended with constructs to handle persistent data
- Programmer can manipulate persistent data directly
  - no need to fetch it into memory and store it back to disk (unlike embedded SQL)
- Persistent objects:
  - **Persistence by class** - explicit declaration of persistence
  - **Persistence by creation** - special syntax to create persistent objects
  - **Persistence by marking** - make objects persistent after creation
  - **Persistence by reachability** - object is persistent if it is declared explicitly to be so or is reachable from a persistent object

Object Identity and Pointers

- Degrees of permanence of object identity
  - **Intraprocedure**: only during execution of a single procedure
  - **Intraprogram**: only during execution of a single program or query
  - **Interprogram**: across program executions, but not if data-storage format on disk changes
  - **Persistent**: interprogram, plus persistent across data reorganizations
- Persistent versions of C++ and Java have been implemented
  - C++
    - ODMG C++
    - ObjectStore
  - Java
    - Java Database Objects (JDO)

Persistent C++ Systems

- Extensions of C++ language to support persistent storage of objects
- Several proposals, ODMG standard proposed, but not much action of late
  - **persistent pointers**: e.g. d_Ref<T>
  - **creation of persistent objects**: e.g. new (db) T()
  - **Class extents**: access to all persistent objects of a particular class
  - **Relationships:** Represented by pointers stored in related objects
    - Issue: consistency of pointers
    - Solution: extension to type system to automatically maintain back-references
  - **Iterator interface**
  - **Transactions**
  - **Updates:** mark_modified() function to tell system that a persistent object that was fetched into memory has been updated
  - **Query language**

Persistent Java Systems

- Standard for adding persistence to Java : **Java Database Objects (JDO)**
  - Persistence by reachability
  - Byte code enhancement
    - Classes separately declared as persistent
    - Byte code modifier program modifies class byte code to support persistence
      - E.g. Fetch object on demand
      - Mark modified objects to be written back to database
  - Database mapping
    - Allows objects to be stored in a relational database
  - Class extents
  - Single reference type
    - no difference between in-memory pointer and persistent pointer
    - Implementation technique based on **hollow objects** (a.k.a. **pointer swizzling**)

Object-Relational Mapping

- **Object-Relational Mapping (ORM)** systems built on top of traditional relational databases
- Implementor provides a mapping from objects to relations
  - Objects are purely transient, no permanent object identity
- Objects can be retried from database
  - System uses mapping to fetch relevant data from relations and construct objects
  - Updated objects are stored back in database by generating corresponding update/insert/delete statements
- The **Hibernate** ORM system is widely used
  - described in Section 9.4.2
  - Provides API to start/end transactions, fetch objects, etc
  - Provides query language operating direcly on object model
    - queries translated to SQL
- Limitations: overheads, especially for bulk updates

Comparison of O-O and O-R Databases

- **Relational systems**
  - simple data types, powerful query languages, high protection.
- **Persistent-programming-language-based OODBs**
  - complex data types, integration with programming language, high performance.
- **Object-relational systems**
  - complex data types, powerful query languages, high protection.
- **Object-relational mapping systems**
  - complex data types integrated with programming language, but built as a layer on top of a relational database system
- Note: Many real systems blur these boundaries
  - E.g. persistent programming language built as a wrapper on a relational database offers first two benefits, but may have poor performance.

End of Chapter 29

Chapter 30: XML

Outline

- Structure of XML Data
- XML Document Schema
- Querying and Transformation
- Application Program Interfaces to XML
- Storage of XML Data
- XML Applications

Introduction

- XML: Extensible Markup Language
- Defined by the WWW Consortium (W3C)
- Derived from SGML (Standard Generalized Markup Language), but simpler to use than SGML
- Documents have tags giving extra information about sections of the document
  - E.g. <title> XML </title> <slide> Introduction …</slide>
- **Extensible**, unlike HTML
  - Users can add new tags, and *separately* specify how the tag should be handled for display

XML Introduction (Cont.)

- The ability to specify new tags, and to create nested tag structures make XML a great way to exchange **data**, not just documents.
  - Much of the use of XML has been in data exchange applications, not as a replacement for HTML
- Tags make data (relatively) self-documenting
  - E.g.

```
<university>
    <department>
      <dept_name> Comp. Sci. </dept_name>
      <building> Taylor </building>
      <budget> 100000 </budget>
    </department>
    <course>
      <course_id> CS-101 </course_id>
      <title> Intro. to Computer Science </title>
      <dept_name> Comp. Sci </dept_name>
      <credits> 4 </credits>
    </course>
</university>
```

XML: Motivation

- Data interchange is critical in today's networked world
  - Examples:
    - Banking: funds transfer
    - Order processing (especially inter-company orders)
    - Scientific data
      - Chemistry: ChemML, …
      - Genetics: BSML (Bio-Sequence Markup Language), …
  - Paper flow of information between organizations is being replaced by electronic flow of information
- Each application area has its own set of standards for representing information
- XML has become the basis for all new generation data interchange formats

XML Motivation (Cont.)

- Earlier generation formats were based on plain text with line headers indicating the meaning of fields
  - Similar in concept to email headers
  - Does not allow for nested structures, no standard “type” language
  - Tied too closely to low level document structure (lines, spaces, etc)
- Each XML based standard defines what are valid elements, using
  - XML type specification languages to specify the syntax
    - DTD (Document Type Descriptors)
    - XML Schema
  - Plus textual descriptions of the semantics
- XML allows new tags to be defined as required
  - However, this may be constrained by DTDs
- A wide variety of tools is available for parsing, browsing and querying XML documents/data

Comparison with Relational Data

- Inefficient: tags, which in effect represent schema information, are repeated
- Better than relational tuples as a data-exchange format
  - Unlike relational tuples, XML data is self-documenting due to presence of tags
  - Non-rigid format: tags can be added
  - Allows nested structures
  - Wide acceptance, not only in database systems, but also in browsers, tools, and applications

Structure of XML Data

- **Tag**: label for a section of data
- **Element**: section of data beginning with <*tagname*> and ending with matching </*tagname*>
- Elements must be properly nested
  - Proper nesting
    - <course> … <title> …. </title> </course>
  - Improper nesting
    - <course> … <title> …. </course> </title>
  - Formally: every start tag must have a unique matching end tag, that is in the context of the same parent element.
- Every document must have a single top-level element

Example of Nested Elements

```
<purchase_order>
    <identifier> P-101 </identifier>
    <purchaser>  …. </purchaser>
    <itemlist>
        <item>
            <identifier> RS1 </identifier>
            <description> Atom powered rocket sled </description>
            <quantity> 2 </quantity>
            <price> 199.95 </price>
        </item>
        <item>
            <identifier> SG2 </identifier>
            <description> Superb glue </description>
            <quantity> 1 </quantity>
            <unit-of-measure> liter </unit-of-measure>
            <price> 29.95 </price>
        </item>
    </itemlist>
</purchase_order>
```

Motivation for Nesting

- Nesting of data is useful in data transfer
  - Example: elements representing *item* nested within an *itemlist* element
- Nesting is not supported, or discouraged, in relational databases
  - With multiple orders, customer name and address are stored redundantly
  - normalization replaces nested structures in each order by foreign key into table storing customer name and address information
  - Nesting is supported in object-relational databases
- But nesting is appropriate when transferring data
  - External application does not have direct access to data referenced by a foreign key

Structure of XML Data (Cont.)

- Mixture of text with sub-elements is legal in XML.
  - Example:

```
<course>
    This course is being offered for the first time in 2009.
    <course id> BIO-399 </course id>
    <title> Computational Biology </title>
    <dept name> Biology </dept name>
    <credits> 3 </credits>
</course>
```

  - Useful for document markup, but discouraged for data representation

Attributes

- Elements can have **attributes**

```
<course course_id= “CS-101”>
    <title> Intro. to Computer Science</title>
    <dept name> Comp. Sci. </dept name>
    <credits> 4 </credits>
</course>
```

- Attributes are specified by *name=value* pairs inside the starting tag of an element
- An element may have several attributes, but each attribute name can only occur once

```
<course  course_id = “CS-101”  credits=“4”>
```

Attributes vs. Subelements

- Distinction between subelement and attribute
  - In the context of documents, attributes are part of markup, while subelement contents are part of the basic document contents
  - In the context of data representation, the difference is unclear and may be confusing
    - Same information can be represented in two ways
      - <course course_id= “CS-101”> … </course>
      - <course>
        <course_id>CS-101</course_id> …
        </course>
  - Suggestion: use attributes for identifiers of elements, and use subelements for contents

Namespaces

- XML data has to be exchanged between organizations
- Same tag name may have different meaning in different organizations, causing confusion on exchanged documents
- Specifying a unique string as an element name avoids confusion
- Better solution: use unique-name:element-name
- Avoid using long unique names all over document by using XML Namespaces

```
<university xmlns:yale="http://www.yale.edu">
    …
    <yale:course>
        <yale:course_id> CS-101 </yale:course_id>
        <yale:title> Intro. to Computer Science</yale:title>
        <yale:dept_name> Comp. Sci. </yale:dept_name>
        <yale:credits> 4 </yale:credits>
    </yale:course>
    …
</university>
```

More on XML Syntax

- Elements without subelements or text content can be abbreviated by ending the start tag with a /> and deleting the end tag
  - <course course_id="CS-101" Title="Intro. To Computer Science" dept_name = "Comp. Sci." credits="4" />
- To store string data that may contain tags, without the tags being interpreted as subelements, use CDATA as below
  - <![CDATA[<course> … </course>]]>

  Here, <course> and </course> are treated as just strings

  CDATA stands for "character data"

XML Document Schema

- Database schemas constrain what information can be stored, and the data types of stored values
- XML documents are not required to have an associated schema
- However, schemas are very important for XML data exchange
  - Otherwise, a site cannot automatically interpret data received from another site
- Two mechanisms for specifying XML schema
  - **Document Type Definition (DTD)**
    - Widely used
  - **XML Schema**
    - Newer, increasing use

Document Type Definition (DTD)

- The type of an XML document can be specified using a DTD
- DTD constraints structure of XML data
  - What elements can occur
  - What attributes can/must an element have
  - What subelements can/must occur inside each element, and how many times.
- DTD does not constrain data types
  - All values represented as strings in XML
- DTD syntax
  - <!ELEMENT element (subelements-specification) >
  - <!ATTLIST element (attributes) >

Element Specification in DTD

- Subelements can be specified as
  - names of elements, or
  - #PCDATA (parsed character data), i.e., character strings
  - EMPTY (no subelements) or ANY (anything can be a subelement)
- Example

```
<! ELEMENT department (dept_name  building, budget)>
<! ELEMENT dept_name (#PCDATA)>
<! ELEMENT budget (#PCDATA)>
```

- Subelement specification may have regular expressions

```
<!ELEMENT university ( ( department | course | instructor | teaches )+)>
```

  - Notation:
    - "|" - alternatives
    - "+" - 1 or more occurrences
    - "*" - 0 or more occurrences

University DTD

```
<!DOCTYPE  university [
    <!ELEMENT university ( (department|course|instructor|teaches)+)>
    <!ELEMENT department ( dept name, building, budget)>
    <!ELEMENT course ( course id, title, dept name, credits)>
    <!ELEMENT instructor (IID, name, dept name, salary)>
    <!ELEMENT teaches (IID, course id)>
    <!ELEMENT dept name( #PCDATA )>
    <!ELEMENT building( #PCDATA )>
    <!ELEMENT budget( #PCDATA )>
    <!ELEMENT course id ( #PCDATA )>
    <!ELEMENT title ( #PCDATA )>
    <!ELEMENT credits( #PCDATA )>
    <!ELEMENT IID( #PCDATA )>
    <!ELEMENT name( #PCDATA )>
    <!ELEMENT salary( #PCDATA )>

]>
```

Attribute Specification in DTD

- Attribute specification : for each attribute
  - Name
  - Type of attribute
    - CDATA
    - ID (identifier) or IDREF (ID reference) or IDREFS (multiple IDREFs)
      - more on this later
  - Whether
    - mandatory (#REQUIRED)
    - has a default value (value),
    - or neither (#IMPLIED)
- Examples
  - <!ATTLIST course course_id CDATA #REQUIRED>, or
  - <!ATTLIST course

```
course_id     ID        #REQUIRED
dept_name     IDREF     #REQUIRED
instructors   IDREFS    #IMPLIED   >
```

IDs and IDREFs

- An element can have at most one attribute of type ID
- The ID attribute value of each element in an XML document must be distinct
  - Thus the ID attribute value is an object identifier
- An attribute of type IDREF must contain the ID value of an element in the same document
- An attribute of type IDREFS contains a set of (0 or more) ID values. Each ID value must contain the ID value of an element in the same document

University DTD with Attributes

- University DTD with ID and IDREF attribute types.

```
<!DOCTYPE university-3 [
    <!ELEMENT university ( (department|course|instructor)+)>
    <!ELEMENT department ( building, budget )>
    <!ATTLIST department
        dept_name ID #REQUIRED >
    <!ELEMENT course (title, credits )>
    <!ATTLIST course
        course_id ID #REQUIRED
        dept_name IDREF #REQUIRED
        instructors IDREFS #IMPLIED >
    <!ELEMENT instructor ( name, salary )>
    <!ATTLIST instructor
        IID ID #REQUIRED
        dept_name IDREF #REQUIRED >
    · · · declarations for title, credits, building,
        budget, name and salary · · ·
]>
```

XML data with ID and IDREF attributes

```
<university-3>
    <department dept name="Comp. Sci.">
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department dept name="Biology">
        <building> Watson </building>
        <budget> 90000 </budget>
    </department>
    <course course id="CS-101" dept name="Comp. Sci"
            instructors="10101 83821">
        <title> Intro. to Computer Science </title>
        <credits> 4 </credits>
    </course>
    ….
    <instructor IID="10101" dept name="Comp. Sci.">
        <name> Srinivasan </name>
        <salary> 65000 </salary>
    </instructor>
    ….
</university-3>
```

Limitations of DTDs

- No typing of text elements and attributes
  - All values are strings, no integers, reals, etc.
- Difficult to specify unordered sets of subelements
  - Order is usually irrelevant in databases (unlike in the document-layout environment from which XML evolved)
  - (A | B)* allows specification of an unordered set, but
    - Cannot ensure that each of A and B occurs only once
- IDs and IDREFs are untyped
  - The *instructors* attribute of an course may contain a reference to another course, which is meaningless
    - *instructors* attribute should ideally be constrained to refer to instructor elements

XML Schema

- XML Schema is a more sophisticated schema language which addresses the drawbacks of DTDs. Supports
  - Typing of values
    - E.g. integer, string, etc
    - Also, constraints on min/max values
  - User-defined, comlex types
  - Many more features, including
    - uniqueness and foreign key constraints, inheritance
- XML Schema is itself specified in XML syntax, unlike DTDs
  - More-standard representation, but verbose
- XML Scheme is integrated with namespaces
- BUT: XML Schema is significantly more complicated than DTDs.

XML Schema Version of Univ. DTD

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="university" type="universityType" />
<xs:element name="department">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="dept name" type="xs:string"/>
            <xs:element name="building" type="xs:string"/>
            <xs:element name="budget" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
….
<xs:element name="instructor">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="IID" type="xs:string"/>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="dept name" type="xs:string"/>
            <xs:element name="salary" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

… Contd.

XML Schema Version of Univ. DTD (Cont.)

```
….
<xs:complexType name="UniversityType">
   <xs:sequence>
      <xs:element ref="department" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="course" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="instructor" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="teaches" minOccurs="0" maxOccurs="unbounded"/>
   </xs:sequence>
</xs:complexType>
</xs:schema>
```

- Choice of "xs:" was ours -- any other namespace prefix could be chosen
- Element "university" has type "universityType", which is defined separately
  - xs:complexType is used later to create the named complex type "UniversityType"

More features of XML Schema

- Attributes specified by xs:attribute tag:
  - <xs:attribute name = “dept_name”/>
  - adding the attribute use = “required” means value must be specified
- Key constraint: “department names form a key for department elements under the root university element:

```
<xs:key name = “deptKey”>
        <xs:selector xpath = “/university/department”/>
        <xs:field xpath = “dept_name”/>
<\xs:key>
```

- Foreign key constraint from course to department:

```
<xs:keyref name = “courseDeptFKey” refer=“deptKey”>
        <xs:selector xpath = “/university/course”/>
        <xs:field xpath = “dept_name”/>
<\xs:keyref>
```

Querying and Transforming XML Data

- Translation of information from one XML schema to another
- Querying on XML data
- Above two are closely related, and handled by the same tools
- Standard XML querying/translation languages
  - XPath
    - Simple language consisting of path expressions
  - XSLT
    - Simple language designed for translation from XML to XML and XML to HTML
  - XQuery
    - An XML query language with a rich set of features

Tree Model of XML Data

- Query and transformation languages are based on a **tree model** of XML data
- An XML document is modeled as a tree, with **nodes** corresponding to elements and attributes
  - Element nodes have child nodes, which can be attributes or subelements
  - Text in an element is modeled as a text node child of the element
  - Children of a node are ordered according to their order in the XML document
  - Element and attribute nodes (except for the root node) have a single parent, which is an element node
  - The root node has a single child, which is the root element of the document

XPath

- XPath is used to address (select) parts of documents using **path expressions**
- A path expression is a sequence of steps separated by "/"
  - Think of file names in a directory hierarchy
- Result of path expression: set of values that along with their containing elements/attributes match the specified path
- E.g. /university-3/instructor/name evaluated on the university-3 data we saw earlier returns

  <name>Srinivasan</name>
  <name>Brandt</name>

- E.g. /university-3/instructor/name/text( )

  returns the same names, but without the enclosing tags

XPath (Cont.)

- The initial "/" denotes root of the document (above the top-level tag)
- Path expressions are evaluated left to right
  - Each step operates on the set of instances produced by the previous step
- Selection predicates may follow any step in a path, in [ ]
  - E.g. /university-3/course[credits >= 4]
    - returns account elements with a balance value greater than 400
    - /university-3/course[credits] returns account elements containing a credits subelement
- Attributes are accessed using "@"
  - E.g. /university-3/course[credits >= 4]/@course_id
    - returns the course identifiers of courses with credits >= 4
  - IDREF attributes are not dereferenced automatically (more on this later)

Functions in XPath

- XPath provides several functions
  - The function count() at the end of a path counts the number of elements in the set generated by the path
    - E.g. /university-2/instructor[count(./teaches/course)> 2]
      - Returns instructors teaching more than 2 courses (on university-2 schema)
  - Also function for testing position (1, 2, ..) of node w.r.t. siblings
- Boolean connectives and and or and function not() can be used in predicates
- IDREFs can be referenced using function id()
  - id() can also be applied to sets of references such as IDREFS and even to strings containing multiple references separated by blanks
  - E.g. /university-3/course/id(@dept_name)
    - returns all department elements referred to from the dept_name attribute of course elements.

More XPath Features

- Operator "|" used to implement union
  - E.g. /university-3/course[@dept name="Comp. Sci"] | /university-3/course[@dept name="Biology"]
    - Gives union of Comp. Sci. and Biology courses
    - However, "|" cannot be nested inside other operators.
- "//" can be used to skip multiple levels of nodes
  - E.g. /university-3//name
    - finds any name element *anywhere* under the /university-3 element, regardless of the element in which it is contained.
- A step in the path can go to parents, siblings, ancestors and descendants of the nodes generated by the previous step, not just to the children
  - "//", described above, is a short from for specifying "all descendants"
  - ".." specifies the parent.
- doc(name) returns the root of a named document

XQuery

- XQuery is a general purpose query language for XML data
- Currently being standardized by the World Wide Web Consortium (W3C)
  - The textbook description is based on a January 2005 draft of the standard. The final version may differ, but major features likely to stay unchanged.
- XQuery is derived from the Quilt query language, which itself borrows from SQL, XQL and XML-QL
- XQuery uses a

  for … let … where … order by …result …

  syntax

  for ⇔ SQL **from**
  where ⇔ SQL **where**
  order by ⇔ SQL **order by**

  result ⇔ SQL **select**
  let allows temporary variables, and has no equivalent in SQL

FLWOR Syntax in XQuery

- For clause uses XPath expressions, and variable in for clause ranges over values in the set returned by XPath
- Simple FLWOR expression in XQuery
  - find all courses with credits > 3, with each result enclosed in an <course_id> .. </course_id> tag

```
for  $x in /university-3/course
let   $courseId := $x/@course_id
where $x/credits > 3
return <course_id> { $courseId } </course id>
```

  - Items in the **return** clause are XML text unless enclosed in {}, in which case they are evaluated
- Let clause not really needed in this query, and selection can be done In XPath. Query can be written as:

```
for $x in /university-3/course[credits > 3]
return <course_id> { $x/@course_id } </course_id>
```

- Alternative notation for constructing elements:

```
return element course_id { element  $x/@course_id }
```

Joins

- Joins are specified in a manner very similar to SQL

```
for $c in /university/course,
    $i in /university/instructor,
    $t in /university/teaches
where $c/course_id= $t/course id and $t/IID = $i/IID
return <course_instructor> { $c $i } </course_instructor>
```

- The same query can be expressed with the selections specified as XPath selections:

```
for $c in /university/course,
    $i in /university/instructor,
    $t in /university/teaches[ $c/course_id= $t/course_id
                                and $t/IID = $i/IID]
return <course_instructor> { $c $i } </course_instructor>
```

Nested Queries

- The following query converts data from the flat structure for university information into the nested structure used in university-1

```
<university-1>
{     for $d in /university/department
      return <department>
                 { $d/* }
                 { for $c in /university/course[dept name = $d/dept name]
                   return $c }
               </department>
}
{     for $i in /university/instructor
      return  <instructor>
                 { $i/* }
                 { for $c in /university/teaches[IID = $i/IID]
                   return $c/course id }
               </instructor>
}
</university-1>
```

- $c/* denotes all the children of the node to which $c is bound, without the enclosing top-level tag

Grouping and Aggregation

- Nested queries are used for grouping

```
for $d in /university/department
return
      <department-total-salary>
          <dept_name> { $d/dept name } </dept_name>
           <total_salary> { fn:sum(
               for $i in /university/instructor[dept_name = $d/dept_name]
               return $i/salary
             ) }
           </total_salary>
       </department-total-salary>
```

Sorting in XQuery

- The **order by** clause can be used at the end of any expression. E.g. to return instructors sorted by name

```
for $i in /university/instructor
order by $i/name
return <instructor> { $i/* } </instructor>
```

- Use **order by** $i/name **descending** to sort in descending order
- Can sort at multiple levels of nesting (sort departments by dept_name, and by courses sorted to course_id within each department)

```
<university-1> {
  for $d in /university/department
  order by $d/dept name
  return
      <department>
         { $d/* }
         { for $c in /university/course[dept name = $d/dept name]
           order by $c/course id
           return <course> { $c/* } </course> }
      </department>
} </university-1>
```

Functions and Other XQuery Features

- User defined functions with the type system of XMLSchema

```
declare function local:dept_courses($iid as xs:string)
        as element(course)*
{
    for $i in /university/instructor[IID = $iid],
        $c in /university/courses[dept_name = $i/dept name]
    return $c
}
```

- Types are optional for function parameters and return values
- The * (as in decimal*) indicates a sequence of values of that type
- Universal and existential quantification in where clause predicates
  - **some** $e **in** *path* **satisfies** *P*
  - **every** $e **in** *path* **satisfies** *P*
  - Add **and fn:exists($e)** to prevent empty $e from satisfying **every** clause
- XQuery also supports If-then-else clauses

XSLT

- A **stylesheet** stores formatting options for a document, usually separately from document
  - E.g. an HTML style sheet may specify font colors and sizes for headings, etc.
- The **XML Stylesheet Language (XSL)** was originally designed for generating HTML from XML
- XSLT is a general-purpose transformation language
  - Can translate XML to XML, and XML to HTML
- XSLT transformations are expressed using rules called **templates**
  - Templates combine selection using XPath with construction of results

Application Program Interface

- There are two standard application program interfaces to XML data:
  - **SAX** (Simple API for XML)
    - Based on parser model, user provides event handlers for parsing events
      - E.g. start of element, end of element
  - **DOM** (Document Object Model)
    - **XML** data is parsed into a tree representation
    - Variety of functions provided for traversing the DOM tree
    - E.g.: Java DOM API provides Node class with methods
      getParentNode( ), getFirstChild( ), getNextSibling( )
      getAttribute( ), getData( ) (for text node)
      getElementsByTagName( ), …
    - Also provides functions for updating DOM tree

Storage of XML Data

- XML data can be stored in
  - Non-relational data stores
    - Flat files
      - Natural for storing XML
      - But has all problems discussed in Chapter 1 (no concurrency, no recovery, …)
    - XML database
      - Database built specifically for storing XML data, supporting DOM model and declarative querying
      - Currently no commercial-grade systems
  - Relational databases
    - Data must be translated into relational form
    - Advantage: mature database systems
    - Disadvantages: overhead of translating data and queries

Storage of XML in Relational Databases

- Alternatives:
  - String Representation
  - Tree Representation
  - Map to relations

String Representation

- Store each top level element as a string field of a tuple in a relational database
  - Use a single relation to store all elements, or
  - Use a separate relation for each top-level element type
    - E.g. account, customer, depositor relations
      - Each with a string-valued attribute to store the element
- Indexing:
  - Store values of subelements/attributes to be indexed as extra fields of the relation, and build indices on these fields
    - E.g. customer_name or account_number
  - Some database systems support **function indices,** which use the result of a function as the key value.
    - The function should return the value of the required subelement/attribute

String Representation (Cont.)

- Benefits:
  - Can store any XML data even without DTD
  - As long as there are many top-level elements in a document, strings are small compared to full document
    - Allows fast access to individual elements.
- Drawback: Need to parse strings to access values inside the elements
  - Parsing is slow.

Tree Representation

- **Tree representation:** model XML data as tree and store using relations

  nodes(id, parent_id, type, label, value)

- Each element/attribute is given a unique identifier
- Type indicates element/attribute
- Label specifies the tag name of the element/name of attribute
- Value is the text value of the element/attribute
- Can add an extra attribute *position* to record ordering of children

Tree Representation (Cont.)

- Benefit: Can store any XML data, even without DTD
- Drawbacks:
  - Data is broken up into too many pieces, increasing space overheads
  - Even simple queries require a large number of joins, which can be slow

Mapping XML Data to Relations

- Relation created for each element type whose schema is known:
  - An id attribute to store a unique id for each element
  - A relation attribute corresponding to each element attribute
  - A parent_id attribute to keep track of parent element
    - As in the tree representation
    - Position information (i^{th} child) can be store too
- All subelements that occur only once can become relation attributes
  - For text-valued subelements, store the text as attribute value
  - For complex subelements, can store the id of the subelement
- Subelements that can occur multiple times represented in a separate table
  - Similar to handling of multivalued attributes when converting ER diagrams to tables

Storing XML Data in Relational Systems

- Applying above ideas to department elements in university-1 schema, with nested course elements, we get
  department(*id*, *dept_name*, *building*, *budget*)
  course(*parent id*, *course_id*, *dept_name*, *title*, *credits*)
- **Publishing**: process of converting relational data to an XML format
- **Shredding**: process of converting an XML document into a set of tuples to be inserted into one or more relations
- XML-enabled database systems support automated publishing and shredding
- Many systems offer *native storage* of XML data using the **xml** data type. Special internal data structures and indices are used for efficiency

SQL/XML

- New standard SQL extension that allows creation of nested XML output
  - Each output tuple is mapped to an XML element *row*

```
<university>
   <department>
      <row>
         <dept name> Comp. Sci. </dept name>
         <building> Taylor </building>
         <budget> 100000 </budget>
      </row>
      …. more rows if there are more output tuples …
   </department>
   … other relations ..
</university>
```

SQL Extensions

- **xmlelement** creates XML elements
- **xmlattributes** creates attributes

```
select xmlelement (name "course",
      xmlattributes (course id as course id, dept name as dept name),
      xmlelement (name "title", title),
      xmlelement (name "credits", credits))
from course
```

- Xmlagg creates a forest of XML elements

```
select xmlelement (name "department",
            dept_name,
            xmlagg (xmlforest(course_id)
                  order by course_id))
from course
group by dept_name
```

XML Applications

- Storing and exchanging data with complex structures
  - E.g. Open Document Format (ODF) format standard for storing Open Office and Office Open XML (OOXML) format standard for storing Microsoft Office documents
  - Numerous other standards for a variety of applications
    - ChemML, MathML
- Standard for data exchange for Web services
  - remote method invocation over HTTP protocol
  - More in next slide
- Data mediation
  - Common data representation format to bridge different systems

Web Services

- The Simple Object Access Protocol (SOAP) standard:
  - Invocation of procedures across applications with distinct databases
  - XML used to represent procedure input and output
- A *Web service* is a site providing a collection of SOAP procedures
  - Described using the Web Services Description Language (WSDL)
  - Directories of Web services are described using the Universal Description, Discovery, and Integration (UDDI) standard

End of Chapter 30

Chapter 31: Information Retrieval

Outline

- Relevance Ranking Using Terms
- Relevance Using Hyperlinks
- Synonyms., Homonyms, and Ontologies
- Indexing of Documents
- Measuring Retrieval Effectiveness
- Web Search Engines
- Information Retrieval and Structured Data
- Directories

Information Retrieval Systems

- **Information retrieval (IR)** systems use a simpler data model than database systems
  - Information organized as a collection of documents
  - Documents are unstructured, no schema
- Information retrieval locates relevant documents, on the basis of user input such as keywords or example documents
  - e.g., find documents containing the words “database systems”
- Can be used even on textual descriptions provided with non-textual data such as images
- Web search engines are the most familiar example of IR systems

Information Retrieval Systems (Cont.)

- Differences from database systems
  - IR systems don't deal with transactional updates (including concurrency control and recovery)
  - Database systems deal with structured data, with schemas that define the data organization
  - IR systems deal with some querying issues not generally addressed by database systems
    - Approximate searching by keywords
    - Ranking of retrieved answers by estimated degree of relevance

Keyword Search

- In **full text** retrieval, all the words in each document are considered to be keywords.
  - We use the word **term** to refer to the words in a document
- Information-retrieval systems typically allow query expressions formed using keywords and the logical connectives *and, or,* and *not*
  - *And*s are implicit, even if not explicitly specified
- Ranking of documents on the basis of estimated relevance to a query is critical
  - Relevance ranking is based on factors such as
    - Term frequency
      - Frequency of occurrence of query keyword in document
    - Inverse document frequency
      - How many documents the query keyword occurs in
        - Fewer ➔ give more importance to keyword
    - Hyperlinks to documents
      - More links to a document ➔ document is more important

Relevance Ranking Using Terms

- **TF-IDF** (Term frequency/Inverse Document frequency) ranking:
  - Let $n(d)$ = number of terms in the document d
  - $n(d, t)$ = number of occurrences of term t in the document d.
  - Relevance of a document d to a *term* t

$$TF(d, t) = log\left(1 + \frac{n(d, t)}{n(d)}\right)$$

    - The log factor is to avoid excessive weight to frequent terms
  - Relevance of document to *query* Q

$$r(d, Q) = \sum_{t \in Q} \frac{TF(d, t)}{n(t)}$$

Relevance Ranking Using Terms (Cont.)

- Most systems add to the above model
  - Words that occur in title, author list, section headings, etc. are given greater importance
  - Words whose first occurrence is late in the document are given lower importance
  - Very common words such as “a”, “an”, “the”, “it” etc. are eliminated
    - Called **stop words**
  - **Proximity**: if keywords in query occur close together in the document, the document has higher importance than if they occur far apart
- Documents are returned in decreasing order of relevance score
  - Usually only top few documents are returned, not all

Similarity Based Retrieval

- Similarity based retrieval - retrieve documents similar to a given document
  - Similarity may be defined on the basis of common words
    - E.g., find *k* terms in A with highest *TF* (*d, t*) / *n* (*t*) and use these terms to find relevance of other documents.
- **Relevance feedback**: Similarity can be used to refine answer set to keyword query
  - User selects a few relevant documents from those retrieved by keyword query, and system finds other documents similar to these
- Vector space model: define an *n*-dimensional space, where *n* is the number of words in the document set.
  - Vector for document *d* goes from origin to a point whose i^{th} coordinate is *TF* (*d,t*) / *n* (*t*)
  - The cosine of the angle between the vectors of two documents is used as a measure of their similarity.

Relevance Using Hyperlinks

- Number of documents relevant to a query can be enormous if only term frequencies are taken into account
- Using term frequencies makes “spamming” easy
  - E.g., a travel agency can add many occurrences of the words “travel” to its page to make its rank very high
- Most of the time people are looking for pages from popular sites
- Idea: use popularity of Web site (e.g., how many people visit it) to rank site pages that match given keywords
- Problem: hard to find actual popularity of site
  - Solution: next slide

Relevance Using Hyperlinks (Cont.)

- Solution: use number of hyperlinks to a site as a measure of the popularity or **prestige** of the site
  - Count only one hyperlink from each site (why? - see previous slide)
  - Popularity measure is for site, not for individual page
    - But, most hyperlinks are to root of site
    - Also, concept of “site” difficult to define since a URL prefix like cs.yale.edu contains many unrelated pages of varying popularity
- Refinements
  - When computing prestige based on links to a site, give more weight to links from sites that themselves have higher prestige
    - Definition is circular
    - Set up and solve system of simultaneous linear equations
  - Above idea is basis of the Google **PageRank** ranking mechanism

Relevance Using Hyperlinks (Cont.)

- Connections to **social networking** theories that ranked prestige of people
  - E.g., the president of the U.S.A has a high prestige since many people know him
  - Someone known by multiple prestigious people has high prestige
- Hub and authority based ranking
  - A **hub** is a page that stores links to many pages (on a topic)
  - An **authority** is a page that contains actual information on a topic
  - Each page gets a **hub prestige** based on prestige of authorities that it points to
  - Each page gets an **authority prestige** based on prestige of hubs that point to it
  - Again, prestige definitions are cyclic, and can be got by solving linear equations
  - Use authority prestige when ranking answers to a query

Synonyms and Homonyms

- **Synonyms**
  - E.g., document: “motorcycle repair”, query: “motorcycle maintenance”
    - Need to realize that “maintenance” and “repair” are synonyms
  - System can extend query as “motorcycle *and* (repair *or* maintenance)”
- Homonyms
  - E.g., “object” has different meanings as noun/verb
  - Can disambiguate meanings (to some extent) from the context
- Extending queries automatically using synonyms can be problematic
  - Need to understand intended meaning in order to infer synonyms
    - Or verify synonyms with user
  - Synonyms may have other meanings as well

Concept-Based Querying

- Approach
  - For each word, determine the **concept** it represents from context
  - Use one or more **ontologies**:
    - Hierarchical structure showing relationship between concepts
    - E.g., the ISA relationship that we saw in the E-R model
- This approach can be used to standardize terminology in a specific field
- Ontologies can link multiple languages
- Foundation of the **Semantic Web** (not covered here)

Indexing of Documents

- An inverted index maps each keyword K_i to a set of documents S_i that contain the keyword
  - Documents identified by identifiers
- Inverted index may record
  - Keyword locations within document to allow proximity based ranking
  - Counts of number of occurrences of keyword to compute TF
- **and** operation: Finds documents that contain all of K_1, K_2, ..., K_n.
  - Intersection $S_1 \cap S_2 \cap ..... \cap S_n$
- **or** operation: documents that contain at least one of K_1, K_2, …, K_n
  - union, $S_1 \cap S_2 \cap ..... \cap S_n$,.
- Each S_i is kept sorted to allow efficient intersection/union by merging
  - "**not**" can also be efficiently implemented by merging of sorted lists

Measuring Retrieval Effectiveness

- Information-retrieval systems save space by using index structures that support only approximate retrieval. May result in:
  - **false negative (false drop)** - some relevant documents may not be retrieved.
  - **false positive** - some irrelevant documents may be retrieved.
  - For many applications a good index should not permit any false drops, but may permit a few false positives.
- Relevant performance metrics:
  - **precision** - what percentage of the retrieved documents are relevant to the query.
  - **recall** - what percentage of the documents relevant to the query were retrieved.

Measuring Retrieval Effectiveness (Cont.)

- Recall vs. precision tradeoff:
  - Can increase recall by retrieving many documents (down to a low level of relevance ranking), but many irrelevant documents would be fetched, reducing precision
- Measures of retrieval effectiveness:
  - Recall as a function of number of documents fetched, or
  - **Precision as a function of recall**
    - Equivalently, as a function of number of documents fetched
  - E.g., "precision of 75% at recall of 50%, and 60% at a recall of 75%"
- Problem: which documents are actually relevant, and which are not

Web Search Engines

- **Web crawlers** are programs that locate and gather information on the Web
  - Recursively follow hyperlinks present in known documents, to find other documents
    - Starting from a *seed* set of documents
  - Fetched documents
    - Handed over to an indexing system
    - Can be discarded after indexing, or store as a *cached* copy
- Crawling the entire Web would take a very large amount of time
  - Search engines typically cover only a part of the Web, not all of it
  - Take months to perform a single crawl

Web Crawling (Cont.)

- Crawling is done by multiple processes on multiple machines, running in parallel
  - Set of links to be crawled stored in a database
  - New links found in crawled pages added to this set, to be crawled later
- Indexing process also runs on multiple machines
  - Creates a new copy of index instead of modifying old index
  - Old index is used to answer queries
  - After a crawl is “completed” new index becomes “old” index
- Multiple machines used to answer queries
  - Indices may be kept in memory
  - Queries may be routed to different machines for load balancing

Information Retrieval and Structured Data

- Information retrieval systems originally treated documents as a collection of words
- **Information extraction systems** infer structure from documents, e.g.:
  - Extraction of house attributes (size, address, number of bedrooms, etc.) from a text advertisement
  - Extraction of topic and people named from a new article
- Relations or XML structures used to store extracted data
  - System seeks connections among data to answer queries
  - **Question answering systems**

Directories

- Storing related documents together in a library facilitates browsing
  - Users can see not only requested document but also related ones.
- Browsing is facilitated by classification system that organizes logically related documents together.
- Organization is hierarchical: **classification hierarchy**

A Classification Hierarchy For A Library System

Classification DAG

- Documents can reside in multiple places in a hierarchy in an information retrieval system, since physical location is not important.
- Classification hierarchy is thus Directed Acyclic Graph (DAG)

A Classification DAG For A Library Information Retrieval System

Web Directories

- A **Web directory** is just a classification directory on Web pages
  - E.g., Yahoo! Directory, Open Directory project
  - Issues:
    - What should the directory hierarchy be?
    - Given a document, which nodes of the directory are categories relevant to the document
  - Often done manually
    - Classification of documents into a hierarchy may be done based on term similarity

End of Chapter 31