Introduction to Database Systems
(Continued)

Narain Gehani

Constraints

- A key aspect of database design is to ensure that the database contains only valid data values consistent with database semantics even in the presence of
  - insertions
  - deletions, and
  - updates.
- Constraints simplify applications the database system perform checking to ensure data validity and consistency.
- Such centralized checking is also more reliable since an application may not perform one or more of the checks

Constraints (contd.)

- A database should only contain valid and consistent data at all times with the possible exception in the middle of updates.
- Some types of data validity can be ensured using SQL constraints.
  - Examples:
    - States are identified using their 2-letter abbreviations.
    - Values must fall within a specific range.
    - Data fields are required and may not be omitted.
    - Values in a column must be unique.
  - Examples of complex database constraints:
    - 2-letter state abbreviations represent valid state abbreviations.
    - Zip codes represent valid zip codes and match the specified address.

Constraints (contd.)

- Database updates must satisfy the constraints
  - otherwise, they will be rejected
  - the transaction making the changes will be aborted.
- A database designer must specify appropriate constraints based on an understanding of the database being designed and its applications.
- Each transaction operating on valid and consistent database should leave the database in a possibly different but valid and consistent state.
- Much of the burden of ensuring database consistency falls on the application or the user modifying the database.
- Data consistency requires transactions to be correctly written.

Constraints (contd.)

- Constraints can be used to ensure syntactic validity & limited forms of semantic validity.
- Full-fledged database semantics
  - known only to applications (transactions) that modify a database
  - information does not reside in the database.
- Consider a bank database: semantics that money transfer between 2 accounts should preserve the total in the 2 accounts
  - known only to money transfer application, not to the database
- Applications (transactions) must be written correctly!
  - Each correctly written transaction operating on valid and consistent database should leave the database in a possibly different but valid and consistent state.
  - The burden of ensuring database consistency falls on the application or the user modifying the database.
- Transaction semantics are another means of ensuring data consistency, e.g.:
  - all or nothing property ensures that a transaction does not leave a database with partial updates, and
  - transactions can be prevented from reading uncommitted updates.

Constraints in SQL

- Constraints are rules to ensure that the database contains only valid values by putting conditions on what can be inserted, deleted, or updated in a table.
  - specified by the database designer
  - enforced by the database system
- No need for constraints if application updates are guaranteed to be correct.
  - Cannot rely on applications to be correct.
- SQL supports several types of constraints.
  - They can be optionally named so that they can be referenced to specify properties and when they need to be deleted.
  - Constraint violations can be checked after execution of each statement or at the end of a transaction.
  - By default, constraints are checked after each statement.

Constraints in SQL
Unique Constraint

- Specifies that the column values must be unique.
  - No two elements of such a column can have the same value (the one exception to this rule is the **`NULL`** value).
  - Inserting a non-unique value into such a column will violate the **`UNIQUE`** constraint and cause the transaction to abort.
- Syntax

  CONSTRAINT *Name* UNIQUE(*col*1, *col*2, …, *coln*)

  specifies that the columns ***col*1**, ***col*2**, …, ***coln***, must either have unique values or the **`NULL`** value.
- For individual columns, the constraint can be specified as part of the column definition using the keyword **`UNIQUE`**.
- Columns whose elements can have only with unique values are called **keys**.

Constraints in SQL
Unique Constraint Example

- Table **`Players`** stores ladder ranks of tennis players.
  - Ranking for each player must be unique
- One definition of the **`Players`** table:

```
CREATE TABLE Players(
      First VARCHAR(30),
      Last VARCHAR(30),
      Rank INT,
      CONSTRAINT Ranking UNIQUE(Rank)
) ENGINE = InnoDB;
```

- Alternate definition

```
CREATE TABLE Players(
      First VARCHAR(30),
      Last VARCHAR(30),
      Rank INT UNIQUE
) ENGINE = InnoDB;
```

Constraints in SQL

Unique Constraint Example (contd.)

- The following statements insert player information into the **Players** table:

```
INSERT INTO Players VALUES('John', 'Blair', 1);
INSERT INTO Players VALUES('Susan', 'Witzel', 2);
INSERT INTO Players VALUES('Arun', 'Neti', NULL);
INSERT INTO Players VALUES('Diya', 'Singh', 1);
```

- The last **INSERT** causes MySQL to complain:

```
ERROR 1062: Duplicate entry '1' for key 1
```

  and the statement is not executed.

- If the above statements are executed as part of a transaction:

```
START TRANSACTION;
INSERT INTO Players VALUES('John', 'Blair', 1);
       ...
INSERT INTO Players VALUES('Diya', 'Singh', 1);
```

  Then aborting the transaction explicitly with the command

```
ROLLBACK;
```

  instead of committing after the last **INSERT** will ensure that the table is not changed.

Constraints in SQL (contd.)

Primary Key

- Similar to UNIQUE except that a column element cannot have a **NULL** value.

  CONSTRAINT *Name* PRIMARY KEY(*col*1,*col*2,…,*coln*)

- For individual columns, the **PRIMARY KEY** constraint can be specified in the column definition:

```
CREATE TABLE Books (
       ISBN CHAR(10) PRIMARY KEY,
       Title VARCHAR(50),
       Price DECIMAL(5,2),
       Authors VARCHAR(50),
       Pages INT,
       PubDate YEAR(4),
       Qty INT
) ENGINE = InnoDB;
```

Constraints in SQL (contd.)

NOT NULL

- A **NOT NULL** constraint is specified in the column definition using the keywords
  ```
  NOT NULL
  ```
- Our **Books** table could have been defined as
  ```
  CREATE TABLE Books (
          ISBN CHAR(10) PRIMARY KEY,
          Title VARCHAR(50) NOT NULL,
          Price DECIMAL(5,2) NOT NULL,
          Authors VARCHAR(50) NOT NULL,
          Pages INT,
          PubDate YEAR(4),
          Qty INT NOT NULL
  ) ENGINE = InnoDB;
  ```
- <u>**Note**</u>: **PRIMARY KEY** constraint is equivalent to **UNIQUE** + **NOT NULL** constraints.

Constraints in SQL (contd.)

Referential Integrity / Foreign Key

- Referential integrity constraints (foreign keys) ensure that data in one table, the "referencing" table, has related and needed data in another table, the "foreign" or "referenced" table.
  - Each element of the column specified as foreign key must match an element of the primary key column of the specified "foreign" table.
- As mentioned earlier, a foreign key can be specified in the referencing column definition as

  REFERENCES ***table*****(*****column*****)**

  Otherwise, if multiple columns are involved, then the foreign key must be specified separately as an property:

  FOREIGN KEY(***columns*****) REFERENCES** ***table*****(*****columns*****)**

Constraints in SQL (contd.)

Referential Integrity / Foreign Key Rules

- The foreign key constraint requires observance of the following three rules:
  - *Insert Rule*: Inserting a non-null foreign key is allowed only if the foreign key matches the primary key value in the associated table as specified.
  - *Update Rule*: A primary key associated with a foreign key cannot be modified if there is a foreign key matching it.
  - *Delete Rule*: A primary key value associated with a foreign key cannot be deleted if there is a foreign key matching it.

Constraints in SQL (contd.)

Foreign Key (contd.)

- Modified **Orders** table with **CustomerId** declared as a foreign key and the **Customers** table:

```
CREATE TABLE Orders (
    OrderId INT(8) PRIMARY KEY, CustomerId INT(8),
    OrderDate DATE, ShipDate DATE,Shipping DECIMAL(5,2),
    SalesTax FLOAT,
    FOREIGN KEY (CustomerId) REFERENCES Customers(Id)
 ) ENGINE = InnoDB;

CREATE TABLE Customers(
    Id INT(8) PRIMARY KEY,
    Company VARCHAR(30),First VARCHAR(30),Last VARCHAR(30),
    Street VARCHAR(50),
    City VARCHAR(30),State2 CHAR(2),Zip CHAR(5),Tel CHAR(10)
) ENGINE = InnoDB;
```

Constraints in SQL (contd.)

Check Constraint

- Specifies a condition that must not be violated. It has the form

  CONSTRAINT *Name* CHECK(*condition*)

- As a column property it has the form

  CHECK(*condition*)

- *condition* is similar to the **WHERE** clause condition.
- Here is the **Books** table modified to include **CHECK**s:

```
CREATE TABLE Books (
       ISBN CHAR(10) PRIMARY KEY,
       Title VARCHAR(50) NOT NULL,
       Price DECIMAL(5,2) NOT NULL CHECK(Price>0.0),
       Authors VARCHAR(50) NOT NULL,
       Pages INT CHECK(PAGES > 0),
       PubDate YEAR(4),
       Qty INT NOT NULL CHECK(Qty >= 0)
) ENGINE = InnoDB;
```


When Are Constraints Executed?

- The constraint check time can be specified to be
  - immediately after the execution of each statement using the **IMMEDIATE** clause (default) or
  - deferred to the end of a transaction using the **DEFERRED** clause.
- **IMMEDIATE** vs. **DEFERRED**
  - **Con**: slows transaction execution because constraints may be checked multiple times during the execution a transaction
  - **Pro**: user will know immediately when a constraint is violated and by which statement.
- Some constraints may require the execution of more than one statement before they are satisfied.
  - Should be checked at the end of the transaction.
  - Otherwise, it will not be possible to execute the transaction because the first statement violating the constraint will cause the transaction to be aborted.

Triggers

- Triggers are actions whose execution is triggered
  - when some event occurs and
  - the specified condition is satisfied.
- Triggers have the form

  event → *condition* → *action*

  or

  condition → *event* → *action*

- When *event* occurs, the *condition* is checked and, if it is satisfied, the *action* is executed. Alternatively, the trigger condition can also be checked before the event occurs.
- In databases, an event can be the occurrence of a table operation such as
  - delete,
  - insert, or
  - update.
- The trigger condition is based on the values of the items in the table.

Triggers (contd.)

- Triggers are used for many purposes in databases, for example:
  - Automatically execute actions whenever an event occurs. This takes away responsibility from the users or applications to execute such actions – they may not even know that the action needs to be executed.
  - Implement constraints and business rules – this happens behind the scenes.
  - Maintain views – this also happens behind the scenes.
- Example
  - Automatically place a new book order for Everest Books whenever the number of copies of a book falls below the specified threshold.
- Triggers are similar to constraints in that they are associated with the occurrence of an event.
  - A constraint's actions (such as aborting a transaction) come into play when the changes being made to the database violate the constraint condition while a trigger's actions come into play when the trigger condition is satisfied.
  - Constraint actions are restricted to ensuring that the constraint condition is not violated.
  - Trigger actions, on the other hand, are not restricted in what they can do – they can be used to execute arbitrary queries.

Triggers
Advantages

- Triggers simplify writing of SQL queries
  - Code common to SQL queries is taken out and stored with the database server.
- Consider, for example, applications that update the Everest Books database to reflect items taken out of the Everest Books inventory.
- In the absence of triggers
  - it will be the responsibility of each application to generate a reorder request for a book whenever its quantity falls below a threshold amount.
  - Each application must have code to do the reordering.
- With triggers
  - the code will no longer be in the application which will make the application smaller.
  - the database administrator will not have to worry about ensuring that every application “taking” books out of the inventory also generates a reorder for the books when needed.

Triggers
Caution – Trigger Cycles

- Users needs to be careful when writing triggers.
  - Triggers can lead to the trigger “cycles.”
- For example:
  - Execution of *Trigger*1 can lead to the execution of *Trigger*2
  - which in turn can lead the execution of *Trigger*1, ad infinitum:

***Trigger*1 → *Trigger*2 → *Trigger*1 → *Trigger*2 ….**

Triggers in SQL

- An SQL trigger is a rule or action along with a condition that is associated with a table.
- The trigger fires, that is, its action is executed upon the occurrence of an event such as an **`INSERT`**, **`DELETE`**, or **`UPDATE`** operation and the satisfaction of the associated condition'
- **Trigger Creation**

```
CREATE TRIGGER tname
[BEFORE|AFTER]
[INSERT|DELETE|UPDATE [OF columns]]
ON table
WHEN (condition)
[FOR EACH ROW]
[statement |BEGIN ATOMIC statements END];
```

- The **`WHEN`** condition is identical to the condition in the **`WHERE`** clause of a **`SELECT`**.
- In case the trigger action consists of multiple statements, then they must be bracketed by **`BEGIN ATOMIC`** and **`END`**.

Triggers in SQL (contd.)

Old & New Values

- Triggers have access to both the old and new values of the tables they are associated with.
- When the trigger event is the
  - **`UPDATE`** operation, SQL allows the trigger to refer to both the old and new values of the table and its fields.
    - The keywords **`OLD`** and **`NEW`** are used for this purpose.
  - **`INSERT`** operation, the trigger has access to the values of the new rows
  - **`DELETE`** operation, the trigger has access to the values of the deleted row.

Triggers in SQL (contd.)
Old & New Values – Example

- Suppose we want to define a trigger on the table **`Books`** that
  - generates a 10 copy replenishment order for a book
  - whenever the number of copies of the books in inventory falls below 5.
- The **`Books`** table was defined earlier as

```
CREATE TABLE Books (
    ISBN CHAR(10) PRIMARY KEY,
    Title VARCHAR(50),
    Price DECIMAL(5,2),
    Authors VARCHAR (50),
    Pages INT,
    PubDate YEAR(4),
    Qty INT
) ENGINE = InnoDB;
```


Triggers in SQL (contd.)
Books Table

ISBN	Title	Price	Authors	Pages	PubDate	Qty
0929306279	Bell Labs	29.95	Gehani	269	2003	121
0929306260	Java	49.95	Sahni & Kumar	465	2003	35
0670031844	White Moghuls	34.95	Dalrymple	459	2003	78
0439357624	Born Confused	16.95	Hidier	432	2002	11

Triggers in SQL (contd.)

- In this example, books will be reordered by inserting appropriate information in the **ReOrder** table (instead of generating an invoice):

```
CREATE TABLE ReOrder(
       ISBN CHAR(10),
       ReOrderDate DATE,
       ReOrderQty INT
);
```

- Table **ReOrder** must be examined periodically to generate a "real" (physical or electronic) order for the books listed.

Triggers in SQL (contd.)

Reorder Table

ISBN	ReorderDate	ReOrderQty
0929306279	5/01/04	10
0929306260	5/01/04	10

- Table **ReOrder** must be examined periodically to generate a "real" (physical or electronic) order for the books listed.
- Of course, the rows used to generate orders must then be deleted from the **ReOrder** table.

Triggers in SQL (contd.)

Old & New Values

- Trigger **ReOrderTrigger** inserts a row in the **ReOrder** table whenever the quantity of a book in stock goes below 5:

```
CREATE TRIGGER ReOrderTrigger
    AFTER UPDATE ON Books
    REFERENCING NEW ROW AS NewRow
    FOR EACH ROW
          WHEN(NewRow.Qty < 5)
          BEGIN ATOMIC
      INSERT INTO ReOrder
          VALUES(NewRow.ISBN,CURRENT_DATE,10);
     END;
```

- The above trigger definition is not quite correct
  - Once the quantity goes below 5, each update causes the trigger to fire as long as the value remains below 5.
  - Should fire only when the number of copies goes from or above 5 to below 5.

Triggers in SQL (contd.)

Old & New Values

- Here is the revised definition:

```
CREATE TRIGGER ReOrderTrigger
    AFTER UPDATE ON Books
    REFERENCING NEW ROW AS NewRow,
                OLD ROW AS OldRow
    FOR EACH ROW
       WHEN(NewRow.Qty<5 AND OldRow.Qty>=5)
       BEGIN ATOMIC
       INSERT INTO ReOrder
       VALUES(NewRow.ISBN,CURRENT_DATE,10);
       END;
```

Triggers in SQL (contd.)

Firing

- Upon the occurrence of a trigger event and if the *condition* is satisfied, then the trigger fires.
- By default, a trigger action is executed only once.
- If multiple rows are changed by the operation activating the trigger (e.g., **`DELETE`** or an **`UPDATE`**), then trigger action can be executed once for each row
  - use the clause **`FOR EACH ROW`**.
- Triggers are executed as part of the transaction activating trigger.
  - If trigger action does not execute successfully, then the transaction activating the trigger is aborted.
  - No change will be made to the database.

Objects

- Extreme popularity of object-oriented languages such as C++ led a push towards object databases.
- Advantages:
  - Can store, retrieve, & update data in terms of the application domain.
  - No “impedance” mismatch.
  - Objects can reference related objects obviating need for joins.
- Disadvantages
  - No formal object database model with clean semantics.
  - No standard declarative language like SQL for querying & manipulating the database.
- Object databases have been relegated to specialized applications.
- Importance of object facilities is now appreciated
  - Relational databases are now morphing into object-relational databases.
- New versions of SQL provides object facilities.
  - Currently MySQL does not support user-defined types.

Objects in SQL

- Object types are sets of values and operations (methods) that can be performed on these values.
- An *object* is an instance of a user-defined type defined as

```
CREATE TYPE TypeName AS
        (Attribute definitions) FinalOrNot
        Method specifications
;
```

- The attributes define the values a user-defined type object can have.
  - *FinalOrNot* indicates whether or not a new type can be based on the type – **`FINAL`** or **`NOT FINAL`**.
- Objects are manipulated using methods.
- Method specifications are “signatures” or declarations of the methods that specify their name, parameters, and the types of values they return.
  - Methods must be defined separately using the **`CREATE METHOD`** statement.
- User-defined types can be used to define both attribute (column) types and row types.

Using Objects

- Consider a database that stores Cartesian coordinates but that allows access to them as polar coordinates.
- Relationship between the Cartesian and polar coordinates

$$x = r \times cos(theta), \;\; y = r \times sin(theta)$$

$$r = sqrt(x^2 + y^2), \; theta = tan^{-1}(y/x)$$

- We define type **`Coordinates`** to store Cartesian coordinates along with methods for polar coordinates:

```
CREATE TYPE Coordinates AS (X FLOAT, Y FLOAT) FINAL
  METHOD GetRadius() RETURNS FLOAT,
  METHOD GetTheta()RETURNS FLOAT,
  METHOD PutPolar(Radius FLOAT, Theta FLOAT)
                       RETURNS Coordinates
;
```

Using Objects (contd.)

- A **`Coordinates`** object can be retrieved and updated in terms of Cartesian coordinate system using the methods generated for the attributes X and Y.
- For each attribute of a user-defined type, SQL generates 2 methods
  - an *observer* method for retrieving the value of the attribute
  - and a *mutator* method to update the attribute.
- Methods **`GetRadius()`**, **`GetTheta()`**, & **`PutPolar()`** for manipulating coordinates in the polar coordinate system.
  - **`GetRadius()`** and **`GetTheta()`** retrieve values but do not update a **`Coordinates`** object and are classified as observer methods.
  - Method **`PutPolar()`** changes the value of the associated **`Coordinates`** object and is classified as a mutator method.

Using Objects (contd.)

- Complete list of the attributes, automatically generated functions, and methods for **`Coordinates`**:
  - **`Coordinates()`**: A *constructor* function that returns a new object of type **`Coordinates`**. A basic constructor function, one without arguments is automatically generated by the database system. Users are allowed to define new constructor functions with parameters to initialize newly created objects.
  - **`X`**, **`Y`**: Attribute names.
  - **`X(X FLOAT)`**, **`Y(Y FLOAT)`**: Automatically generated mutator methods for changing the values of attributes **`X`** and **`Y`**.
  - **`X()`**, **`Y()`**: Automatically generated observer methods that return values of the attributes **`X`** and **`Y`**.
  - **`GetRadius()`**, **`GetTheta()`**, and **`PutPolar()`**: Explicitly defined methods for manipulating, as polar coordinate values, objects of type **`Coordinates`**.

Using Objects (contd.)

- Object components are accessed using methods invoked in the context of an object instance.

  objectInstance.method call
- Within a method, the object instance can be referenced using the keyword **SELF**.
- As an example, consider the table

```
CREATE TABLE Locations (
        Company VARCHAR(30),
        LocatedAt Coordinates
);
```

- The following query prints the location of company Acme in both Cartesian and polar coordinates:

```
SELECT Company, LocatedAt.X(), LocatedAt.Y(),
        LocatedAt.Radius(), LocatedAt.Theta()
FROM Locations
WHERE Company = 'Acme';
```


Object Types – Method Bodies

```
CREATE INSTANCE METHOD GetRadius() RETURNS FLOAT FOR
  Coordinates
BEGIN
    RETURN SQRT(SELF.X*SELF.X + SELF.Y*SELF.Y);
END;

CREATE INSTANCE METHOD GetTheta()RETURNS FLOAT FOR
  Coordinates
BEGIN
    RETURN ATAN(SELF.X/SELF.Y);
END;

CREATE INSTANCE METHOD PutPolar(Radius FLOAT, Theta FLOAT)
        RETURNS Coordinates FOR COORDINATES
BEGIN
    SET SELF.X = Radius * COS(Theta);
    SET SELF.Y = Radius * SIN(Theta);
    RETURN SELF;
END;
```

Creating an Object & Using it

- Creating an object of type **`Coordinates`** and using it in a row inserted into the **`Locations`** table:

```
DECLARE Pos Coordinates;
SET Pos = Coordinates();
SET Pos = Pos.X(2.5);
SET Pos = Pos.Y(2.5);
INSERT INTO Locations VALUES('Logix', Pos);
```


Constructors

- The default constructor for **`Coordinates`** does not assign a value to a newly created object. However, we can define one that does :

```
CREATE FUNCTION Coordinates(X FLOAT,Y FLOAT)
      RETURNS Coordinates
BEGIN
      DECLARE Pos Coordinates;
      SET Pos = Coordinates();
      SET Pos.X = X;
      SET Pos.Y = Y;
      RETURN Pos;
END;
```

- The code for inserting the location of Logix can now be written as

```
INSERT INTO Locations
     VALUES('Logix', Coordinates(2.5, 2.5));
```

Table of Objects

```
CREATE Type Address AS (
        Company VARCHAR(30), Street VARCHAR(50),
        City VARCHAR(30), State2 CHAR(2),Zip CHAR(5)
) FINAL;

CREATE TABLE CompanyAddresses OF Address
        REF IS AddrId SYSTEM GENERATED;
```

- **`CompanyAddresses`** is called a *typed* table – each row is an object of type **`Address`**.
- Each object is given a unique id generated by the database system whose value is stored in the reference field **`AddrId`**, which is associated with each **`Address`** object
- Example code :

```
BEGIN
    DECLARE Addr Address;
    SET Addr = Address(); SET Addr = Addr.Company('Avaya');
    SET Addr = Addr.Street('211 Mt. Airy Rd');
    SET Addr = Addr.City('Basking Ridge');
    SET Addr = Addr.State2('NJ');SET Addr = Addr.Zip('07920');
    INSERT INTO CompanyAddresses VALUES(Addr);
END;
```


Referencing Objects

- SQL references (pointers) are similar to object ids in object databases.
- Consider a modified version of table **`Locations`**, modified to include a reference to the company address:

```
CREATE TABLE Locations2 (
        Company VARCHAR(30),
        Addr REF(Address),
        LocatedAt Coordinates
);
```

- **`Addr`** will point to a row an **`Address`** object, a row of the table **`CompanyAddresses`**.
- Operator **`->`** is used to refer to components of the referenced object.
- Consider the following query that prints the locations of companies in NJ:

```
SELECT Company, LocatedAt.X(), LocatedAt.Y()
FROM Locations2
WHERE Addr->State2() = 'NJ';
```

Referencing Objects

- The following example illustrates finding a reference to a row of a typed table and its use.
  - We will find the reference to the row containing the address of Avaya and insert it into **`Locations2`**.
  - Note: **`AddrId`** is a system generated reference to the rows in **`CompanyAddresses`**:

```
BEGIN
        DECLARE AvayaPos Coordinates;
        SET AvayaPos = Coordinates();
        SET AvayaPos = AvayaPos.X(2.0);
        SET AvayaPos = AvayaPos.Y(3.0);
        INSERT INTO Locations2
                SELECT 'Avaya', AddrId, AvayaPos
                FROM CompanyAddresses
                WHERE Company() = 'Avaya';
END;
```


Comparing Objects

- To search for a value of a user-defined type, it is necessary to define when two such values are equal.
  - a function named **`EQUALS`** is defined for the user-defined type.
  - This function returns **`TRUE`** if two values given as arguments are equal, **`FALSE`** otherwise.
  - This function enables operations **`=`**, **`<>`**, **`DISTINCT`**, & **`GROUP BY`**.
- The statement

```
CREATE ORDERING FOR Coordinates
            EQUALS ONLY BY STATE;
```

creates an **`EQUALS`** function
  - returns **`TRUE`** if all attributes of two **`Coordinates`** values given as arguments are equal.
  - Otherwise, it returns **`FALSE`** .

Comparing Objects (contd.)

- To order user-defined values it is necessary to be able to rank the values.
- A general comparison function can be defined as a **RELATIVE** function using the **CREATE ORDERING** statement
  - A **RELATIVE** function enables operations =, <>, **DISTINCT**, **GROUP BY**, <, >, <=, >=, <>, and **ORDER BY**.
  - A **RELATIVE** function takes two arguments A and B and returns an integer less than, equal to, or greater than 0 depending upon whether A is less than, equal to, or greater than B.
- Example:

```
CREATE ORDERING FOR Coordinates
        ORDER FULL BY RELATIVE WITH FUNCTION Cmp;
```

- The body of the this function is defined as

```
CREATE FUNCTION Cmp(C1 Coordinates,C2 Coordinates) RETURNS INTEGER
BEGIN
    IF C1.X() = C2.X() AND C1.Y() = C2.Y()
    THEN RETURN(0);
    ELSEIF C1.X()*C1.X()+C1.Y()*C1.Y()<C2.X()*C2.X()+ C2.Y()*C2.Y()
    THEN RETURN(-1);
    ELSE RETURN(1);
    END IF;
END;
```


Indexes

- A database *index* is a data structure that enables a database system to improve performance.
  - Enables searches for specific rows in a table without having to potentially scan the whole table.
- Databases typically store data as files on disks.
  - Files are composed of disk blocks, the unit of retrieval from and writing to disk.
  - Tables are stored using disk blocks, which may be contiguous or linked together.
  - A disk block will typically store many rows, the number depending upon the size of the disk block and the size of the row.
  - Disk access is much slower than memory access.
  - Scanning the rows in a disk block once it has been brought to memory is very fast compared to a disk access.
  - The idea behind indexes is to minimize disk accesses.

Indexes (contd.)

- Correct design of a database and transactions is of paramount importance.
- Also of great importance is transaction speed.
- Without indexes, transactions may take a very long time to run.
  - The Books table will have millions of entries reflecting the books in print.
  - Locating a book without the help of an index will potentially require scanning the whole table – which could take a long time.
- An index allows searching for values of interest without requiring scanning of all the rows in a table.
  - Scanning a table takes processing time and disk I/O time that is proportional to the size of the table.
  - Moreover, scanning a whole table may reduce concurrency, that is, it can prevent multiple transactions from running in parallel.

Indexes (contd.)

- Indexes operate behind the scenes.
  - Specified by the database administrator or the authorized users.
  - Database systems keep indexes up to date & use them in queries as appropriate.
- Indexes help speed up operations such as
  - **`SELECT`** operations with **`WHERE`** clauses
  - joins
  - aggregation functions such as **`MAX`**.
- A database index is similar in concept to
  - a book index
  - a library catalog
- For example, here is part of an index of a book:

```
AT&T Labs 3, 42
AT&T trivestiture 35, 41
Baby Bells 36, 38, 41
bankers hours, better than 108
```

- Without an index
  - looking for a word or a phrase could involve scanning the whole book.
  - **Overhead**: the word or phrase has to be searched in the index and in the page

Indexes (contd.)

- Index Issues:
  - how an index is used,
  - the cost of using an index (time and space),
  - database items that should be indexed,
  - the different types of indexes, and
  - when should indexes be updated.
- Indexes used in databases are often more sophisticated than the book index or the library catalog
  - they serve the same function.
  - they help locate items quickly.
- In case of databases, end users do not have to be aware of the index.
  - An appropriate index, if one exists, is automatically used and kept up to date behind the scenes.
  - Only database administrators who design the databases, programmers, users defining their own databases, and serious database users need know about indexes.

Indexes (contd.)

- There is a cost to using the index.
  - searching the index,
  - building the index
  - space used by it
  - *index maintenance* – changes with new items being added, items being deleted, and item values being changed, can require rearrangement of the index.
- Indexes are associated with tables
  - based on one or more columns
  - these columns are called the *search* fields.
- A database index is a data structure that facilitates quick look up of items with specific values.
- An index entry can be a pair consisting of an attribute (column entry) value and a pointer to a location in the database, the disk address of a disk block, where an item with this value can be found.
- Indexes are stored on disk for permanence.

Indexes (contd.)

ISBN	Title	Price	Authors	Pages	PubDate	Qty
0929306279	Bell Labs	29.95	Gehani	269	2003	121
0929306260	Java	49.95	Sahni & Kumar	465	2003	35
0670031844	White Moghuls	34.95	Dalrymple	459	2003	78
0439357624	Born Confused	16.95	Hidier	432	2002	11

- Suppose **`Books`** stores a significant portion of *Books in Print*®, which has about 3.5 million entries.
- Without an index, queries such as
  - *List the titles all the books authored only by Gehani*
  - *List the titles of all the books authored only by Gehani and published between 1992 and 1995*

would require scanning the 3.5 million entries

- Queries would be painfully slow
  - large number of disk accesses.
- An index on **`Authors`** column, will allow these queries to execute quickly
  - information found with a with few disk accesses.

Indexes (contd.)

```
SELECT Title
FROM   Books
WHERE  Authors = 'Gehani';
```

and

```
SELECT Title
FROM   Books
WHERE  Authors = 'Gehani' AND
       (PubDate >= 1992 AND PubDate <=1995)
```

- An index on the **`Authors`** column will help to quickly locate all rows with Gehani as author.
- In case of the 2nd query, for the 2nd part of the **`WHERE`** expression, rows that do not have the right publication date must be filtered out.
- However, with an index on **`PubDate`** a sophisticated optimization algorithm
  - could find disk addresses of rows with Gehani as author
  - then find addresses of rows with the right publication date,
  - determine intersection of these two sets of addresses, and
  - use result to retrieve rows satisfying the **`WHERE`** clause.

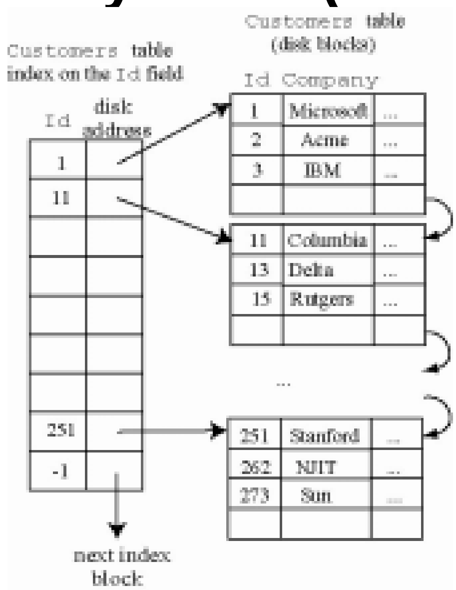
How Does an Index Work?

- An index consists of column values or value ranges and disk addresses
  - values and value ranges are used to control search for locating rows.
- Instead of scanning a table to find rows for a query, the index entries corresponding to these rows are first located.
  - Rows are then located using the disk addresses in the indexes and then retrieved.
- All this happens behind the scenes.
  - Queries are automatically transformed to use indexes.
  - Whenever a database item is changed or deleted, or a new item added, appropriate indexes are automatically updated.
- The index is stored on disk.
  - Portions of an index are brought to memory as needed and left there (as long as memory is available).
  - If the whole index can fit into memory, then database accesses will be speeded up even more.
  - Indexes are another example of space vs. time tradeoff.
- Indexes that work on multiple search values simultaneously are called multi-dimensional indexes.

Primary Index

- An index on the primary key column is called the *primary index*.
- Indexes on other columns are called *secondary indexes*.
- The rows in a table are ordered as needed by the primary index to facilitate fast access using and this affects their layout on disk.
- As an example, consider the following primary index on the (customer) Id column of the **`Customers`** table

Primary Index (contd.)



Primary Index (contd.)

- **`Customer`** table rows are ordered on the primary index (key), that is, **`Id`** values.
  - records with index values "close" to each other are located in close proximity of each other on disk.
- Such an index, called a "clustering index"
  - makes it efficient to search for a range of values.
- In the figure, the first index entry points to rows on disk starting with **`Id`**s >= 1 but < 11.
- The location of a row with a specific id is determined by finding the appropriate index entry
  - using binary search
- The disk block (possibly) containing the row is brought to memory and the row is found
  - using binary search.

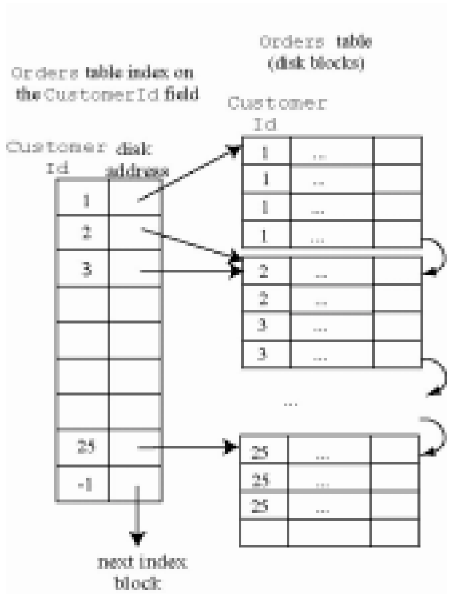
Primary Index (contd.)

- Deletions are easy but insertions are tricky. Strategies:
  - Leave empty slots for new rows in each disk block when table is first created.
    - Row is inserted in the "appropriate" position in the disk block.
    - When disk block runs out of space, a new block is created & "chained" to the original block & its successor and used for overflow.
  - Split full block into two, distribute rows between the two blocks, and update the index.
- Tables stored in files organized are referred to as *index-sequential files*.
  - Allow random access to specific records using the index
  - Whole table or part of it can be accessed sequentially

Clustering Indexes

- An index is said to be *clustering* if proximity of index entries implies that the corresponding rows on disk are close together.
- Clustering indexes are
  - valuable for queries such as range queries.
  - very fast for locating rows with the same search field value.
- The primary index is a clustering index
  - E.g., index on `Id` column of the `Customers` table is a clustering index.
  - In presence of primary index, secondary indexes are unclustering because rows are ordered by the columns making up the primary index.
  - A secondary index can be clustering only if there is no primary index.
- Consider potential uses of the `Orders` table.
  - look up all or a subset of the orders of a specific customer, that is, rows with the same `CustomerId`.
  - A clustering index on `CustomerId` will be of value for such queries.

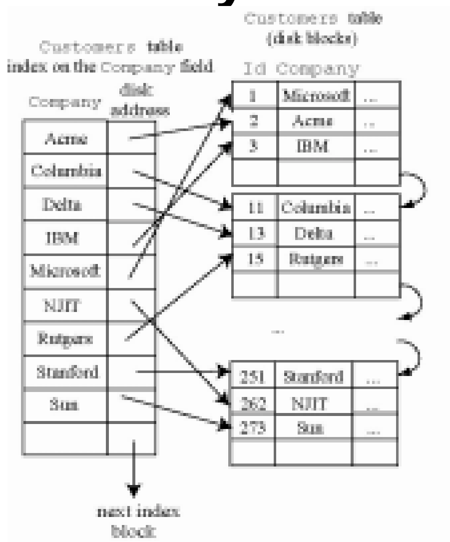
Clustering Indexes (contd.)



Secondary Indexes

- Indexes other than the primary index are called *secondary* indexes.
- They do not determine the order in which the rows of a table are stored.
- Because secondary indexes cannot rely on row ordering, they must be dense indexes
  - must have an entry for each value in the column being indexed.

Secondary Indexes



Dense vs. Sparse Indexes

- Unlike a dense index, a sparse index does not contain an entry for each row of the table
  - Contains entries for some rows of a table, for example, entries for the first row of each disk block containing sorted rows.
  - Updated only when insertion of a new row requires splitting a disk block into two, when deleting a row results in an empty block, and possibly when a row is updated.
  - Rows without entries in the index can be located by starting with the indexed rows.
- Dense indexes require more disk space than sparse indexes because of the extra entries.
  - Speedup is significant in case of secondary indexes because otherwise, searching for a row with a specific non primary key value would require a sequential search of the table.
  - Require more maintenance because they must be updated whenever a new row is inserted, the indexed value is updated, or a row is deleted.

Multi-level indexes

- In case of very large tables, a sparse index can become very large.
- To speed up search, an additional index may be needed to search the index.
  - Such additional indexes are often seen on Web pages.
- Suppose you are looking for information about company BAAN at a website.
  - the website will display an index consisting of the letters in the alphabet
    A B C D E ... Z
  - Clicking on B will lead you to another index
    Ba Be Bi ...
  - Clicking on Ba yields a list of all companies whose name begin with Ba
- The above is an example of a 2-level multi-level index
  - All levels except the last level are indexes for finding entries at the next level.
  - The last level of the multi-level index, called the leaf level, points to the data records.

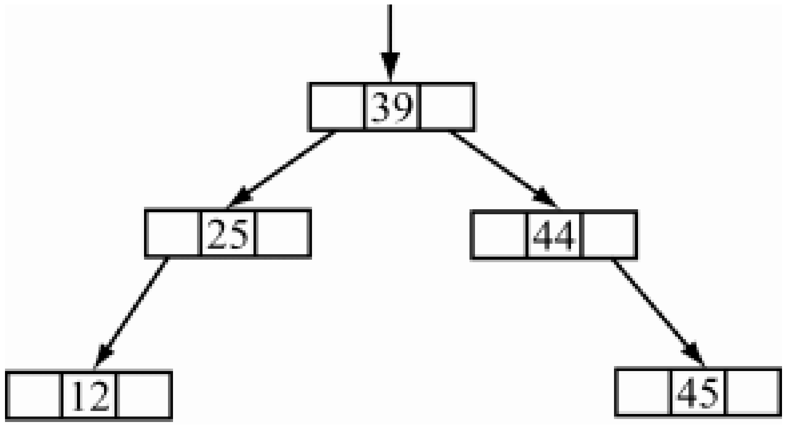
B-Tree Indexes

- We have discussed indexes with entries organized linearly.
  - By organizing the index as a tree, the index search can be speeded up.
- Two important tree indexes are the B-tree and B+-tree indexes
  - disk-based cousins of the balanced binary tree – stored entirely in main memory.

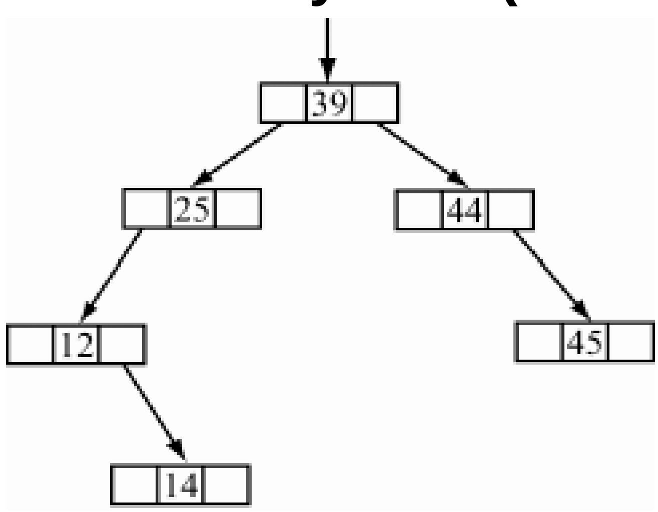
Balanced Binary Tree

- *Balanced binary trees*
  - binary trees
  - subtrees are kept balanced, i.e., each subtree, all the leaf nodes are at the same depth or distance, ±1, from the root node of the subtree.
- In a balanced binary tree with n nodes
  - at most $|\log n| + 1$ nodes need to be examined to find a value in the tree.
- A ordinary binary tree with n nodes can degenerate to a linked list
  - up to n nodes may have to be examined to find a value in the tree

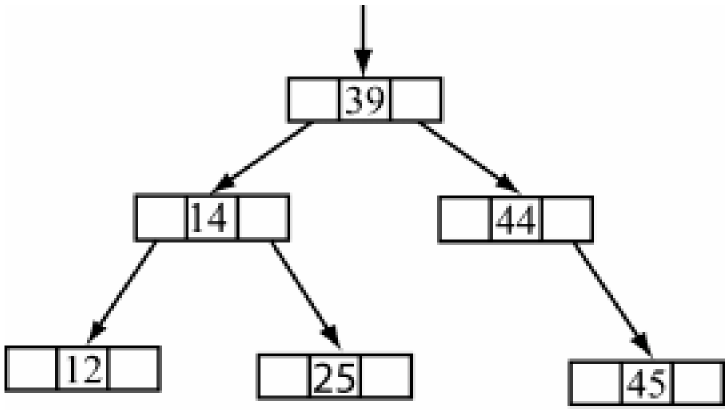
Balanced Binary Tree (Contd.)



Balanced Binary Tree (Contd.)



Balanced Binary Tree (Contd.)



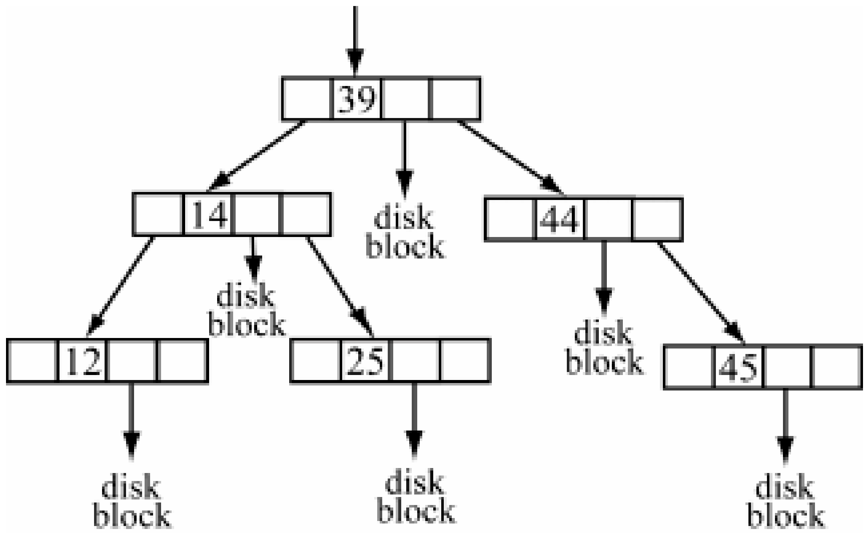
B-Trees

- A *B-tree* is a multi-level index structure for use with data on disks and it is itself stored on disk.
  - Similar to main memory balanced binary trees.
  - Each node is a disk block – can hold much more than 2 pointers.
  - As a result, the depth (height) of B-trees is very small.
    - E.g., in case of 1024 values, if each node has 32 subtrees, then the depth of such a B-tree will be 2.
- The smaller depth is very important because accessing disk blocks is slow and it is important to minimize disk accesses.
  - Going from one level of the B-tree to another requires fetching a disk block .
  - The smaller the depth, the smaller the number of disk access required to locate a value.

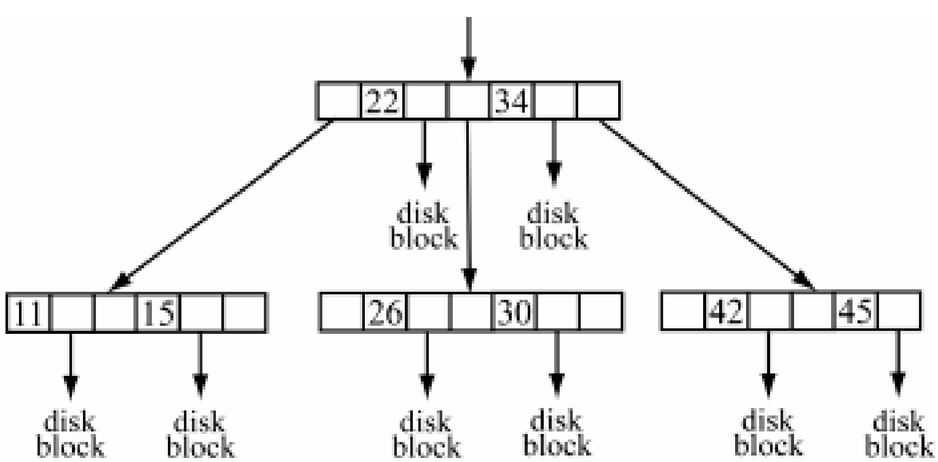
<u>Example of a B-tree node</u>

$addr_1$	$value_1$	data $addr_1$	$addr_2$	…	$addr_{n-1}$	$value_n$	data $addr_n$	$addr_{n+1}$

B-Tree of Order 2



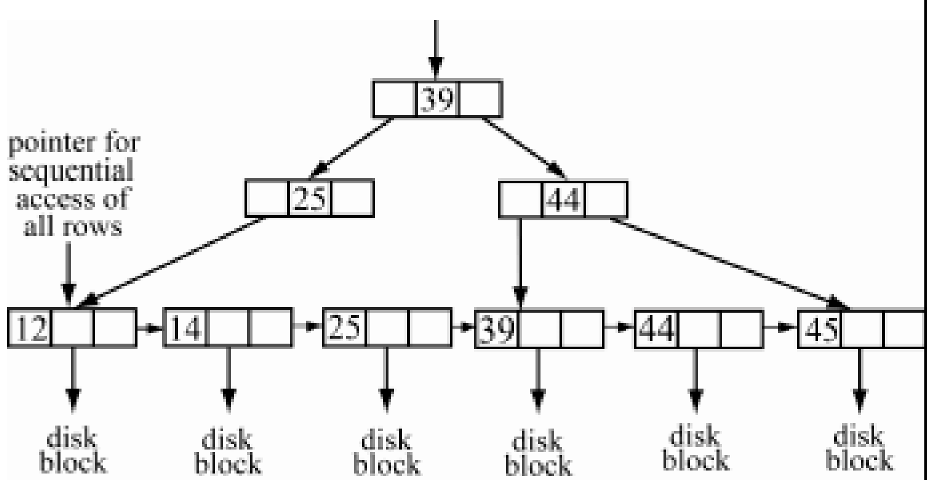
B-Tree of Order 3



B+-Trees

- B+-trees are similar to B-trees
  - pointers to data records are stored only in the leaf nodes.
- B+-trees combine the multi-level indexing properties of the B-tree with those of a clustering index.
- Random rows can be accessed using multi-level indexing
- All or some subset of the rows with can be accessed sequentially in sorted order of the search value.

B+-Tree Order 2



© Narain Gehani

Hash Indexes

- A hash function takes a search value, member of a large set, and maps it to another value, member of small set.
- Hash indexes use hashed values as subscripts for the *hash table*, to store & retrieve the address of the disk block that contains the row with the search value.
  - Elements of hash table are called *buckets*.
  - Hash table is stored on disk & brought to memory before it is used.
- Search value may belong to a large set but in practice it will be one a of small set.
  - The set of hashed values is also a small set.
  - The hash set values act as “directory” entries to find values stored in the database.
- Hash indexes are good for equality searches.
- Hashing maps the search value into a bucket
  - contains search values + addresses of the disk blocks containing rows with the search values.
- Inserting a new row in the database
  - requires updating the hash table by inserting the search value in the appropriate bucket + address of the disk block containing the row with the search value.
  - If the bucket is full, then adding a new row in the hash table requires an an “overflow” bucket.

Hash Indexes (Contd.)

- Hash function *h* maps a search value into a disk address

hashTable[*h*(*searchValue*)].*diskAddress*

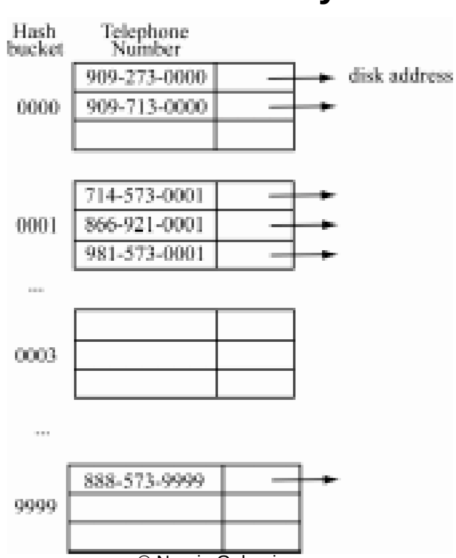
- A hash function maps the set of search values, say *k* values, into *n* values, where *n* is the number of buckets.
  - If $k >> n$, then there can be many collisions
  - When this happens, a bucket may overflow leading to multiple disk blocks which will need to be chained together for searching.
- A good hash function will map each search value to a different bucket.
  - Will require a large amount of storage.
  - In practice, the number of possible search values at any given time is going to be much smaller than the number of possible search fields.
  - This is what the hash function counts on.
- One difference between hash indexes and other indexes
  - hashing uses a calculated value to search for a row instead of a data structure, like a tree, that must be traversed.
- Hashing is used to speed up searches
  - Calculation of the hash value should be fast
  - Number of collisions small.

Hashing Example -- Electric Utility

- Electric utility has 10,000 customers
  - wants to use 10-digit telephone numbers to locate customer information.
- A 10-digit telephone number, can have up to 10^{10} possible values.
  - Can use an array of 10^{10} elements, each element pointing to a disk block with the appropriate customer information.
  - Will make locating customer information very simple.
  - But using an array with 10^{10} elements requires a lot of space!!!
  - Since company has only about 10,000 customers, using an array with about 10,000 elements, or even 20,000 elements will be reasonable
  - Will allow us to have our cake and eat it – we will get fast access to customer information without using much storage.
- The idea behind hashing is take a large number of values and map them into a smaller number of values.
  - In this example, challenge is to map 10^{10} possible values into 10,000 values,
  - Then we need only allocate an array of 10,000 elements for the hash table.
- One hash function that can be used is one that maps a telephone number to its last four digits.
  - Simple function maps 10^{10} values to 10,000 values.

Hashing Examples (Contd.)

Electric Utility



Hashing Examples (Contd.)
Electric Utility

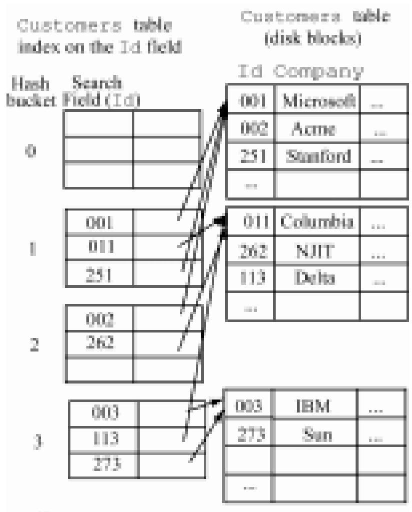
- The hash table is an array that is normally stored on disk,
  - the bucket indexes range from 0 to 9999.
  - each bucket can store multiple telephone numbers (3 as shown).
  - overflow → additional buckets are created and chained together.
- The hash function maps a customer's telephone number to a bucket.
  - To retrieve customer information, we find the customer's telephone number in the bucket and then use the associated disk address to access the data on disk.
  - If a new customer is being added to the database, then the hash table must be updated by inserting the customer's telephone number in the appropriate bucket along with the address of the customer information on disk.
- One issue in using hash functions is handling *collisions*.
  - lots of telephone numbers have the same last four digits.
  - Collisions are not a problem, because each bucket stores the customer's telephone number which is used to get the customer information from disk.

Hashing Examples (Contd.)
Customer Table

- Consider the use of a hash index for the customer id column **`Id`** of the **`Customers`** table.
- The hash function we will use for the **`Customers`** table will map Ids to the last digit of the **`Id`**.
- Note that because **`Id`** is a key of the customer table, there will be no duplicates.

Hashing Examples (Contd.)

Customer Table

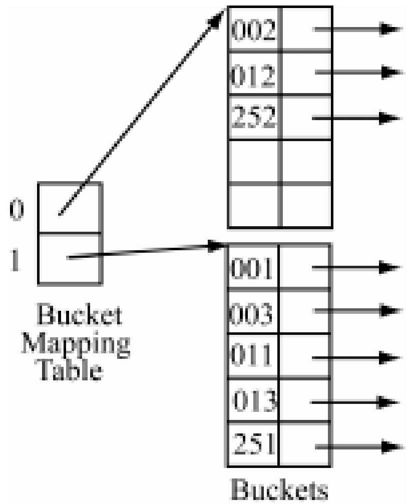


Dynamic Hashing

- *Static* hashing – Hashing using a fixed hash function and a fixed number of buckets
  - This type of hashing may not meet the needs of a growing or shrinking database.
  - Database growth may increase the collisions leading to overflow buckets.
  - Shrinking database can lead to space wastage with unused buckets.
- *Dynamic* hashing – Hashing functions cognizant of database size and can map to a variable number of buckets as the database grows or shrinks.
  - Unlike static hashing, dynamic hashing uses a data structure to map search values to buckets.
- In one version of dynamic hashing, called *extendible hashing*, the number of bits of the hashed value used *to locate* a bucket can increase or decrease depending upon the database size.
  - Bits point to an entry in a *bucket-mapping* table (*directory*), which contains the address of the bucket.
  - The number of buckets increase or decrease as the number of bits used to locate the buckets increase or decrease.
  - Buckets are increased or decreased by splitting a bucket into two or combining two buckets into one.

Dynamic Hashing Example

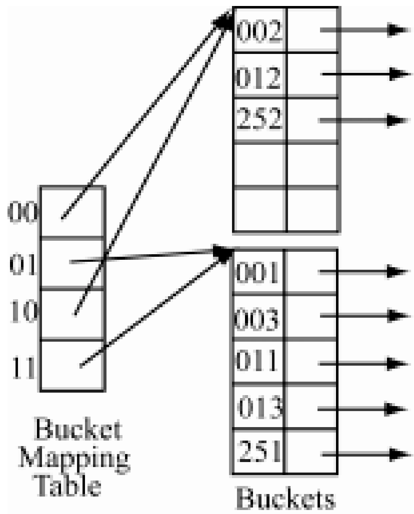
- Suppose we map a search value to the last bit of its hash value (last 0 or 1)
- Bit used as an index of the bucket mapping table, at this time of size 2.



Dynamic Hashing (Contd.)

- Suppose the size of database increases.
- Now, instead of using the last bit of the hash value, we will use the last 2 bits.
- This will increase the size of the bucket mapping table to 4.
- And we will partition the overflowing buckets (we can now have up to 4 buckets).
  - Assume, as shown, that the second bucket is full.
  - In the first step we split the bucket mapping table:

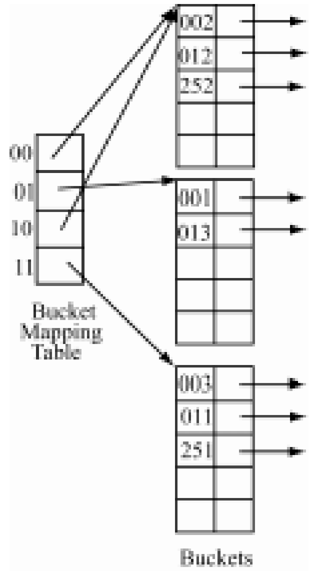
Dynamic Hashing (Contd.)



Dynamic Hashing (Contd.)

- The bucket mapping table has been expanded to have 4 entries and each entry points to the appropriate bucket.
- However, these entries still point to the original two buckets.
  - So far, we have not done anything with the buckets.
- The next step is to split the buckets to take into account that the bucket mapping table can map to 4 buckets.
  - At this time, it is only necessary to split the second bucket, which is filled up and ready to overflow.
  - This bucket originally contained hash values with the last bit equal to 1.
  - We will split this bucket into two so that the buckets either contain values that end in 01 or in 11

Dynamic Hashing (Contd.)



Thoughts on Indexing

- Indexes are particularly effective
  - when the tables are large
  - the number of updates is much smaller compared to the number of read-only queries, and
  - when query speed is very important.
- There is a cost to using indexes in terms of storage and execution time.
- **B-Tree vs. B+-Trees**
- B- and B+-trees are both multi-level indexes with access structures tghat aim to minimize the search. They differ in that in case of B+-trees *only* the leaf nodes to data records (rows of tables).
- The biggest advantage of B+-trees over B-trees is that the table rows stored on disk can be accessed sequentially in the order of the sorted values of the indexed column. Compared to B-trees, there is a small additional cost for locating rows randomly based on search value. The index must be searched to the leaf level (not much of a cost if it is already in memory) before the data blocks are accessed and some search values have to be stored multiple times – in the internal (non-leaf) nodes and in the leaf nodes. Also, the leaf nodes contain the search values for each row in the database.

Thoughts on Indexing (Contd.)

B-Tree vs. B+-Trees

- B- and B+-trees are both multi-level indexes with access structures that aim to minimize the search.
- They differ in that in case of B+-trees *only* the leaf nodes point to data records (rows of tables).
- The biggest advantage of B+-trees over B-trees is that the rows stored on disk can be accessed sequentially in the order of the sorted values of the indexed column.
- In B+-trees, there is a small additional cost for locating rows randomly based on search value.
  - The index must be searched to the leaf level (not much of a cost if it is already in memory) before the data blocks are accessed.
  - Some search values have to be stored multiple times – in the internal (non-leaf) nodes and in the leaf nodes.

Hashing vs. Indexing

- Differences between hashing (hash indexes) and indexing
  - Static hashing does not require traversing an access data structure.
  - Dynamic hashing requires traversing an access data structure – but overhead is minor.
  - In hashing, data access is based on the hashed value while in indexing it is based on the search value.
- Advantages of hashing
  - Fast for equality searches.
- Disadvantages of hashing
  - May not be order preserving to allow sequential access of rows in sorted order of search value.
  - Typically does not lead to clustering.
  - Inappropriate for use in queries looking for values in a specified range or values that match partially.
  - A good hash function design that minimizes collisions and is simple to compute depends upon knowing the expected size and distribution of the data. Such information may not be available.
  - Can waste space if the buckets are sparsely populated.
  - Overflow buckets slow data access.

Using Hashing vs. Indexing

- We will only consider static hashing (common), and B+-tree indexes which have many advantages over index-sequential file or B-tree indexes.
- Hashing is good for equality searches.
  - Fast because bucket address is computed and address of row on disk can be located quickly.
  - Using a B+-tree requires traversing the tree to locate search value.
- Unlike hashing, B+-trees support sequential scans, range queries, partial match queries, in addition to equality searches but these are not as fast as in case of hashing.

Index Performance

- Based on the number of disk accesses required to locate a database item.
- Accessing a database item
  - hashing, on the average, takes a constant amount of time
  - indexing takes time that is proportional to the number of items in the index.
  - In case of B- or B+-trees, the time is proportional to the height of the tree, which depends upon the size of the database, the number of disk blocks in use, size of the disk blocks, and the "fan out" of the tree.

Cost of Using an Index

- The use of an index to speed up performance does not come for free.
- The costs of using an index must be weighed the with respect to the speedup provided by the index.
  - *Building an Index*: An index must be built initially. This will take up CPU time but this is a one-time operation.
  - *Looking up the Index*: Not much especially if the portion of the index needed is already in memory.
  - *Index Maintenance*: The index must be updated to reflect additions, deletions, and updates of the items that it indexes. This will slow update queries since an addition, deletion, and update of a row potentially requires a corresponding index update.
  - *Storage Space*: Indexes use space and in return give fast access – another example of the classic storage versus execution time tradeoff.

Indexes in MySQL

- SQL focuses on the logical aspects of data.
  - Considers other items outside its purview.
- Indexes are no longer considered part of SQL
  - performance items
  - affect the physical view but not the logical view of the data.
- Databases include indexes in their SQLs
  - indexes are very important for fast data manipulation.
- Indexes in MySQL assist the database system in quickly
  - finding rows that match a WHERE clause,
  - eliminating rows from consideration,
  - getting rows from tables when performing joins,
  - computing the MAX() and MIN() functions,
  - sorting or grouping the results of a query, etc.
- All columns types can be indexed and an index can involve multiple columns .
- MySQL (InnoDB) uses B-tree indexes.
  - Uses only one index per query.
  - Picks index that gives the best results.

Indexes Creation in MySQL

- MySQL indexes can be specified in the table definition or created separately as

  CREATE INDEX [USING *IndexType*] *IndexName* ON *table*(*columns*);

  IndexType is just BTREE for InnoDB, which is the default.

```
CREATE TABLE Customers(
        Id INT(8) PRIMARY KEY, Company VARCHAR(30),
        First VARCHAR(30), Last VARCHAR(30),
        Street VARCHAR(50), City VARCHAR(30),
        State2 CHAR(2), Zip CHAR(5),Tel CHAR(10)
) ENGINE = InnoDB;
```

- We can define a index on the **First** and **Last** names as

```
CREATE INDEX CustomerName ON Customers(Last, First);
```

  last name was given first to allow the index to be used for the **Last** column by itself.
- MySQL will use this index for queries involving the column
  - **Last**
  - **Last** and **First**

Indexes in MySQL (Contd.)

Queries Helped by Indexes

```
SELECT *
FROM Customers
WHERE First = 'Liza' AND Last = 'Singh';

SELECT *
FROM Customers
WHERE Last = 'Singh';

SELECT *
FROM Customers
WHERE Last LIKE 'Si%';
```

Note: % matches any substring

Indexes in MySQL (Contd.)

- The **EXPLAIN** statement can help determine whether or not an index is being used in a query.
  - The **EXPLAIN** statement consists of the keyword **EXPLAIN** before the **SELECT** statement.
  - Will give information about how the **SELECT** statement will be executed.
  - In particular, will yield information about the indexes considered for the **SELECT** statement (under the column **POSSIBLE_KEYS**) and the index actually used (under the column **KEY**).
- For example,

```
mysql> EXPLAIN SELECT *
    -> FROM Customers
    -> WHERE Last = 'Singh';
```

yields a table that has information saying that MySQL considered the index **CustomerName** for the query and that it will use it.

Views

- A *view* is a table derived from other tables, called *base* tables.
  - A view can also be based on other views.
- Views are a mechanism for providing an alternative view of the data.
  - convenience, performance, and security.
- *Example*: views defined to contain sales data for each quarter for each of the top ten books.
  - Executives can simply look at these tables.
  - There will be no need for them to write queries to extract this data from the **OrderInfo** and the **Orders** tables.
- *Example*: A view in a corporate personnel database defined for non-managerial staff in HR to allow them to look at all columns of an **Employee** table except employee salary and performance ranking columns.
  - HR managers could be authorized to access the latter information.
- Views are defined as queries – *view definition queries*.
- Responsibility of the database system to ensure that a view is always “current.

Views (contd.)

- We will define a view **`SalesRank`** that lists the titles and quantities of books sold in the order of decreasing sales:

SALESRANK

Title	Total
Born Confused	4
Bell Labs	2
White Moghuls	2
Java	2

Views (contd.)

```
CREATE VIEW SalesRank AS
SELECT Books.Title,
    SUM(OrderInfo.QTY) AS Total
FROM OrderInfo, Books
WHERE Books.ISBN = OrderInfo.ISBN
GROUP BY Books.ISBN, Title
ORDER BY Total DESC;
```

Views (contd.)

- View definition query joins tables **`OrderInfo`** & **`Books`** on **`ISBN`**.
  - The **`GROUP BY`** clause groups together rows with the same **`ISBN`**
  - the aggregation function **`SUM()`** totals the sales quantities in each group, the column containing the sales totals is named **`Total`**
  - **`ORDER BY`** clause orders rows in descending order of **`Total`** column.
- View **`SalesRank`** can be used much like any table for queries.
  - some updates may be allowed depending upon the view definition and the database system.
- Queries that use the view
  - can be translated into queries that use the base tables.
  - may lead the database system to first compute the view and allow the resulting table to be used by the queries.
  - may use a pre-computed view stored as a table by the database system

Uses of Views

- *Ease of use*
  - Give a simplified role-based view of the database to users who do not need to see the complete database.
- *Personalized view*
  - Give users a personalized view of the data.
- *Simplify writing of queries*
  - Views are essentially prewritten queries.
  - In the absence of views, queries written against views will have to be rewritten to include what the view definition was doing thus making the query more complex.
- *Security*
  - Views help implement data security by including only data that should be accessible by the users.
  - SQL access privilege mechanisms can be used to further control user access.
- *Efficiency*
  - Views can be pre-computed and stored as “materialized” views.
  - For example, joins in a view definition query do not have to be recomputed every time when queries use the view. This would not be the case if the queries used the base tables directly.

Implementing Views

- *Virtual view*
  - The view is computed on demand. It is always current but has to be recomputed for every “user” query against the view.
- *Materialized view*
  - The view is computed and then stored just like an ordinary table.
  - It does not have to be recomputed for every user query against the view.
  - But it needs to be recomputed every time one of its base tables is updated (at some point before the next query).
  - However, in some cases, depending upon the policy being enforced by the database administrator, the view may be “out-of-date” for some time period – until the next time the view is scheduled to be recomputed.

Virtual Views

- Virtual views are computed by running the queries specified in their definitions, prior to running queries on the views.
- Alternatively, the database system may transform a query that uses views to an equivalent query on the base tables.

Materialized Views

- Materialized by executing view definition queries & storing the results.
  - If the base tables are modified, then the view may need to be recomputed.
  - Materialized views offer fast access, but they use storage.
  - Used to pre-compute potentially expensive and frequent queries such as joins and aggregations
  - **Motto**: Compute once but use results often.
- Materialized views can be updated, i.e., *maintained*, as part of the transaction that updates the base tables.
  - can increase transaction time significantly making it unacceptable.
  - not used in places where transactions need to execute fast.
- Materialized views are important for applications where the underlying base tables do not change frequently but in which the data is queried frequently.
  - E.g., data warehouses, which are very large databases that are updated infrequently.
- Materialized views are used to offer users quick access to remote databases.
  - Some applications are satisfied with almost current data provided they could get it fast.
  - Used to create a version of the database close to the user to avoid remote access delay.

Materialized Vs. Virtual Views

- Compute time versus storage tradeoff
  - Materialized views require storage but queries run faster since the views do not have to be recomputed for each query.
- A user query posed against a virtual view can be thought of as executing in two steps:
  - The view is first computed by executing the view definition query.
  - The user query is then executed to operate against the view.
- Execution of a user query against a materialized view is similar to the execution of a user query against a table
  - view may have to be "refreshed" if its base tables have been updated after its creation or the last refresh.
  - semantics must be the same as those of running a query against virtual views.

Cost of Views

- Databases systems do a lot of work behind the scenes to make views appear like ordinary tables to the users. For example:
  - *virtual views*: "view" definition query must be executed each time the view is referenced in another query
  - *materialized views*: views are updated (maintained) whenever anyone of their base tables are updated.
- Some rules that can help decide how to implement a view, as virtual or materialized:
  - If view is going to be queried frequently but its base tables will be updated infrequently, then materialize view.
  - If queries that use views need to be executed quickly, then best to materialize these views (to avoid computing a view to answer query).
  - If specific table updates need to be fast, then a view based on these tables should not be materialized (assuming materialized views are updated as part of the transaction updating base tables).
  - If number of updates on some tables is large and queries on views based on these tables infrequent, then best to implement views as virtual.

Updateable Views

- Views are like tables in many ways but there are differences.
  - views may not be updateable
  - only some types of updates may be allowed.
  - an apparently simple view update may take a long time because of much work behind the scenes.
- However, if views are to be more like tables, they should be updateable.
  - Updates must be automatically reflected in base tables.
- Only view updates that have enough information for base tables to be updated are theoretically allowable.
  - It is not always possible to map items in the view back to the base tables.
  - Critical data necessary for identifying the rows and columns in the based tables may not be present in the view.
  - Only those view updates where it is possible to correctly propagate the changes back to the base tables can be allowed.
- For example, if each element of a column in a view contains the sum of some rows of the base tables, then it is not possible to reflect a change to the sums back to base tables.

View Creation

```
CREATE VIEW viewName AS select-query;
```

```
CREATE VIEW SALES_2004_APRIL AS
SELECT OrderInfo.ISBN, Title,
       SUM(OrderInfo.Qty*OrderInfo.Price) AS
             Sales
FROM OrderInfo, Orders, Books
WHERE OrderInfo.ISBN = Books.ISBN AND
        OrderInfo.OrderId = Orders.OrderId
     AND
        ShipDate >= '2004-04-01' AND
        ShipDate <= '2004-04-30'
GROUP BY OrderInfo.ISBN, Title;
```


Spatial Databases

- Spatial databases are used for representing multi-dimensional objects
  - mountains
  - cities, roads
  - locations
  - VLSI layouts
- The standard relational model is not a good match for storing spatial data
  - data types and query operations do not match.
- Spatial databases are now being widely used
  - Extensions to relational databases to handle spatial data have been proposed.
  - Facilitate the building of store locators.
  - Facilitate the building of “yellow page searches.”

www.mapsonus.com

Spatial Databases (Contd.)

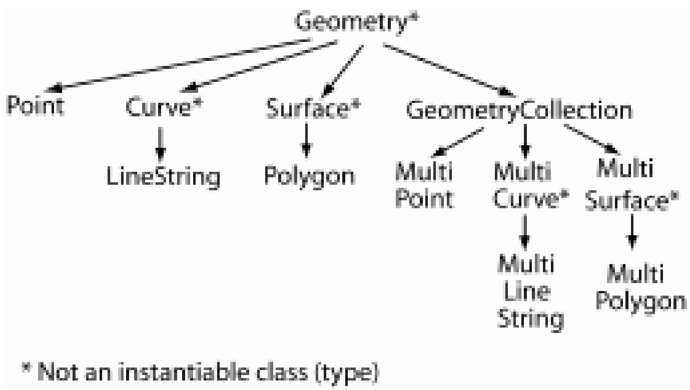
- Database systems that support the building of spatial databases provide
  - special data types, called *geometry* or *geographic* types
  - facilities to retrieve, query, and manipulate spatial objects
  - spatial indexes to speed up the retrieval, querying, and manipulation of spatial objects.
- Some examples of geometric types are
  - point
  - line
  - polygon, and
  - circle.
- Operations on geometric types allow users to determine whether or not
  - one object is contained in another
  - two objects intersect
  - one object is below or above another object, etc.

Spatial Database – MySQL

- Spatial database facilities are not part of standard SQL.
  - Many databases provide SQL extensions for storing, retrieving, & querying spatial objects.
- The Open Geospatial Consortium has proposed spatial extensions for SQL.
  - MySQL implements a subset of these spatial extensions.
  - Currently available only for MyISAM tables.
  - Note: MyISAM, unlike InnoDB, does not support transactions.

MySQL Spatial Data Types

- MySQL's supports spatial data types, called classes
  - All based on the class **`Geometry`**.
  - Users can define methods
  - Specialization tree shows spatial data types supported by MySQL.



Spatial Objects

Creating, storing, and retrieving

- Define one or more columns using the spatial data types.
  - Values of spatial types are defined using a textual description. E.g.:

```
'POINT(1 3)'
'POLYGON((0 0,4 0,4 4,4 0, 0 0))'
```

- Before storing a spatial value specified using the text description, it must be converted to a geometric type, using an appropriate function for the type, e.g., **`PointFromText`** for the type **`POINT`**

```
PointFromText('POINT(1 3)')
```

- When printing a spatial value, it must first be converted from a geometry type to text using the function **`astext`**. E.g.,

```
astext(Location)
```

Spatial Functions

- Take 2 **Geometry** type arguments ***g*1** and ***g*2** and return 1 (true) or 0 (false).
  - Can be called with any spatial data types since all types are specializations of **Geometry**.
- Functions that determine the relative positions of two spatial objects:
  - **Contains(*g*1, *g*2**): Returns 1 or 0 depending upon whether or not ***g*1** is completely within ***g*2**.
  - **Crosses(*g*1, *g*2**): Returns 1 or 0 depending upon whether or not ***g*1** spatially crosses *g*2. Two geometries cross if they intersect.
  - **Disjoint(*g*1, *g*2**): Returns 1 or 0 depending upon whether or not ***g*1** and ***g*2** intersect.
  - **Distance(*g*1, *g*2**): Returns the shortest distance between the two points in ***g*1** and ***g*2**.
  - **Equals(*g*1, *g*2**): Returns 1 or 0 depending upon whether or not ***g*1** is spatially equal to ***g*2**.
  - **Intersects(*g*1, *g*2**): Returns 1 or 0 depending upon whether or not ***g*1** intersects **g2**.
  - **Touches(*g*1, *g*2**): Returns 1 or 0 depending upon whether or not ***g*1** spatially touches ***g*2**.
  - **Within(*g*1, *g*2**): Returns 1 or 0 depending upon whether or not ***g*1** is spatially within ***g*2**.

Spatial Functions (Contd.)

- Functions that operate on the minimum bounding rectangle (MBR) of a 2-D spatial object:
  - **MBRContains(*g*1, *g*2)**: Returns 1 or 0 depending upon whether or not MBR of ***g*1** contains MBR of ***g*2**.
  - **MBRDisjoint(*g*1, *g*2)**: Returns 1 or 0 depending upon whether or not MBRs of ***g*1** and ***g*2** are disjoint.
  - **MBREqual(*g*1, *g*2)**: Returns 1 or 0 depending upon whether or not the MBRs of ***g*1** and ***g*2** are the same.
  - **MBRIntersects(*g*1**, ***g*2)**: Returns 1 or 0 depending upon whether or not the MBRs of ***g*1** and ***g*2** intersect.
  - **MBRTouches(*g*1, *g*2)**: Returns 1 or 0 depending upon whether or not the MBRs of ***g*1** and ***g*2** touch.
  - **MBRWithin(*g*1, *g*2)**: Returns 1 or 0 depending upon whether or not the MBR of ***g*1** is within the MBR of ***g*2**.

Spatial Type Functions

- **Class GEOMETRY Functions**
  - **Dimension(*g*):** Returns the dimension of **g**.
  - **Envelope(*g*)**: Returns the MBR of ***g***.
  - **GeometryType(*g*)**: Returns name of class of the ***g***.
- **Class POINT Functions**
  - functions **X()** and **Y()** yield the ***x*** and ***y*** coordinates
- **Class LINESTRING Functions –** Linestrings consists of **POINT**s
  - **EndPoint(*ls*):** Returns the end point of ***ls***.
  - **IsClosed(*ls*):** Returns 1 if start and end points of ***ls*** are same; otherwise 0.
  - **NumPoints(*ls*):** Returns the number of points making up ***ls***.
  - **PointN(*ls*, *n*):** Returns the n^{th} point in ***ls***.
  - **StartPoint(*ls*):** Returns the starting point of ***ls***.
- **Text Functions**
  - Convert text representations of spatial objects into types

```
PointFromText('POINT(1 3)')
LineStringFromText('LINESTRING (5 2,4 3)')
```

  - Function **AsText()** does the reverse – yields the text representation of a spatial object.

Spatial Database Example
Store Locator

- The database will store a table with
  - locations of the store
  - their geocoding, that is, their latitude and longitude
- User will specify an address near which stores are to be found.
  - address will be geocoded
  - geocoded address used to find the nearest stores.
- For example, geocoding the address of Lucent's headquarters at

600 Mountain Ave, Murray Hill, NJ 07020

- yields the latitude/longitude coordinates

40.68644, -74.40161

- In our example, for simplicity and convenience, we will use fictional *x* and *y* coordinates for the store locations
- Table Stores will be used to store the location information about Everest Books' stores

Spatial Database Example (Contd.)

```
CREATE TABLE Stores(
     Company VARCHAR(30) NOT NULL,
     Location POINT NOT NULL,
    Street VARCHAR(50),
     City VARCHAR(30),
     State2 CHAR(2),
     Zip CHAR(5),
    SPATIAL INDEX(Location)
) ENGINE = MyISAM;
```


Spatial Database Example (Contd.)

```
INSERT INTO Stores VALUES('Everest Books',
            PointFromText('POINT(1 3)'),
            '32 South Street','Delhi',
            'NY', '13201');
INSERT INTO Stores VALUES('Everest Books',
             PointFromText('POINT(11 34)'),
            '25 West Lane','Moscow',
            'NJ', '07201');
INSERT INTO Stores VALUES('Everest Books',
             PointFromText('POINT(0 2)'),
            '2 Second Street','London',
            'NJ', '07221');
```

Spatial Database Example (Contd.)

STORES

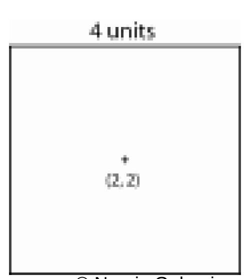
Company	Location	Street	City	State2	Zip
Everest Books	POINT(1 3)	32 South Street	Delhi	NY	13201
Everest Books	POINT(11 34)	25 West Lane	Moscow	NJ	07201
Everest Books	POINT(0 2)	2 Second Street	London	NJ	07221

Spatial Database Example (Contd.)

- List Everest Books stores within 2 units of coordinates 2, 2:

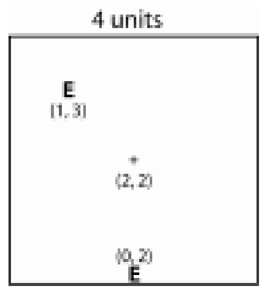
```
SELECT astext(Location) AS Location,
             Company, Street, City, State2, Zip
FROM Stores
WHERE Company = 'Everest Books' AND
       Distance(PointFromText('POINT(2 2)'),
                    Location) <= 2;
```

`Distance` has not been implemented in MySQL. So we will find stores within a square of side 4 and centered at 2, 2:



Spatial Database Example (Contd.)

```
SELECT astext(Location) as Location,
         Company, Street, City, State2, Zip
FROM Stores
WHERE Company = 'Everest Books' AND
   MBRContains(
     PolygonFromText('POLYGON((0 0,4 0,4 4,0 4,0 0))'),
     Location
   )
;
```



Security

- Data is often the most valuable asset of a corporation.
- New importance for security because of hackers and terrorism.
- Lax organizations → US Government mandating data privacy
  - HIPAA (health care)
  - Sarbanes-Oxley (corporate governance)
  - GLBA (financial services).
- Database security easier to handle when
  - databases could only be accessed from within the enterprise
  - number of users was limited.
- World has changed.
  - Databases can now be accessed from the Internet.
  - Although protected by firewalls, databases can be compromised.
- Data needs to be protected from unauthorized
  - disclosure of private information,
  - updates that are seemingly correct,
  - data corruption, and
  - deletion.
- Unauthorized reads & updates → enormous financial damage

Types of Security

- *Physical Security*
- *Network Security*
  - Prevent outsiders from breaking perimeter defenses.
  - Firewalls provide perimeter defenses.
- *User Authentication*
  - Ensure that a user is who he/she claims to be before allowing access to resources.
- *Connection Security*
  - Encryption protects content, during transmission, from eavesdroppers
  - Many databases provide support for encrypted interaction.
- *User Authorization*
  - To protect databases from unauthorized access, authorization ensures that users do only what they are allowed to do
  - Databases use privileges for this purpose.

Database Security Guidelines

- Database accessed by several users
  - decide what users can access what data
  - users can do what they like with their own data
  - restrictions on what users can do with others' data
- Enterprise security systems must authenticate user
- Suggestions for keeping data secure
  - Password protection
  - Only authorized users allowed to access the database
  - Users should not be given more privileges than warranted.
  - Sensitive tables & columns should be given names that mask their sensitive nature
  - Sensitive tables should be encrypted
  - Encrypt connections over the insecure public Internet
  - Limit the time a user is allowed to connect to the database
  - Limit the hours when a user can connect to the database
  - Restrict user access only from specific locations at specified times
- SQL provides support for implementing some of the above restrictions.
  - Others restrictions must be implemented by the enterprise infrastructure.

Security – Authorization

- Users must have authorizations to perform database actions.
  - determined by privileges granted.
- Examples of authorizations
  - read a table,
  - insert a row, delete a row.
  - create a table, drop a table,
- Most users are given limited set of authorizations.
  - One user, who typically has all authorizations, is the DBA.
- Users can pass authorizations to others, provided they have the authority.
  - Revoking privileges revokes privileges granted by the user to others
- Even in case of Everest Books database, we can benefit from controlling access. E.g.,
  - Employees whose sole task is to update the **`Books`** table should not be able to access the **`Customers`** table.
  - Only supervisors or persons handling returns should have permission to modify the **`Orders`** and **`OrderInfo`** tables.

Security - SQL

- Most databases used an id and password protection scheme.
  - A SQL *authorization id* identifiers a user and the user's privileges.
  - A SQL *privilege* authorizes a user to perform specific operations
    - Privileges are granted to users by the owner of the data, the DBA, and other users who have privileges that they can pass onto others.
- Authorization ids refer to users or roles.
  - Examples of roles are a customer care agent, a manager, or a sales person.
  - Associated with a role, as with a user, is a set of privileges.
  - Privileges can be associated with roles and granted to users as a bundle.

Security – SQL (Contd.)

- SQL does not provide facilities for creating & deleting users
  - MySQL provides such facilities.

Creating New Users

```
CREATE USER user
[IDENTIFIED BY PASSWORD 'password'];
```

Deleting Users

```
DROP USER user;
```

- Prior to deleting a user, all user's privileges must be revoked.
  - Note: The **SHOW GRANT** statement can be used to list user privileges and the **REVOKE** statement to delete user privileges.

Privileges

- The creator of a data item (e.g., table, view) has all the privileges with respect to the item and can grant privileges related to the item.
- Generally people granted a privilege for an item cannot pass the privilege on to other users
  - unless they have been granted the privilege with the **GRANT OPTION**.
- Privileges are granted to users based on the
  - needs expressed by the user and approved by someone in authority, and/or
  - role of the user in the organization.
- Privileges can be used to control user actions.

Some Privileges in SQL

Privilege	Allows users to
ALTER	modify table structure
CREATE	create tables
CREATE VIEW	create views
DELETE	delete rows
DROP	delete databases, tables, views
EXECUTE	execute stored procedures
GRANT OPTION	grant privileges to others
INDEX	create or delete an index
SELECT	select rows from tables
INSERT	insert rows
UPDATE	modify rows in tables
ALL	have all the "simple" privileges

Privileges (Contd.)

- The SQL model of privileges has the following four privilege levels:
  - ***Global***: Privileges apply to all databases.
  - ***Database***: Privileges apply to all tables in a database.
  - ***Table***: Privileges apply to all columns.
  - ***Column***: Privileges apply to specific columns in a table.
- For brevity, I shall restrict our discussion to granting privileges that apply to tables.

Granting Privileges

```
GRANT privileges
ON TABLE item
TO user
[WITH GRANT OPTION];
```

- Example

```
GRANT INSERT
ON TABLE Books
TO Aroon
WITH GRANT OPTION;
```

  - grants user **Aroon** the privilege of inserting rows in the table **Books**
  - allows **Aroon** to grant this privilege to others.

- The keyword **PUBLIC** denotes all users. E.g.,

```
GRANT ALL PRIVILEGES
ON TABLE Books
TO PUBLIC;
```

grants all privileges on table **Books** to all users.

Revoking Privileges

- The **REVOKE** statement takes back privileges from users.

```
REVOKE privileges
FROM user;
```

- As an example, the following statements

```
REVOKE ALL PRIVILEGES ON TABLE Books
FROM Aroon;

REVOKE GRANT OPTION FOR UPDATE
      ON TABLE Customers
FROM Aroon;
```

revoke all privileges on table **Books** and the **GRANT OPTION** from the table **Customers** from user **Aroon**.

- When privileges of a user are removed, any privileges passed on by the user to others are revoked

Roles

- A *role* represents a set of privileges
  - makes it convenient to grant users a bundle of privileges.
- SQL roles correspond to user roles in an organization.

Creating Roles

```
CREATE ROLE role;
```

A new role is initially empty.

- Example

```
CREATE ROLE Accountant;
```

Privileges and Roles

- Privileges are granted to roles much like they are granted to users

```
GRANT privileges
ON TABLE item
TO role
[WITH GRANT OPTION];
```


Roles (Contd.)

- Privileges are taken back from a role using the REVOKE statement:

```
REVOKE privileges
FROM role;
```

Granting & Revoking Roles

```
GRANT role
TO user;
```

- The statement

```
GRANT Accountant
TO Susan;
```

assigns role **`Accountant`** to **`Susan`** → gets all related privileges.

- Roles are revoked using the **`REVOKE`** statement.

Deleting Roles

```
DROP ROLE role;
```

Views – Security

- Views are an important data security tool
  - They can be used to limit the data that can be accessed by users (in conjunction with SQL privileges).
- Views can help restrict access to specific sets of rows.
- There is a cost to using views to implement security.
  - Virtual views require extra computation time
  - Materialized views require storage and maintenance.
  - Indexes on views will require both storage and processing time.
- Then there is the potential disadvantage that the views may not be updateable.

Views – Security Example

- On April 1, 2004, customers calling Everest Books experienced long delays.
- Everest Books decided to call all customers who placed an order on that date and apologize.
- They assigned this task to a few employees but they did not want to give them access to the complete database.
- They wanted to give them access to just the names and telephone numbers of the customers who placed an order on April 1, 2004.

```
CREATE VIEW ServiceProblems20040401 AS
SELECT First, Last, Tel
FROM Customers, Orders
WHERE id = CustomerId AND
    OrderDate = '2004-04-01';
```

Logs & Recovery

- To protect against loss of information, databases need to be backed up periodically.
- There are many kinds of failures that can affect databases
  - database server crash,
  - disk crash,
  - disk corruption, etc.
- In case of failure, what happens to the database and the updates made to it?
  - A backup copy helps a lot, but what about the updates that were made to the database after the backup copy was made?
  - What about partial updates that leave the database in an "inconsistent" state?

Failure Types

- Database recovery is the process of recovering a consistent database, from an inconsistent or unusable database, with minimum loss of information.
- Database failures can be classified into two categories:
  - soft, and
  - hard or catastrophic.
- Soft failures lead to an inconsistent database, which, given the right information, can be fixed.
- Catastrophic failures result in unusable databases, which require starting with a copy of the database.

Soft Failure

- A *soft failure* is caused by the incomplete or partial execution of a transaction that leaves the database in an inconsistent state.
- Examples of soft failures
  - computers running the database server or the client crashing, freezing, or stopping because of power failure,
  - database client crashing,
  - database server crashing,
  - network crashing,
  - network getting congested enough to block interactions with the database server making it think that the client has crashed,
  - application errors, and so on.
- To recover a consistent database
  - updates of completed transactions must be reflected in the database
  - changes of uncommitted transactions taken out of the database.

Hard Failure

- Hard or catastrophic failures are ones in which the database becomes unusable.
- The database cannot be used because the
  - media has been physically damaged or
  - data has been corrupted and is no longer meaningful (garbage).
- Examples of catastrophic failures
  - a disk crash,
  - the disk getting corrupted,
  - a virus infecting the database, and so on.
- In such a case, recovering the database requires starting with a copy of the database (if one exists) and bringing it up to date.

Logs

- As protection against failure, database systems record updates in a "table" called the *log*.
  - can be used to recreate a consistent and current database.
- Specifically, the log is used for *database recovery*
  - redo updates of committed transactions
  - undo updates by transactions that did not commit.
- To recover a database
  - Start with an empty database and begin applying updates of committed transactions (skip uncommitted transaction updates).
  - More efficient to start with a backup copy of the database than with an empty database.
  - Better yet, if the database is available but is inconsistent, then we can apply updates of committed transactions that did not get to the database and undo updates of uncommitted transactions.

Logs & Database Updates

- When a transaction updates a database
  - updates are first made to the database items in data buffer.
  - prior to making the updates on disk, a record of the updates is logged to disk.
- Specifically
  - Updates to database items are first made to database items stored in the data buffer (may require bringing items from disk to buffer).
  - Log entries corresponding to the updates made by transactions are first recorded in memory in the *log buffer*.
  - Contents of the log buffer are written to log (on disk) and the buffer flushed (emptied).
  - The data buffer's contents are written to the database (on disk).

Write-Ahead Logging (WAL) Protocol

- Recording database updates in the log (on disk) before modifying the database is known as the *write-ahead logging* (WAL) protocol.
- Two variants of WAL:
  - ***Deferred Update***: Updates are made to database only after the transaction has committed (after updates have been recorded in the log).
  - ***Immediate Update***: Updates are made to the database without waiting for the transaction to commit but after they have been recorded in log.
- The commit operation is made as small as possible
  - for speed
  - to minimize chances of a system crash during commit
- Consequently
  - all work of moving entries in the log buffer to disk is done beforehand, possibly in parallel with transaction execution.
  - if all log records relating to a transaction's execution are on disk, the commit operation consists of just updating a pointer on disk to make these records become part of the log.

Log Details

- Logs play a very important role in consistency and durability of a database (C & D parts of ACID transactions).
- A database log contains entries such as
  - Transaction id
  - Actions
    - ABORT,
    - BEGIN TRANSACTION,
    - DELETE,
    - INSERT, etc.
  - Disk location of database item.
- In case of
  - an update, both old and new values of item being updated are recorded;
  - an insertion, the new value is recorded;
  - a deletion, the old value is recorded.
- Old values used to undo effects of transactions
- New values used to reflect updates made by a transaction.
- Logs typically do not contain a record of the "read" components of transactions.
  - Not needed for database recovery.

Log Characteristics

- The complete log represents the record of the evolution of a database from day zero.
  - In practice, the complete log may not be available
- A log contains all the information contained in a database plus a lot more.
- It is also a temporal database, tracking the evolution of the database, who made what updates and when, how long queries took to execute, etc.
- Logs can be used for:
  - ***Database Replication***
  - ***Database Audit***
  - ***Performance Analysis***
- A log can be "viewed" as an append only table.
  - Only the database system is allowed to append rows to the log.
- The log is kept on disk for durability.

Automatic Recovery Restart

- Most database systems, when restarted, check to see if the database server shut down normally.
- If not, they assume that the database is potentially inconsistent.
  - An automatic recovery process is initiated

```
C: > mysqld
InnoDB: Database was not shut down normally.
InnoDB: Starting recovery from log files...
InnoDB: Starting log scan based on checkpoint at log sequence number 0
14674032
InnoDB: Doing recovery: scanned up to log sequence number 0 14839521
InnoDB: Doing recovery: scanned up to log sequence number 0 14905056
...
InnoDB: Doing recovery: scanned up to log sequence number 0 21555224
InnoDB: 1 uncommitted transaction(s) which must be rolled back
InnoDB: Starting rollback of uncommitted transactions
InnoDB: Rolling back transaction number 26745
InnoDB: Rolling back of transaction number 26745 completed
InnoDB: Rollback of uncommitted transactions completed
InnoDB: Starting an apply batch of log records to the database...
InnoDB: Apply batch completed
InnoDB: Started
mysqld: ready for connections
```

Note: InnoDB uses the immediate update variant of WAL protocol for database updates.

Recovery Strategy

- Recovery strategy used depends upon
  - the nature of failure
  - variant of WAL protocol.
- Soft failure
  - Log is used to take the database to a consistent state.
  - Updates made by committed transactions are applied (*redo*).
  - If the immediate update strategy is used, then updates of uncommitted transactions are undone (*undo*).
- Catastrophic failure
  - latest backup used as a starting point.
  - log used to take database to a current and consistent state.
- To recover from an inconsistent database, *database recovery system* needs to know which transactions have been applied to
  - apply updates of remaining committed transactions
  - undo updates of uncommitted transactions.
- Databases "checkpointed" to speedup recovery
  - checkpoint entry in log indicates all updates of committed transactions have been written to disk (at checkpoint).
  - need only consider transactions that committed or aborted after most recent checkpoint.

Backups

- In case of catastrophic failure, log can be used to recover the database.
  - requires complete log.
  - may take a long time because log may be very large.
- Backups can be used to make recovery process faster and more efficient
- Log entries needed for recovery need only reflect updates made after the backup was
  - made in case of an *offline* backup, or
  - started in case of an *online* backup.
- The more recent the backup copy, the less time it will take to recover the database.
- Logs should also be backed up to protect against catastrophic failure.
  - Logs are also backed up along with the database.
  - log truncated (emptied).
- Backups should be kept on another disk or media – different from database and the log.

Types of Backups

Offline Or Cold Backup

- Made after taking the database offline, i.e., stopping all updates to the database.
- Reflects a consistent view of the database state at a point in time.

Online Or Hot Backup

- Made while allowing transactions to concurrently update database.
- Does not reflect a consistent state of the database – since some updates made by a transaction may not make it to the backup copy.
- The backup database copy represents an approximation of the database as it changed while the backup was being made.
- Nevertheless, with the information in the log, a consistent and current version of the database can be created using the backup.

Recovery Operations

- 3 major operations in database recovery: redo, undo, & checkpoint.

Redo Operation

- A *redo* reapplies an update to the database.
  - Log entries contain new values of items.
  - Used to ensure that updates of committed transactions are reflected in the database.
- Recovering a database after catastrophic failure involves starting with a backup copy.
  - Updates following the backup are applied to the backup.
  - Redo operations are applied in the forward log direction.
- Similarly, when recovering from an inconsistent database, redos are applied in forward log direction.
- The database system does not guarantee a specific commit order for concurrently executing transactions.
  - Only guarantees that the commit order will lead to a consistent database.
  - Different commit orders may lead to different consistent databases.
  - To ensure recovery of a consistent database matching the database had there been no failure, updates must be applied in the commit order noted in log.

Recovery Operations (Contd.)

Undo Operation

- An *undo* operation undoes an update made by a transaction.
  - Undo operations are used to reverse updates made by transactions that did not commit
  - Such updates happen with the immediate update strategy.
  - Transactions may not commit because they were aborted or because of soft failures.
- Updates made by transactions that do not commit make a database inconsistent.
- By undoing the updates of such transactions, an inconsistent database can be brought to a consistent state.
- To reverse a database update, the corresponding log entry needs to contain the original value.
  - Undo operations are applied in the reverse of the order in which the updates were made – as noted in the log.

Recovery Operations (Contd.)

Checkpoints

- To recover from an inconsistent database, the database recovery system scans the log to
  - apply updates of committed transactions
  - undo updates of uncommitted transactions.
  - determine which transactions have committed and their updates applied; then it can skip part of log corresponding to these transactions.
- To speed up recovery, database systems use *checkpointing* to tell the recovery system how much of the log it need examine.
  - Checkpointing is forcing updates of committed transactions to be written to the database (on disk).
  - Databases periodically checkpoint
  - A checkpoint entry is written to the log.
- The recovery system can start at most recent checkpoint, since updates of transactions committing before the checkpoint are guaranteed to have been applied to the database.
- Checkpoint entries can be used to record additional information to help the recovery manger, e.g.,
  - a list of uncommitted transactions at checkpoint time.

Recovery From Soft Failure

Immediate Update WAL Protocol

- Recovering from an inconsistent database
  - The most recent checkpoint is located by backward scanning the log.
  - While scanning backwards, lists of committed & uncommitted transactions are built – transactions that updated the database after the checkpoint.
  - To the list of uncommitted transactions are added uncommitted transactions specified in the checkpoint but not in the list of uncommitted transactions.
  - Starting from the checkpoint entry, the log is forward scanned while applying the updates of the committed transactions to the database.
  - The log is backward scanned to undo updates that might have been made in the database by the uncommitted transactions.
- The end result is a database that is both consistent and current.

Deferred Update WAL Protocol

- Similar to above except that there is no need to worry about uncommitted transactions.

Recovery From Catastrophic Failure

Starting With A Cold Backup Copy

- Backup copy represents a consistent snapshot of database.
- To get a current database,
  - start with the backup copy
  - apply all updates made by all committed transactions after the backup was made.

Recovery From Catastrophic Failure

Starting With A Hot Backup Copy

- The backup copy is a fuzzy inconsistent snapshot of the database
  - During backup transactions were updating the database.
  - No guarantee that all updates of these transactions were included in backup.
- To bring the backup copy to a consistent state
  - updates of transactions that committed after backup was started must be applied to database
  - updates of transactions that did not commit must be undone.
- To recover the database, the database recovery system takes the following actions:
  - Scans the log backwards to the most recent checkpoint in the log before the start of the backup.
  - Starting from this checkpoint the recovery steps are the same as in the case of recovery from soft failure when the immediate update WAL protocol is used.

Database Recovery Example

- We will assume that the database system uses the deferred update WAL protocol
  - the database will not contain any updates made by uncommitted transactions.
- The database will be recovered by applying, from the log, the updates of committed transactions.
- Consider the following two tables, **`Orders`** and **`OrderInfo`**, in the Everest Books database:

Tables Orders & OrderInfo

OrderId	CustomerId	OrderDate	ShipDate	Shipping	SalesTax
1	1	2004-03-31	2004-03-31	4.99	0.00
2	1	2004-04-01	2004-04-02	5.99	0.00
3	2	2004-04-01	2004-04-02	3.99	0.00
4	3	2004-04-02	2004-04-02	6.99	0.00

OrderId	ISBN	Qty	Price
1	0929306279	1	29.95
1	0929306260	1	49.95
2	0439357624	3	16.95
3	0670031844	1	34.95
4	0929306279	1	29.95
4	0929306260	1	49.95
4	0439357624	1	16.95
4	0670031844	1	34.95

Updates Before Crash

- Transactions T1, T2, T3, & T4, arrive at database server soon after which it crashes:
  - T1 generates an order for a copy of *White Moguls* for customer with id 2.
  - T2 generates an order for a copy of *White Moguls* for customer with id 3.
  - T3 updates the order for customer with id 2 to two copies of *White Moguls* (shipping charge changed to $4.99).
  - T4 generates an order for a copy *White Moguls* for customer with id 1.
- When server crashes:
  - T1 committed and its updates propagated to the database.
  - T2 committed but its updates do not make it to the database.
  - T3 committed but its updates do not make it to the database.
  - T4 does not get a chance to commit.
- Committing → all updates made by a transaction are in the log + a transaction committed entry.
  - Updates made by committed transactions to copies of tables in the data buffer may not have been written to database.

Orders – Just Before Crash

- Only T1's updates to the **`Orders`** table are reflected in the database (shown in gray):

ORDERS (JUST BEFORE SERVER CRASH)

OrderId	CustomerId	OrderDate	ShipDate	Shipping	Sales Tax
1	1	2004-03-31	2004-03-31	4.99	0.06
2	1	2004-04-01	2004-04-02	5.99	0.00
3	2	2004-04-01	2004-04-02	3.99	0.00
4	3	2004-04-02	2004-04-02	6.99	0.00
5	2	2004-05-03	2004-05-03	3.99	0.00

Updates to Orders – in Data Buffer

- Updates made by transactions T2, T3, and T4, to the data buffer copy of the **`Orders`** table (updates shown below) do not get propagated to the database:

UPDATES TO ORDERS (IN THE DATA BUFFER)

OrderId	CustomerId	OrderDate	ShipDate	Shipping	Sales Tax
6	3	2004-05-03	2004-05-03	3.99	0.00
5	2	2004-05-03	2004-05-03	4.99	0.00
7	1	2004-05-03	2004-05-03	3.99	0.00

OrderInfo Just Before Crash

- Transaction T1's updates to the **`OrderInfo`** table are reflected in the database (row shown in gray):

ORDERINFO (JUST BEFORE SERVER CRASH)

OrderId	ISBN	Qty	Price
1	929306279	1	29.95
1	929306260	1	49.95
2	439357624	3	16.95
3	670031844	1	34.95
4	929306279	1	29.95
4	929306260	1	49.95
4	439357624	1	16.95
4	670031844	1	34.95
5	670031844	1	34.95

Updates to OrderInfo – in Data Buffer

- Updates made by other transactions to the copy of the **`OrderInfo`** table in the data buffer (updates shown below) do not get to propagate to the database.

UPDATES TO ORDERINFO (IN THE DATA BUFFER)

OrderId	ISBN	Qty	Price
6	670031844	1	34.95
5	670031844	2	34.95
7	670031844	1	34.95

Log At Time of System Crash

LOG AT SYSTEM CRASH TIME

Trans ID	Action	Item Updated	Before Value / Other Info	After Value
...				
T1	BEGIN TRANS			
T1	INSERT	Orders	<5,2,2004-05-03,2004-05-08,3.99,0.00>	
T1	INSERT	OrderInfo	<5,670031844,1,24.95>	
T1	COMMIT			
T2	BEGIN TRANS			
T2	INSERT	Orders	<6,3,2004-05-03,2004-05-08,3.99,0.00>	
	CHECK POINT		T2 is active	
T2	INSERT	OrderInfo	<6,670031844,1,24.95>	
T2	COMMIT			
T3	BEGIN TRANS			
T3	WRITE	OrderInfo	<5,670031844,1,24.95>	<5,670031844,2,34.95>
T3	WRITE	Orders	<5,2,2004-05-03,2004-05-08,3.99,0.00>	<5,2,2004-05-03,2004-05-03,4.99,0.00>
T3	COMMIT			
T4	BEGIN TRANS			
T4	INSERT	Orders	<7,1,2004-05-03,2004-05-08,3.99,0.00>	

Trying To Recover

- Log entries for 4 transactions are shown as contiguous for ease of reading.
  - In general, they may be interleaved.
- Updates made by T2 and T3 will appear in the recovered database, because they committed before the server crashed.
- Some updates made by T4 were recorded in the log but T4 did not get to commit.
  - None of its updates will appear in the recovered database.
- Updates made by T1 were already in the database at the point of the crash.
  - These updates will be updated by T3.
- To restore database to a consistent state
  - Log is scanned backward to the most recent checkpoint – while doing so, a list of transactions that committed after the checkpoint is built.
  - Then log is scanned forward and updates made by the committed transactions are applied to the database.

After Recovery

ORDERS (AFTER RECOVERY)

OrderId	CustomerId	OrderDate	Ship Date	Shipping	Sales Tax
1	1	2004-03-31	2004-03-31	4.99	0.06
2	1	2004-04-01	2004-04-02	5.99	0.00
3	2	2004-04-01	2004-04-02	3.99	0.00
4	3	2004-04-02	2004-04-02	6.99	0.00
5	2	2004-05-03	2004-05-03	4.99	0.00
6	3	2004-05-03	2004-05-03	3.99	0.00

After Recovery (Contd.)

ORDERINFO (AFTER RECOVERY)

OrderId	ISBN	Qty	Price
1	929306279	1	29.95
1	929306260	1	49.95
2	439357624	3	16.95
3	670031844	1	34.95
4	929306279	1	29.95
4	929306260	1	49.95
4	439357624	1	16.95
4	670031844	1	34.95
5	670031844	2	34.95
6	670031844	1	34.95

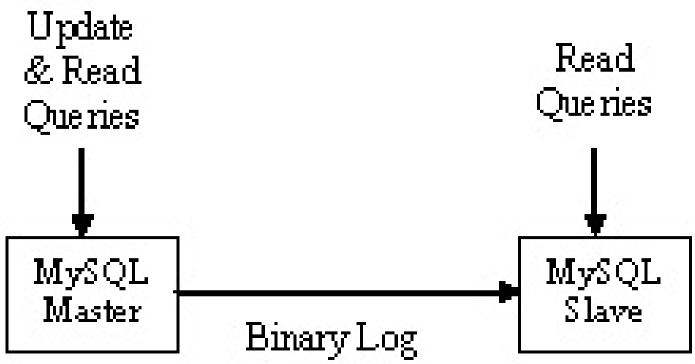
Replication

- Database replication is the creation and managing of duplicate copies (replicas) of a database, all of which are available to users.
- Updates made to one database copy are automatically propagated to all the other replicas.
- In some database replication models, one of the replicas is designated as the *master* and all updates are directed to it.
  – Read queries can be directed to the master or to the other replicas.
- Here are some uses of database replication:
  – *Availability*: One replica can be used as a “hot” spare.
  – *Load Balancing*: Replicas are extremely valuable for load balancing heavily used databases.
  – *Proximity*: Replication allows the placement of databases closer to users and applications thus speeding up response time.
  – *Security*: Hackers may not be able to get to all the replicas, especially if some have extremely restricted access.
  – *Backups*: Replicas can be used as backups.

Replication (Contd.)

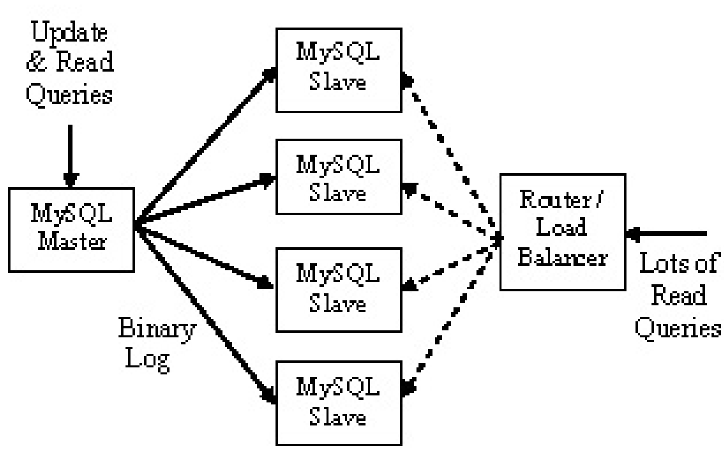
- MySQL (version 5.0) supports one-way, asynchronous replication, in which one database server (replica) acts as the master while others act as slaves.
- MySQL replication is based on the master database server recording all updates in binary logs.
  – The replicas connect to the master to read the updates from its binary log, and then run the updates to catch up with the master.
  – To avoid conflicts, update queries are directed to the master while read queries can go to the master or the slaves.
- In case of problems with the master, the slaves can be used and one of them designated as the master.
- I will now show you briefly describe two configurations. One is used primarily for database availability and the other for load balancing.

Replication for Availability



© Narain Gehani
Introduction to Databases Slide 167

Replication for Load Balancing



© Narain Gehani
Introduction to Databases Slide 168

Tuning

- Database performance is measured in terms of benchmarks such as
  - *response time:* the time it takes a user to get the result of a query, and
  - *throughput:* the number of queries a database can handle per second.
- Database performance becomes an important issue in the presence of items such as
  - large amounts of data,
  - complex queries,
  - queries manipulating large amounts of data,
  - long queries,
  - queries that lock every one else out,
  - large numbers of users, and
  - limited bandwidth.

Tuning (contd.)

- Database *tuning* is the process minimizing response time and maximizing throughput. It impacts
  - table design, index design, and other aspects of schema design;
  - how queries are written, the order of their execution, and determining what data processing is done by the database and the client; and
  - determining how and when tables and indexes are loaded into memory.
- Database systems are designed to give the best performance in general.
  - Database system implementers do not have information about specific databases and their use.
  - Tuning refers to maximizing performance in the design and deployment of specific databases.
- The best improvements can be achieved in the design phase.
  - Altering a large database in use can be expensive and practical considerations may put constraints on what can be changed.

Tuning (contd.)

- Tuning can be difference between a query taking milliseconds or minutes to execute.
- Suppose that **`Books`** contains information about 3.5 million books. Now consider the query:

```
SELECT Title
FROM Books
WHERE Author = 'Gehani';
```

- Without an index on **`Author`**, the database system may, in the worst case, have to read 3.5 million rows.
- Suppose it takes 1ms to read a disk block and each block contains 10 rows.
  - Reading the whole **`Books`** table will take about 350 seconds!
- Suppose that there is a B^+-tree index for the column **`Author`**.
  - The number of disk blocks read in the worst case will be about $\log_b(3.5 \times 105)+1$
  - *b* is the maximum number of subtrees of each node.
  - If *b* is 32, then the number of disk blocks read will be about 5.
- If queries using column **`Author`** in a filtering condition are going to be frequent, then **`Author`** should be indexed.
- In this simple example, tuning involves determining whether or not an index is necessary for column **`Author`**.

System Characteristics

- Performance is influenced by characteristics of the system where the database server is hosted:
- Disk input/output (I/O) speed.
- Amount of memory available.
- CPU speed.
- Network bandwidth.

Disk I/O Speed

- Data is stored on disk for persistence
  - before the data can be processed, it must be brought to memory.
- The unit of disk I/O is a *block*.
  - An example block size is 16K bytes.
  - Even if the data requested occupies a small portion of a block, the entire block is retrieved.
- Disk accesses are very slow compared to memory operations.
  - Database performance depends heavily on disk access speed.
  - In addition to using faster disks, database performance can be improved by minimizing disk accesses.
  - Once data is brought to memory (data buffer), keeping it there for future use will save on disk accesses and lead to fast query execution.
  - Larger data buffers → more data can be kept in memory
  - But memory is much more expensive than disk.

Why are Disks Slow?

- Disks are mechanical devices and this is what makes them slow as compared to memory which is an electronic device.
  - Data is stored on *tracks* on CD-like platters, which rotate on a spindle and make up a disk.
  - For each platter, there is an *arm* with a *head* that moves across tracks on the platter reading data from or writing data on them.
- The time to read/write data consists of three components:
  - *Seek Time*: Time taken by the disk head to move to the appropriate track.
  - *Rotation Time*: Time taken by the platter to rotate so that the appropriate block on the track is under the disk head.
  - *Transfer Time*: Time to read or write the data block.

  Seek time is greater than rotational time which is greater than transfer time.

Optimizing Disk Accesses

- Data that is likely to be accessed together should be stored contiguously in one block or, if more space is needed, then in consecutive blocks on a disk platter.
  - Doing this will reduce seek and rotational times.
  - If an index can fit in one block, then reading it will be very fast.
  - The execution time of a query, such as a **`SELECT`** statement without a **`WHERE`** clause, that needs to access a complete table will be minimized if the table is stored in contiguous blocks.
- Database systems optimize placement of data on disk by storing tables, indexes, and other database items in a manner that minimizes seek and rotational times.
  - Rows inserted into a table are stored contiguously, in a block with other rows or in adjacent blocks, as appropriate.
  - Such decisions are made by database systems and are transparent from a user perspective.

Optimizing Disk Accesses (contd.)

- The unit of disk I/O is a disk block.
  - The size of the block can be explicitly specified by a database designer or a DBA, overriding the default size.
  - A typical block size ranges from 2K to 16K bytes.
  - The block size is specified when a database is created and cannot be altered later.
- Large block sizes reduce the ratio of bytes read or written per seek, which is good since seeks are relatively very expensive.
  - But large blocks might not be optimal for applications with updates involving several blocks, because each update will take longer.
  - Larger the block, the more expensive is writing a block to disk because more data must be written.
  - For most common applications, the default block size should work well.
- In case of databases with special needs, it may be advantageous to explicitly specify alternative block sizes.
  - For example, multimedia databases, which have large items, large blocks sizes should be used to reduce the number of items needing than one disk block for storage.

Memory

- To process a query, the needed data must be in memory (data buffer).
- The database system checks to see whether or not the data is in the buffer.
  - If yes, then disk accesses are not required.
  - If no, then disk blocks containing the data are read from disk.
  - The larger the data buffer, the larger the probability of finding the needed data in the data buffer.
- If data buffer does not have enough space for new items, then some items must be deleted.
  - If these items have been updated and new values not reflected on disk, then they must be written to disk before deletion.
  - To determine items to be deleted, database systems use LRU, FIFO, etc. algorithms.
  - Note that the data buffer is similar to an operating system cache.
- The DBA has the primary responsibility of determining and specifying memory needs for optimal performance.
  - The computer system running the database server should have enough memory.
  - To tune performance, the DBA must determine and specify sizes for the different database buffers.
- In MySQL, it is possible to specify sizes of buffers for items such as
  - the index,
  - joins,
  - data,
  - caching the query, and
  - caching query results.

CPU Speed

- Faster CPUs are always a plus.
- Typically, disks are likely to be the cause of performance problems because of slow I/O.
- However, if there are multiple disks and I/O is being performed in parallel, then CPUs can be the bottleneck.
  - In such cases, faster CPUs can help speed up response time and improve throughput.

Network Bandwidth

- In a client-server architecture, one of the factors affecting response time is available bandwidth.
- Typically, in case of intranets and broadband networks, network bandwidth is usually not an issue.
- However, even in such cases, transferring large amounts of data between the client and the database server can be time consuming, especially if the transfer involves multiple interactions.

Improving Application Design

- A SQL statement or transaction can be written in multiple different ways to perform the same task.
- Database systems rewrite SQL code to improve performance – *query optimization*.
  - query optimizer uses information such as table sizes, indexes available, & distribution of data.
- To ensure good performance, an application writer should understand
  - the database,
  - how it will be used, and
  - the nature and frequency of read and insert/delete/update operations.
- An application writer should avoid queries requiring more processing time than others
  - Range queries can be slower than exact match queries.
  - Join and nested queries and queries that use clauses such as **`DISTINCT`** and **`ORDER BY`** can be time expensive.
- The database designer should be cognizant of system and user characteristics.
  - A large number of users can slow response time if
    - the system does not have enough memory and/or CPU cycles, or
    - the transactions conflict a lot.

Improving Application Design
Joins

- Joins are expensive.
  - Conceptually in a join, a row from one table, the first table, is read and matched with each row in the second table, then the same happens with the 2nd row of the first table, and so on.
  - This means that all rows in the two tables have to be read.
  - Join queries often have filtering conditions that restrict rows that can participate in the join.
  - Indexes can be used to skip reading unwanted rows.
- The first table in the above join description is called the *driving* table*.*
  - Statistics are used to determine the driving table.
- The following query finds all customers whose orders were shipped in April 2004:

```
SELECT First, Last
FROM Orders, Customers
WHERE ShipDate > '2004-04-01'
  AND ShipDate < '2004-04-31'
  AND CustomerId = Id;
```

- The driving table should be the smallest table as stored in the database or after rows have been eliminated by the filtering conditions.
- The shipping date filter reduces of rows of **`Orders`** that need to be considered in the join
- **`Orders`** could be the driving table
  - provided that the reduced number of rows of **`Orders`** is less than the total number of rows in **`Customers`**.

Improving Application Design
Joins (contd.)

- Consider the following query :

```
SELECT First, Last
FROM Orders, Customers
WHERE State2 = 'NJ' AND CustomerId = Id;
```

- The filter condition

```
State2 = 'NJ'
```

  reduces the rows of **`Customers`** to be considered for the join.
- **`Customers`** could be the driving table (provided the reduced number of rows is less than the number of rows in **`Orders`**).
- Now consider the following query that lists the names of all NJ customers whose orders were shipped in April 2004:

```
SELECT First, Last
FROM Orders, Customers
WHERE ShipDate > '2004-04-01'
  AND Shipdate < '2004-04-31'
  AND State2 = 'NJ'
  AND CustomerId = Id;
```

- This query has filtering conditions for both **`Customers`** and **`Orders`** tables.
  - The driving table will be determined by the *selectivity* of the filter conditions and table sizes

Improving Application Design
Nested Queries

- Consider the following two equivalent queries, one a join and the other a nested query. The join version:

```
SELECT DISTINCT Title
FROM Books, OrderInfo
WHERE Books.ISBN = OrderInfo.ISBN;
```

- The nested query:

```
SELECT DISTINCT Title
FROM Books
WHERE ISBN IN (SELECT ISBN FROM OrderInfo);
```

- Database systems typically optimize joins well. The output of

```
EXPLAIN
  SELECT DISTINCT Title
  FROM Books, OrderInfo
  WHERE Books.ISBN = OrderInfo.ISBN;
```

  indicates that MySQL will use the index for **ISBN** in **Books** to execute the join query.
- On the other hand, in case of the nested query, the **EXPLAIN** statement indicates that MySQL will not use an index.

Improving Application Design
Eliminating Joins

- Consider the following query that lists the names of customers and the titles of the books ordered by them and were shipped in April 2004:

```
SELECT First, Last, Title
FROM Customers, Orders, OrderInfo, Books
WHERE ShipDate > '2004-04-01'
  AND ShipDate < '2004-04-31'AND Id = CustomerId
  AND Orders.orderId = OrderInfo.OrderId
  AND OrderInfo.ISBN = Books.ISBN;
```

- Assuming that table Orders is selected as the driving table and that the joins are executed in order written, the 4 tables will be joined as follows:
  - Rows in Orders that satisfy the filter are joined with the rows in Customers based on matching customer ids.
  - Rows from the join of Orders and Customers are joined with the rows of OrderInfo based on matching order ids.
  - Rows from the join of Orders, Customers, and OrderInfo are joined with rows of Books based on matching ISBNs.

Improving Application Design
Eliminating Joins (contd.)

- Sometimes, database systems may not do a good job optimizing. Writing a query differently may give better performance. Consider the query:

```
SELECT Title
FROM Customers, Orders, OrderInfo, Books
WHERE ShipDate > '2004-04-01‘ AND ShipDate < '2004-04-31'
  AND Last = 'Jones‘ AND CustomerId = Id
  AND Orders.OrderId = OrderInfo.OrderId
  AND OrderInfo.ISBN = Books.ISBN;
```

- Joins are expensive. Rewrite query as 2 statements, replacing 1 join with a **`SELECT`**.
- First statement finds id of Jones and stores it in **`BuyerId`**.

```
SELECT @BuyerId := Id
FROM Customers
WHERE Customers.Last = 'Jones';
```

- We will now be able to avoid one of the joins:

```
SELECT Title
FROM Orders, OrderInfo, Books
WHERE ShipDate > '2004-04-01‘ AND ShipDate < '2004-04-31'
  AND CustomerId = @BuyerId
  AND Orders.OrderId = OrderInfo.OrderId
  AND OrderInfo.ISBN = Books.ISBN;
```


Using Non-Normalized Tables

- To improve performance databases are sometimes designed to store redundant data.
  - Lose advantages of normalized tables such as avoiding data consistency problems.
- Consider the following 3-join query:

```
SELECT Title
FROM Customers, Orders, OrderInfo, Books
WHERE Last = 'Jones' AND Id = CustomerId
  AND Orders.OrderId = OrderInfo.OrderId
  AND OrderInfo.ISBN = Books.ISBN;
```

- The last join matches rows with the same ISBNs in **`OrderInfo`** and in **`Books`** so that the titles of the books ordered can be listed.
- This join would be unnecessary if **`OrderInfo`** contained book titles.
- Moving in this direction, we add a new column **`Title2`** to **`OrderInfo`** and call the modified table **`OrderInfo2`**.

Using Non-Normalized Tables (contd.)

- The previous query can now be written with one less join:

```
SELECT Title2
FROM Customers, Orders, OrderInfo2
WHERE Last = 'Jones'
  AND Id = CustomerId
  AND Orders.OrderId = OrderInfo2.OrderId;
```

- The 2nd join matches customer ids in rows of **Orders** & **OrderInfo2** so that we retain only rows of customers who ordered books.
- If these ids were stored in **OrderInfo2**, then this join would not be needed.
- Storing customer ids in the new column **CustomerId3**, the above query can be written using only one join as:

```
SELECT Title2
FROM Customers, OrderInfo3
WHERE Last = 'Jones'
AND Id = CustomerId3;
```


Storing Summary Information To Improve Performance

- The following query determines how much each customer purchase totals:

```
SELECT First, Last, Company,
   SUM(OrderInfo.Qty*OrderInfo.Price) AS Purchases
FROM Customers, Orders, OrderInfo
WHERE Id = CustomerId
  AND Orders.OrderId = OrderInfo.OrderId
GROUP BY Id, First, Last, Company
ORDER BY Purchases DESC;
```

  Query may be slow to execute because of the joins.
- `Customers` is modified to include purchase dollars, precomputed and updated with every new order and order changes.
  - Column `Purchases` in modified `Customers`, called `CustomersWithSummaryInfo`, will store purchase dollars.
- This information can now be retrieved with the query:

```
SELECT First, Last, Company, Purchases
FROM CustomersWithSummaryInfo
ORDER BY Purchases DESC;
```

- This query will be faster since it does not have joins or aggregations.

Materialized Views To Improve Performance

- Using materialized views to store data computed by frequently executed queries can be advantageous, in the context of large tables & much disk I/O.
- Suppose summary information about past sales is queried frequently and the queries are expensive because they involve joins of very large tables.
- To improve performance, sales data is precomputed using materialized views.
- An example is the view `Sales2004` (shown earlier):

```
CREATE VIEW Sales2004 AS
SELECT OrderInfo.ISBN, Title,
  SUM(OrderInfo.Qty)AS CopiesSold,
  SUM(OrderInfo.Qty*OrderInfo.Price)AS Sales
FROM OrderInfo, Orders, Books
WHERE OrderInfo.ISBN = Books.ISBN
  AND OrderInfo.OrderId = Orders.OrderId
  AND ShipDate >= '2004-01-01'
  AND ShipDate <= '2004-12-31'
GROUP BY OrderInfo.ISBN, Title;
```

- If materialized views are not supported, then tables can be used.
  - must be maintained manually.
  - Past years' sales data not likely to change.

Transactions & Performance

- In some cases, performance can be enhanced by using smaller transactions.
  - Splitting long or long-running transactions into smaller ones may
    - reduce chances of deadlock
    - increase concurrency.
  - Consider an Everest Books transaction which records
    - the information in the database (if it does not exist), and
    - order information in the database.
  - Splitting this transaction into 2 will enhance concurrency
    - `Customers` table will not be locked while `Orders` & `OrderInfo` tables are being updated & vice versa.
- In some cases, it may be better to combine several small transactions so that disk writes at the end of each transaction to commit them are combined.
  - Suppose a customer is ordering multiple books.
  - Instead of a separate transaction to record order information for each book, it would be efficient to use one transaction for all the books.
- Using non-locking reads gives better performance provided approximate answers are acceptable.

Indexing

- Using indexes can drastically reduce database access time.
- But indexes are not a magic solution that always work
  - require maintenance, processing time, disk accesses, and storage.
- Decision to index will depends on the applications.
- Consider, for example, the query:

```
SELECT Title
FROM Books
WHERE Authors LIKE 'Ge%';
```

- If result contains a small number, say 5%, of rows in **Books**, then an index may be beneficial.
  - If number is large, say 75%, then an index may not help.
- As a table grows larger, the benefit of using an index increases.
- The breakeven point for using an index, also depends upon the query result size, query frequencies & need for fast responses.
- To facilitate decisions about performance, database systems collect statistics about data distribution, etc. and to determine query selectivity and result size.
- In MySQL, the **ANALYZE TABLE** statement yields information about data distribution.
- Indexes incur a maintenance overhead even if not used. Such Indexes should be removed.

Type of Index

- Two popular indexes commonly found in database systems are B-tree and hash indexes.
  - Hash indexes are very fast for equality searches.
  - B-tree indexes, on the other hand, perform well for both equality and range queries (not as fast as a hash index for equality searches).
- Consider the following query:

```
SELECT *
FROM Books
WHERE ISBN = '0929306279';
```

- If there are going to be many such queries, then a hash index on column **`ISBN`** would be an appropriate index to create and use.
- But for range queries such as

```
SELECT *
FROM Books
WHERE PubDate > 2000;
```

- it would be best to use a B-tree index on the column **`PubDate`**.

What to Index?

- Analyzing & understanding frequently executed queries can help decide on indexes.
- For example, assume queries such as

```
SELECT Title
FROM Books, OrderInfo
WHERE Books.ISBN = OrderInfo.ISBN
AND Authors = 'Gehani';
```

  are executed frequently.

- A database system will try to minimize the number of rows involved in a join.
  - Assuming **Books** is larger than **OrderInfo**, the database system will first select rows from **Books** that satisfy
    Authors = 'Gehani'
    especially, it there is an index for **Authors**.
  - ISBN numbers of filtered rows will be matched with those of rows in **OrderInfo**.
    - An index on the **ISBN** column of the **OrderInfo** table will help select matching rows.
  - Thus having indexes for **Authors** in **Books** and **ISBN** in **OrderInfo** will be advantageous.

Query Execution Plan

- Each query is executed according to a plan, selected by the database system, that maximizes performance.
- Plan selection depends on factors such as the indexes available and selectivity of the filter conditions in the query.
- Database systems provide tools that show how a query will be executed.
  - An application writer can modify the query, define a new index, or modify the schema to improve performance.
- In MySQL, the **EXPLAIN** statement is used to see how a query will be executed. For example:

```
EXPLAIN
   SELECT Title
   FROM Books, OrderInfo
   WHERE Books.ISBN = OrderInfo.ISBN
     AND Books.Authors = 'Gehani';
```

  produces a description of the how the specified **SELECT** statement will be executed.

Client-Server Interactions

- All users (clients) will expect a good response time.
- E.g., Everest Books customers and staff accessing & manipulating the database simultaneously would not like to be kept waiting.
- With a web interface, the client is the web server since it, and not the user, interacts with the database.
  - Web server executes code – the database application – that interacts with a database server.
- Some factors in the interactions between the client & database server affecting performance:
  - *Data Processing at the Client or the Database Server*
    - After data has been retrieved from the database, additional processing may need to be performed.
    - If done at the database server, then this may slow the server.
    - If done at the client, then transmitting the data over the network can take time.
  - *Connections to the Database Server*
    - Instead of opening a new connection for each interaction, one connection should be kept open for the whole session.
    - Opening and closing connections is expensive.
  - *Data Transfer*
    - Fetching rows individually from the database will be slower than fetching groups of rows.

Is Your Database Truly Relational?

1. ***Information Specification***: All information in a relational database is logically specified in exactly one way – as values in tables.
   - Storage and layout on disk are not specified.
2. ***Guaranteed Access***: Each and every value in a table in a relational database is guaranteed to be logically accessible by specifying the table name, the primary key value, and the column name.
3. ***Systematic Treatment of Null Values****:* Null values, which are distinct from "zero" values such as the empty character string or a zero must be supported for representing missing or inapplicable information, and independent of the data type.
4. ***Catalog Based on the Relational Database Model***: The data description, i.e., the catalog, is represented logically like ordinary data, so that authorized users can use the same relational language to query both catalog and ordinary data.

Is Your Database Truly Relational? (contd.)

5. ***Comprehensive Data Language***: Must support at least one language whose statements are expressible as text and provides the following capabilities:
   - Data definition.
   - View definition.
   - Data manipulation (interactively and via a program)
   - Integrity constraints.
   - Authorization.
   - Transactions
6. ***View Updates:*** All theoretically updateable views should be allowed.
7. ***High-Level Insert, Update, and Delete***: The ability to specify a base or derived table as a single operand applies not only to data retrieval but also to data insertion, update, and deletion.
8. ***Physical Data Independence***: The logical view of the database is not affected by changes made to either storage representation or access methods. Thus such changes do not affect applications or user interaction with the database.

Is Your Database Truly Relational? (contd.)

9. ***Logical Data Independence***: Applications and user interaction are logically unaffected when theoretically permitted information-preserving changes are made to the base tables.
10. ***Integrity Independence***. Integrity constraints must be definable in the relational data language and storable in the catalog, not in the application programs.
11. ***Distribution Independence***: Users should not have to know that a relational database is distributed.
12. ***Non Subversion Rule***: If a relational system has a low-level (single record at a time) language, then this language cannot be used to subvert or bypass the integrity rules and constraints.

Database APIs

- SQL is a very powerful and expressive language for manipulating databases
- But SQL is not a full-fledged programming language for writing database applications.
- For example, SQL does not provide facilities for
  - statements to accept input from users,
  - processing the data before sending it to the database,
  - processing the data after retrieving it from the database
  - writing GUIs,
  - generating reports, etc.
- Such facilities are necessary for writing sophisticated and user-friendly database applications.
- SQL is not "computationally complete"
  - it is not possible to do all possible computations with it as is possible in a programming language such as Java or C++.
- To make up for missing capabilities, SQL is typically used from within a "host" programming language for writing applications.
  - Host programming language facilities are used where SQL is inadequate.
  - SQL APIs are used to communicate with a database.

Database APIs (contd.)

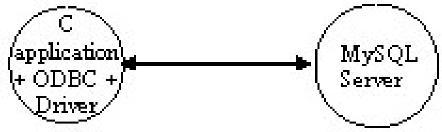
- SQL APIs have been defined for many programming languages
  - For example: C, C++, Java, and PHP.
  - All the SQL facilities available when directly interacting with the database server are also available using these APIs.
- SQL operations are set (table or relation) operations.
  - However, when using SQL APIs, processing of the results returned to the host programming language application is typically one row at a time.
  - A variable, called the cursor, is available to traverse the table returned as the result, either backwards or forwards.
- The scenario for writing an application, such as a GUI, is to use the API to
  - connect to the database,
  - send the query to the database, and
  - act upon the result returned.
- Using the API involves sending SQL to the database
  - SQL typically cannot be changed in an ad hoc manner as is the case when interacting directly with the database server.

Database APIs (contd.)

- Major relational databases support of variety of APIs such as those for C, C++, and Java programming languages, the programming languages in the .NET environment, and scripting languages such as PHP.
- When using these APIs, an application program must be compiled with a driver that acts as an interface and manages the interaction between the application and the database server. Different drivers serve as interfaces with different databases thus ensuring that the same application can work with different SQL database systems.
- To use these APIs, the API library, the host language, and the database driver must be installed on a computer, and the computer connected to the database server via some sort of network connection.
- MySQL supports several APIs for accessing MySQL databases from a variety of environments. Information about these APIs and how to download the necessary modules can be found at the MySQL website
- www.mysql.com
- I will now discuss APIs called ODBC used by C and C++ and JDBC used by Java to communicate with SQL databases, the former briefly and the latter in some detail. I will also briefly discuss how PHP scripts communicate with MySQL.

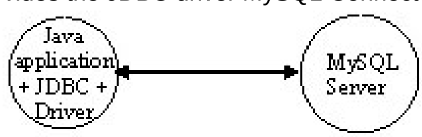
ODBC (Open Database Connectivity)

- Industry standard for interacting with SQL database systems from a variety of languages including C and C++ and their variants.
- Both database server and the host language must be ODBC compliant
- An appropriate ODBC driver needs to exist to allow the programming language to communicate with the database server.
- ODBC provides a
  - library that allows applications to connect to, and access and manipulate relational databases by executing SQL statements.
  - standard representation for the data types.
- MySQL supports ODBC.



JDBC (Java Database Connectivity)

- JDBC is the Java API for relational databases.
- JDBC is based on ODBC
  - Set of classes for interacting with a relational database.
- JDBC provides facilities for
  - connecting to a database server,
  - converting MySQL types to Java types,
  - sending SQL statements to a database server for execution,
  - processing the query results, and
  - closing the database connection.
- Database systems provide JDBC drivers to enable Java programs using JDBC to communicate with them.
  - MySQL provides the JDBC driver MySQL Connector/J



JDBC (contd.)

- A Java program can send SQL statements to the server for execution using the following classes :
  - Class **`Statement`**: Used to send SQL statements, which are specified as strings.
  - Class **`PreparedStatement`**: Used to send SQL statements that have been “precompiled” for fast execution.
  - Class **`CallableStatement`**: Used for sending stored procedures.
- The result of an SQL query is an object of type **`ResultSet`**
  - The result table is accessed one row at a time, i.e., the result table is “scrollable.”
  - A cursor, in the **`ResultSet`** object, points to the row currently being accessed.
  - The cursor initially points to the first row.
  - The rows can be accessed in any order by moving the cursor forward or backwards.
  - The fields can also be accessed in any order.
- Class **`ResultSetMetaData`** can be used to get information, such as column types, about the result table.

JDBC Example

- The example involves writing a Java program that communicates with the MySQL database server using JDBC to list the ISBNs of the books in the database and their prices.
- Here is the **`Books`** table (reproduced for your convenience):

ISBN	Title	Price	Authors	Pages	PubDate	Qty
0929306279	Bell Labs	29.95	Gehani	269	2003	121
0929306260	Java	49.95	Sahni & Kumar	465	2003	35
0670031844	White Moghuls	34.95	Dalrymple	459	2003	78
0439357624	Born Confused	16.95	Hidier	432	2002	11

JDBC Example (contd.)

```
import java.sql.*;
class ListIsbnPrice {
  public static void main(String[] args) {
    System.out.println("Hello from MySQL!\n");
    try { Class.forName("com.mysql.jdbc.Driver");
    } catch(java.lang.ClassNotFoundException e) {
       System.err.println("ClassNotFoundException: ");
       System.err.println("Message: " + e.getMessage()); }
    try { Connection con; Statement stmt;
      // JDBC URL Syntax: jdbc:TYPE://machine:port/DB_NAM; default port=3306
      //establishing connection; supply URL, user, password
      con = DriverManager.getConnection("jdbc:mysql://localhost/everest","root","mgr");
      stmt = con.createStatement();
      String query = "SELECT ISBN, Price FROM Books;"; ResultSet rs =
    stmt.executeQuery(query);
      System.out.println("  ISBN       PRICE");
      while (rs.next()) {
        String s = rs.getString("ISBN"); float n = rs.getFloat("PRICE");
        System.out.println(s + "   " + n); }
    stmt.close(); con.close();
} catch(SQLException e) {
      System.err.println(" SQL Exceptions \n");
      while (e != null) {
        System.out.println("Error: "+ e.getMessage());
        System.out.println("SQL State: " + e.getSQLState());
        System.out.println("Error Code:" + e.getErrorCode());
        e = e.getNextException(); System.out.println(""); } }
  }
}
```

PHP

- PHP is a server-side scripting language that facilitates the generation of dynamic web pages.
  - PHP is embedded in HTML pages.
  - The web server translates PHP scripts into HTML before sending a page to a web browser.
  - PHP is implemented as a preprocessor.
  - After a HTML page embedded with PHP is preprocessed, the result should be a valid HTML page.
  - A browser does not see any PHP.
- PHP scripts are inserted into HTML pages must be bracketed by

  `<?` and **`?>`**

- PHP offers facilities for connecting to relational databases.
- PHP and MySQL are an excellent combination for building data-driven websites.
- PHP provides built-in support for MySQL in the form of functions, e.g.,
  - mysql_connect(): Used to connect to a MySQL server.
  - mysql_select_db(): Used to specify the database to be accessed.
  - mysql_query(): Used to send a SQL statement to a MySQL server for execution.
  - mysql_result(): Used to display the result.

Importing & Exporting Data

- Importing data into a table from another database or from a file may be needed when creating a new database or when merging data from an external source.
- Similarly exporting data, e.g., to a file, may be needed for a variety of reasons, for example, to create a text file for use by another application.
- Inserting data using **`INSERT`** statements can be very inefficient because of
  - constraint checking,
  - index maintenance,
  - locking, etc.
- Consequently, databases provide special facilities for loading data in bulk that address the above issues.

Importing Data

- MySQL provides a bulk load facility, the **`LOAD DATA INFILE`** statement, for extremely fast import of data stored in text files.
  - During bulk loading, many checks such as the foreign key constraint can be turned off.
- Simple version of the **`LOAD DATA INFILE`** statement has the form

LOAD DATA INFILE '*file name*' INTO TABLE *table*;

- If the file name is a relative file name, then the server looks for the file relative to the folder (directory) where it stores the database.
- By default, MySQL expects to find the file on the server. If the file is on the client machine then keyword **`LOCAL`** must be specified:

LOAD DATA INFILE LOCAL '*file name*' INTO TABLE *table*;

- By default, MySQL expects fields to be separated by tab characters. And it expects data for each row to be on a separate line.
  - The **`LOAD DATA INFILE`** statement provides options to specify how the fields are separated and specified, and how the rows start and terminate.

Exporting Data

- To export a table in MySQL, the following form of the **`SELECT`** statement is used:

SELECT * INTO OUTFILE '*file name*' FROM *table*;

- By default, the fields will be separated by tabs and each row will be written on a new line.
- The file is created on the server.
- If an absolute file name is not specified, then the file will be created relative to the folder where MySQL creates the database.
- To specify alternative field termination and specification characters, the **`FIELDS`** clause is used:

SELECT * INTO OUTFILE '*file name*'
FIELDS TERMINATED BY '*character*'
ENCLOSED BY '*character*'
FROM *table*;

Importing/Exporting Data Example

- Write contents of the **Books** table to a file and then load this data into **Books2** whose definition is identical to that of **Books**. The following **SELECT** statement writes contents of **Books** to file **books.txt**:

```
SELECT *
INTO OUTFILE 'C:/temp/books.txt'
FIELDS TERMINATED BY ':'
FROM Books;
```

  Note use of **/** instead of **** even in the Windows environment
- File **temp.txt** now contains the following:

```
0439357624:Born Confused:16.95:Hidier:432:2002:11
0670031844:White Moghuls:34.95:Dalrymple:459:2003:78
0929306260:Java:49.95:Sahni & Kumar:465:2003:35
0929306279:Bell Labs:29.95:Gehani:269:2003:121
```

- The above text file can be loaded into table **Books2** as follows:

```
LOAD DATA INFILE 'C:/temp/books.txt'
INTO TABLE Books2
FIELDS TERMINATED BY ':';
```