Introduction to Database Systems

Narain Gehani

CS 431

- Welcome
- Who should take this course
- Syllabus
- Book – copies / volunteer / $
- TA
- Your background
- If I go faster, I will have a review class if you like

Introduction

- What is a database?
  - Any repository of data (paper file cabinets, Word files, Excel spread sheets, database systems)
  - Facilitates for storage, manipulation, retrieval, persistence, concurrency, fast access
- Users of databases
  - End users
  - **Database users**
  - **Database designers**
  - **Database administrators (DBAs)**
  - **Application programmers**
  - Database system implementers

Introduction (Contd.)

- Will teach you how database systems work, **not** how database systems are implemented.
- Why the above?
- Focus on concepts & fundamentals on how to use (relational) databases effectively.
- SQL – great for querying/manipulating relational databases but not a full-fledged language
- Host language such as Java/C++ make up for deficiencies – control structures, formatting, networking, etc

MySQL

- We will use MySQL
- “Open source” relational database
- Over 5 million installed systems

Relational Database Model

- Relational database
  - Data stored as tables
  - Data is extracted by throwing away unwanted rows and columns.
  - Tables can be joined in the process of extracting data
- Relational databases have won the war between competing database models
  - Conceptual simplicity
  - Separation of logical organization from physical details
  - Simple declarative language
  - Sound theoretical underpinnings (Codd)

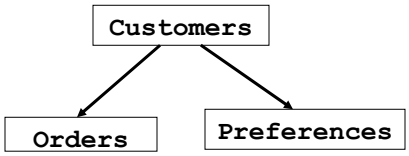
Other Database Models

- Hierarchical
- Network
- Object-oriented (Ode Object Database)
- XML

Hierarchical Databases

- Data is stored hierarchically with parent-child relationships between data items at adjacent levels – querying based on this.
- Consists of sets of records called *record types* organized as nodes of a tree.
- Record types and records correspond to the tables and rows in relational databases.

Hierarchical Databases (contd.)



- **Customers** is a record type – one record per customer.
- Each customer record is associated with records of types – **Orders** and **Preferences**.
- Each customer can have multiple orders & preferences.

Network Databases

- Consists of a bunch of sets
- Set elements are called records.
- Sets and records correspond to the tables and rows in relational databases.
- A record can belong to more than one set and this defines a relationship.
- The network formed by these relationships is what gives this database model its name.
- The network model eliminates the hierarchical limitation of hierarchal databases.
- No restriction on the number & type of relationships makes the database complex hard to understand.

Object Databases

- Started appearing circa 1987 with the growing popularity of C++.
- Allow users to structure, retrieve, and update data in terms of objects in the application domain.
- No “impedance” mismatch between the database & the application,
  - no need to convert data from the application data model to the database model & vice versa.
- The object database model, particularly the C++ model, does not have the simplicity or the sound theoretical underpinning of the relational database model.
- Object databases also have other disadvantages. E.g., there is no formal definition of the semantics of the C++ or its object model.
- Nevertheless, object databases have had a significant impact & object capabilities are being incorporated into relational databases.

XML Databases

- The Extensible Markup Language (XML) is a text markup language designed, circa 1996, for specifying the syntax of data and electronic documents [XML] such as Web pages. XML is particularly useful for describing semi-structured data. However, XML has proved to be so versatile that it is now being used extensively to describe the syntax and “semantics” of data in a wide variety of domains such as ecommerce, protocols that exchange data, etc.

XML Databases (contd.)

Example invoice stored in a XML database

Invoice Number: 5785796
Date: 1/4/2004

PC Inc.
1100 South Street
Morristown, NJ 07960

Susan Witzel
32 Broadway
Hoboken, NJ 07030

Item Number	Quantity	Description	Price
1	1	PC Pentium 4, 2 GHz, Mem 1GB	$1099.00
2	1	Printer HP 970Cxi	$399.00
		Total	$1498.00
		Paid	$0.0
		BALANCE DUE	$1498.00

XML Datbases (contd.)

```
<Invoice>
    <Number> 5785796 </Number>
    <Date> 1/4/2004 </Date>
    <Vendor>
        <CompanyName> PC Inc. </CompanyName>
            <Address>
                <Number> 1100 </Number>
                <Street> South Street </Street>
                <City> Morristown </City>
                <State> NJ </State>
                <Zip> 07960 </Zip>
            </Address>
    </Vendor>

    <Customer>
        <First> Susan </First>
        <Last> Witzel </Last>
        <Address>
            <Number> 32 </Number>
            <Street> Broadway </Street>
            <City> Hoboken </City>
            <State> NJ </State>
            <Zip> 07030 </Zip>
        </Address>
    </Customer>
```

XML Datbases (contd.)

```
   <Item>
        <Number> 1 </Number>
        <Quantity> 1 </Quantity>
        <PC>
            <Processor> Pentium 4 </Processor>
            <Speed> 2 GHz </Speed>
            <Memory> 1GB </Memory>
            <Price> $1099.00 </Price>
        </PC>
    </Item>
    <Item>
        <Number> 2 </Number>
        <Quantity> 1 </Quantity>
        <Printer>
            <Make> HP </Make>
            <Model> 970Cxi </Model>
            <Price> $399.00 </Price>
        </Printer>
    </Item>
    <Summary>
        <TotalCost> $1498.00 </TotalCost>
        <Paid> 0.00 </Paid>
        <BalanceDue> $1498.00 </BalanceDue>
    </Summary>
</Invoice>
```


XML Datbases (contd.)

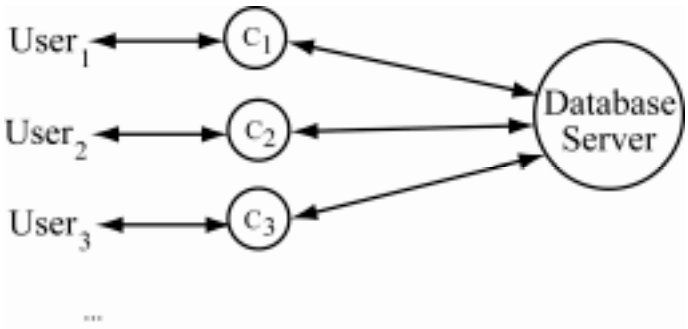
- XML describes the data part of the invoice but not the formatting, which is done with "style sheets" that are also written in XML
- XML databases are natural for storing & retrieving XML documents:
  - invoices, product information, medical records, B2B transaction logs.
- XML documents can contain both data and metadata
  - Relational databases are designed for storing data but not metadata.
  - XML documents can be stored in a relational database but the database will not be able to differentiate between data and metadata. Moreover, SQL will not understand XML.
- Storing XML documents in a relational database requires back & forth conversion – significant overhead.
- XML databases
  - will allow queries using XML concepts. In case of the invoice example, users will be able to write queries using components such as customer name and items ordered.
  - Locking, indexing, storage organization, etc. will be in terms of XML concepts leading to faster queries as compared to queries relating to XML documents stored in relational databases.
- XML databases are currently far from approaching the success of the relational databases in terms of simplicity, efficiency, and, most importantly, acceptance.

Interacting with a Database
Application or Single-User Mode

- The database runs on a workstation or a PC as an application that is invoked every time a user wants to use the database:

Interacting with a Database
Client-Server Mode

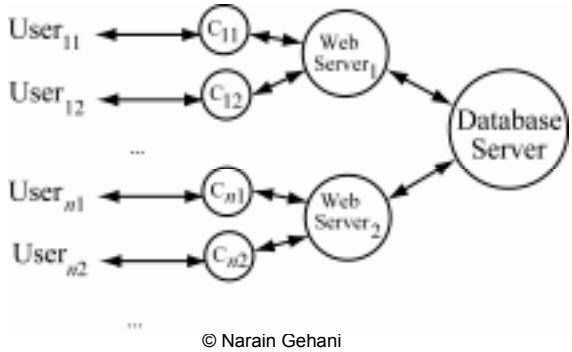
- When running in the "client-server" mode, typically multiple users can simultaneously interact with the database. The server runs continuously waiting for requests from clients (Ci) who come and go:



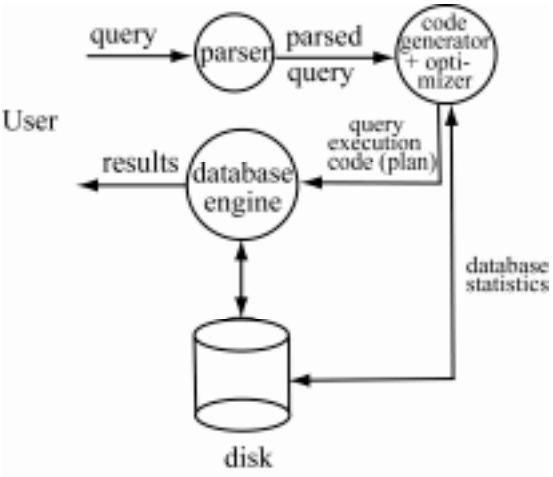
Interacting with a Database

Client-Server Mode (as a backend)

- The database server can also operate behind an application. **Example scenario**: users using a browser client (C*ij*) to interact with a web server which interacts with a database server:



Under the Hood (Mechanics)



Disk vs. Main Memory

- Typically, databases store data on disks
  - bring data to memory only when needed
  - write the data back to disk if it is changed.
- Since memory is much faster than disk, why not use memory?
  - Disk storage is persistent unlike main memory which is volatile
  - Disk storage is cheaper than main memory
- Items retrieved by the database from disk are stored in area of memory, called the *data buffer.*
  - Size of data buffer is typically much smaller than size of database because only a portion of database is accessed to answer a query.
  - If data needed to answer a query is larger than the buffer size, then either the buffer size is increased or the data is brought to the buffer in batches, each batch being processed & replaced by the next batch.

Disk vs. Main Memory (contd.)

- Buffer Operation
  - *Read Query*: If items needed for the query are not in the buffer they are brought from disk and put in the buffer.
  - *Update Query*: If the items to be updated are not in the buffer, they are brought from disk and placed in the buffer. They are updated and then written to disk (to make them persistent).
  - *Insert Query*: Items to be inserted in the database are first inserted in the buffer and then copied to disk.
  - *Delete Query*: Items are deleted from the disk and also from the buffer, if present in the buffer.
- After a data item in the buffer has been read or written to disk, it is not automatically discarded. Only when the buffer gets full, items are deleted to make space for new items.
- The bigger the buffer, the higher the probability that the data needed by a query will be in the buffer. Consequently, the larger the data buffer, faster the queries.
  - Classic tradeoff of speed vs. memory.

Disk vs. Main Memory (contd.)

Why Not Keep the Database in Memory?

- Since memory is getting cheap, one option is to make the data buffer as large as the database. However, database algorithms, in disk-based databases, have been designed for storing data items on disk. They will not make optimal use of such a buffer.
- To get maximal performance, database algorithms must be specially designed for databases that fits into memory. Such databases will be kept in main memory from the beginning with a copy on disk for persistence.
- An example of a commercial main-memory database is TimesTen (www.timesten.com).

Everest Books Database

- Everest Books is a book seller that
  - buys books from publishers and distributors and then
  - sells books to customers.
- To support the information needs of its business, Everest Books uses a database for
  - tracking the books bought and sold,
  - tracking payments,
  - generating invoices, and
  - generating a variety of analysis reports on demand.

Everest Books Database (contd.)

Inovice

EVEREST BOOKS
2300 Great Scenic View
Summit Top, VT 08211

Ship Date 2/4/04 Invoice # 004 Customer Id # 003
Order Date 2/4/04

Liza Singh
FastTrack
155 Route 133
Holmdel, FL 48901

ISBN	Title	Qty	Unit Price	Total
0929306279	Bell Labs	1	29.95	29.95
0929306260	Java	1	49.95	49.95
0439357624	Born Confused	1	16.85	16.85
0670031844	White Moghuls	1	34.85	34.85

Subtotal	131.80
Sales Tax	0.00
Shipping	6.99
TOTAL DUE	138.79

Everest Books Database (contd.)

Sales Report

EVEREST BOOKS

SALES REPORT

7/1/04 to 12/31/04

ISBN	Title	Qty	Book Sales
0929306279	Bell Labs	189	5660.95
0929306260	Java	145	7242.75
0439357624	Born Confused	89	1508.55
0670031844	White Moghuls	78	1221.50
		TOTAL	$15633.75

Everest Books Database (contd.)
Database Design

1. Determine queries that are needed
2. Determine data that needs to be stored
3. Requirements reality check
4. Design

- Iterative process between requirements and design

Everest Books Database (contd.)
Queries

- Invoicing
  - Invoice Generation
  - Lookup old invoices
- Explicit Database Updates needed when
  - more copies of existing books arrive,
  - new books (not in the database) arrive,
  - book prices change,
  - making corrections,
  - recording payments, etc.
- Lookups: Users should be able to
  - access book information,
  - look up information about their orders, etc.
- End Of Period Reports
  - sales per book,
  - total sales,
  - total sales tax collected,
  - total shipping charges,
  - cash received per book,
  - total cash received, etc.

Everest Books Database (contd.)

Data to be Stored

- Book Data
- Customer Data
- Order Data

Everest Books Database (contd.)

Specification Reality Check

- Besides the fact that nothing has been said about the user interface, much information has been left unspecified in the requirements in the book. For example:
  - "etc." has been used several times when specifying the data that needs to be stored.
  - The data format of the items is unspecified. For example, what exactly is an ISBN, …?
  - The report contents and formats are unspecified
  - The number of users accessing the database simultaneously is unspecified.
  - The number of orders and queries expected is unspecified, etc.
- The design of the database will be affected by the specifics of the above requirements. We will have to manage with an informal specification.
- Fortunately, the informal and incomplete nature of the above requirements specification also has a positive aspect – much freedom in producing a final database design.

Everest Books Database (contd.)

Functionality That Will Not Implemented To Keep the Database Simple

- Database will not track some activities, e.g., it will not
  - record the price Everest Books pays to buy books from publishers and distributors
  - handle disbursements
  - receipts for invoice payments.
- Some information will not be recorded to reduce the number of columns in the tables so that the tables can be displayed on a book size page. E.g.,
  - customer contact information
- No provision for discounts, different types of shipping, no shipping rates table, etc.
- No restrictions on who can look at what data.
- Order shipping information will not be recorded. Changes to orders should be entertained only if the order has not been shipped.
- The database will not be integrated with ecommerce facilities such as a shopping cart and credit card authorization.

Relational Databases

- A relational database, consists of relations (tables), which are manipulated by
  - cutting, selecting, joining, along with union, insertion, difference, and deletion operations to extract, add, and delete data.
- The “schema” of is a logical description (meta data) of the tables in the database, restrictions on the data, if any, alert actions (triggers), and data structures (indexes) specified for fast access.
- Relational databases are closely associated with the SQL query language which is used for manipulating relational databases.

MySQL

- MySQL supports entry-level SQL-92 and is aiming to support the full SQL-2003. MySQL also supports some non-standard SQL.
- Most databases support transactions which have many desirable properties – grouping of multiple operations into one atomic action, multiple users can manipulate the database simultaneously without interfering with each other, etc.
  - MySQL databases can have both transaction-safe & non-transaction-safe tables.
  - Using **transaction-safe** tables means automatic recovery in case of failures, grouping of multiple actions into one atomic action, concurrent users, etc.
  - MySQL treats each operation on **non-transaction-safe** tables as atomic but multiple operations cannot be grouped & treated as a single atomic operation.
    - Fine for single-user databases but multiple simultaneous users can lead to an inconsistent database.
    - A front-end application, such as a web server, can ensure that multiple users are serialized, that is, the one user at a time is allowed to manipulate the database.
- Databases that do not support transactions are typically much faster, use less disk space and less memory.

MySQL

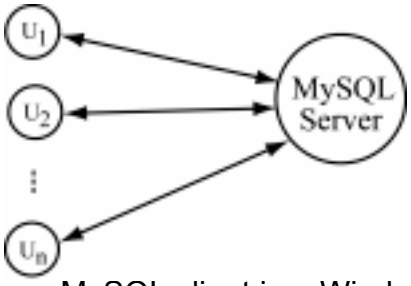
Some Storage Engines

MySQL databases can to use different storage engines, each with different characteristics.

Engine	Support	Comment
MyISAM	Default	Default engine – great performance
HEAP	Yes	Alias for MEMORY
MEMORY	Yes	Stored in memory, useful for temporary tables
InnoDB	Yes	Transactions, row-level locking, and foreign keys
BDB	No	Transactions & page-level locking
NDBCLUSTER	No	Clustered, fault-tolerant, memory-based tables

MySQL
Client-Server

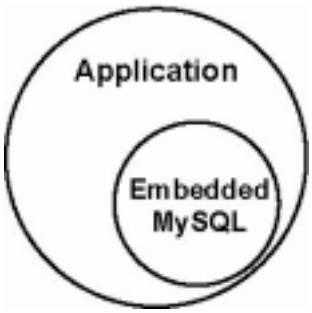
- In the "client-server" mode, MySQL runs as a server application
  - arbitrary number of users (U_i) called clients:



- Each user runs a MySQL client in a Windows command prompt window
- Clients manipulate a MySQL database by sending SQL commands to the server, which executes the requests, & sends the command status and results back to the client.

MySQL
Embedded Server

- For standalone applications with their own embedded database MySQL provides a library that allows a MySQL database server to be embedded in the application.
  - ideal for applications where a database is needed "behind the scenes" but where users do not need to directly interact with the database.



Firing up MySQL

- **Starting the Database Server**
  - The MySQL database server is started in a command prompt window by typing
    ```
    C> mysqld
    ```
    Assumption: Windows `PATH` environment variable has been modified to include " MySQL".
- **Starting the Database Client**
  - The client is started in a Windows command prompt window by typing
    ```
    C> mysql
    ```
  - Execution of the above command generates the following prompt
    ```
    mysql>
    ```
    provided the MySQL server `mysqld` has been started.
  - A user can now enter SQL queries which are sent by the client to the MySQL server.
- **Stopping the Client and Server**
  - A MySQL client can be terminated by typing quit or `\q`, e.g.,
    ```
    mysql> quit
    ```
  - The MySQL database server can be terminated with the command
    ```
    C> mysqladmin shutdown
    ```
    in a different command prompt window.

Creating the Everest Books (empty) Database

- Create a new and empty database for Everest Books and make it the default database:
  ```
  mysql> CREATE DATABASE Everest;
  mysql> USE Everest;
  ```
- To see all the databases being managed by the MySQL database server:
  ```
  mysql> SHOW DATABASES;
  ```
- The following command shows all the tables in the default database:
  ```
  mysql> SHOW TABLES;
  ```
- Incidentally, as you can see, a semicolon must also terminate MySQL commands (and SQL queries), when entered at the MySQL client.

Using the Everest Books Database

- Assuming that database has been created & populated:

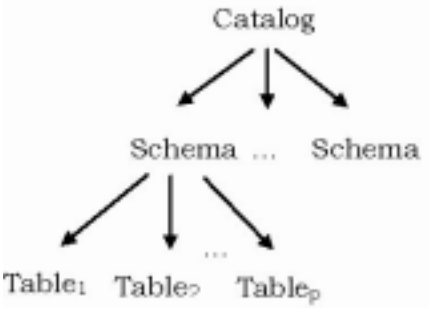
```
mysql> SELECT Title
    -> FROM Books;
```

- `Title` is a column of table `Books` (coming up).
- The MySQL server returns
  - the result (as a table), the number of rows, execution time.
- as follows:

```
+---------------+
| Title         |
+---------------+
| Born Confused |
| White Moghuls |
| Java          |
| Bell Labs     |
+---------------+
4 rows in set (0.02 sec)
mysql>
```


Relational Databases
Organization

- Relational databases are “flat”. But, there is hierarchy in the database organization:



In MySQL, the command

```
mysql> show databases;
```

displays database names, in effect, the catalog.

Tables
Customer Table

Id	Company	First	Last	Street	City	State2	Zip	Tel
1	Acme	Tom	Jones	25 Scenic Road	Hilltop	NJ	18901	9189088919
2		Susan	Wise	32 New Road	Union	AZ	78901	8588188119
3	FastTrack	Liza	Singh	155 Route 133	Holmdel	FL	48901	2185552223
4	Clover	Alan	Feuer	113 Waltham Ave	Freehold	MA	08901	6192152152
5	TechSmart	Vinod	Oza	233 11th St	Chatham	NJ	18901	2980989889

Creating & Using the Database
Creating Tables (the Customer Table)

```
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;
```

Creating & Using the Database
Populating Tables (the Customer Table)

```
INSERT INTO Customers
VALUES(1, 'Acme', 'Tom','Jones',
       '25 Scenic Road', 'Hilltop',
        'NJ', '18901', '9189088919');
```


Creating & Using the Database
Queries

- Lists all the rows of the `Customers` table:

```
SELECT *
FROM Customers;
```

- The above query unconditionally selects all the rows in the `Customers` table.
- If the table is large, we may not want to print the whole table. E.g., to list only the rows for customers in NJ:

```
SELECT *
FROM Customers
WHERE State2 = 'NJ';
```

Id	Company	First	Last	Street	City	State2	Zip	Tel
1	Acme	Tom	Jones	25 Scenic Road	Hilltop	NJ	18901	9189088919
5	TechSmart	Vinod	Oza	233 11th St	Chatham	NJ	18901	2980989889

Creating & Using the Database

Queries (contd.)

- Irrelevant columns can be left out by explicitly listing only columns of interest:

```
SELECT Company, First, Last, Tel
FROM Customers
WHERE State2 = 'NJ';
```

Company	First	Last	Tel
Acme	Tom	Jones	9189088919
TechSmart	Vinod	Oza	2980989889

Steps in Designing the Database

1. Problem or requirements definition.
2. Determining the data that needs to be stored in the database.
3. Deciding what tables will contain what data.
   - The tables should reflect the problem structure.
   - Data should be stored only once as far as possible because data redundancy will require multiple updates and can lead to data inconsistency.
4. Deciding upon data properties based on the requirements.
5. Deciding what queries have to be implemented.

Everest Books Tables

Descriptions

Table	Stores
`Books`	Information about each book in the database. Each book will have a unique id, called the ISBN, which are standard id's used in the book industry to uniquely identify books.
`Customers`	Contact information about each customer. Each customer will be assigned a unique customer id to facilitate customer lookups.
`Orders`	Contains information about each order. Each order is identified by the order id `OrderId`.
`OrderInfo`	Information about the books contained in each order. The order associated with the books is identified by the order id.

Everest Books Tables

Comments On Orders & OrderInfo Tables

- **Problem**: In the **`Orders`** table, it would be nice to record each order as a single row.
  - Unfortunately, orders can have variable number of books → variable number of columns.
  - Tables cannot have a variable number of columns.
- **Possible Solution**
  - Define the table to have a large number of columns, allowing most orders to fit in one row.
  - Larger orders will be have to be placed and filled as multiple orders.
  - In many cases, space will be wasted.
- The above solution is not pleasing.
  - Multiple orders for one logical order is not good because it will require reentering shipping and other information for each of the multiple orders.
  - A large book limit is wasteful of storage.

Everest Books Tables

Comments On Orders & OrderInfo Tables (contd.)

Solution Used Typically

- Shunt the variable items to another table whose rows are used for the variable number of items – one per row. An id is used to identify the multiple rows in the new table with a single row in the original table.
  - We will define a new table **`OrderInfo`** that will have a row for each book in an order, and each such row will have an id that associates it with the order in the **`Orders`**.

Manipulating the Database

- The relational database model defines an algebra for manipulating tables as mathematical objects.
  - Not a practical language for manipulating a real world database.
  - For example, the relational algebra does not provide facilities for creating tables and updating tables.
- The SQL database language, based on the relational algebra, is practical for manipulating databases.
- Before we take a close look at SQL, we will look at the relational algebra
  - shows the beauty of relational databases
  - helps understand important relational database concepts.

Example Tables

- Sample data coming up
- Small tables both in number of rows and columns
  - presentation purpose
  - can be easily extended

Everest Book Tables

Books

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

Everest Book Tables
Customers

Id	Company	First	Last	Street	City	State2	Zip	Tel
1	Acme	Tom	Jones	25 Scenic Road	Hilltop	NJ	18901	9189088919
2		Susan	Wise	32 New Road	Union	AZ	78901	8588188119
3	FastTrack	Liza	Singh	155 Route 133	Holmdel	FL	48901	2185552223
4	Clover	Alan	Feuer	113 Waltham Ave	Freehold	MA	08901	6192152152
5	TechSmart	Vinod	Oza	233 11th St	Chatham	NJ	18901	2980989889

Everest Book Tables
Orders

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

Everest Book Tables

OrderInfo

.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

Some Table Characteristics

- Tables (relations, to be precise) can be treated as mathematical objects.
  - Tables are sets of rows.
  - Values in each column are of the same basic (atomic) type.
  - No duplicate rows in the relational algebra. SQL query results can have duplicate rows – these can optionally be eliminated.
- Each row in a table can be uniquely identified using a subset of column values called the ***key***
  - This subset can be the whole row since each row in a table is unique since duplicates are not allowed.
  - A multi-column key is called a ***composite*** key.
  - A table can have many keys, but only one can be selected as the ***primary*** key. A primary key value cannot be a null value or, in case of a composite key, cannot contain null values.
  - A key whose role is not going to change over time, and is likely to remain a key over time is a good candidate for being selected as the primary key.
  - Database systems take advantage of key information, e.g., rows are ordered according to the primary key to support fast searches based on primary key.

Some Table Characteristics
Column Names

- Columns in different tables can have the same name. To avoid ambiguity, such column names can be prefixed with their table names:

  tableName.columnName

- Using the same column name in different tables does not imply that the columns or of the same type and/or that they have the same semantics.
  - Column **`Qty`** in **`Books`** represents the number of copies of a book in stock while in **`OrderInfo`** it specifies the number of copies ordered by a customer.
  - Column **`Price`** in **`Books`** represents the current price of book but in **`OrderInfo`** represents the price of the book when the order was placed.
- Different columns names can represent the same semantic value while identical column names can refer to semantically different items. Columns in different tables may have the same name for reasons such as
  - convenience, and
  - tables were defined by different persons.

Relational Algebra

- Mathematical foundation
- Closed under its operations
- Not a full fledged programming language
- SQL based on this (not a full fledged programming language)

Relational Algebra (contd.)

Operation	Symbol	Semantics
Projection	π	Select columns of a table.
Selection	σ	Select rows of a table.
Cross Product	$\times$	Join two tables by pasting all rows of the second table to each row of the first table.
Join	$\bowtie$	Join two tables by pasting related rows of the two tables together.
Union	$\cup$	Compute a new table from two tables such that each row in the new table belongs to at least one of the two tables.
Intersection	$\cap$	Compute a new table from two tables such that each row in the new table belongs to both tables.
Difference	$-$	Compute a new table from two tables such that each row in the new table belongs to the first table but not to the second table.

Relational Algebra
Projection

- The projection operator π is used to extract columns of interest from a table by cutting out the other columns

$$\pi \text{ Company, First, Last, Tel(Customers)}$$

yields

Company	First	Last	Tel
Acme	Tom	Jones	9189088919
	Susan	Wise	8588188119
FastTrack	Liza	Singh	2185552223
Clover	Alan	Feuer	6192152152
TechSmart	Vinod	Oza	2980989889

Relational Algebra
Projection (contd.)

- General form

π *column-names*(*table*)

- Unlike in SQL, duplicate rows are automatically eliminated

π State2(Customers)

for example,

State2
NJ
AZ
FL
MA

Relational Algebra
Projection (contd.)

Should duplicates be automatically eliminated?

Relational Algebra
Selection

- The selection operator σ is used to select or pick only those rows from a table that satisfy the specified criteria. E.g.:

$$\sigma_{State2=NJ(Customers)}$$

Id	Company	First	Last	Street	City	State2	Zip	Tel
1	Acme	Tom	Jones	25 Scenic Road	Hilltop	NJ	18901	9189088919
5	TechSmart	Vinod	Oza	233 11th St	Chatham	NJ	18901	2980989889

Relational Algebra
Selection (contd.)

- General Form

$$\sigma_{selection\text{-}condition(table)}$$

- Selection condition is a Boolean condition
- Closed under algebra – nested operations

$$\pi_{Company,First,Last,Tel(\sigma_{State2=NJ(Customers)})}$$

Company	First	Last	Tel
Acme	Tom	Jones	9189088919
TechSmart	Vinod	Oza	2980989889

Relational Algebra

Nested Operations: Selection & Projection

- Better to do selection first and then projection
  - Why?
- We can in many cases do the projection first and then the selection.
  - However, this will not work in the above example because if we first delete the State2 column, then it will not be possible to do the selection since there will be no state values left on which to base the selection.

Relational Algebra

Cross Product

- The cross product and join operators are used to paste together two tables. The cross product

  $R \times S$

  produces a new table in which each row of R is appended with each row of S. Thus if R has *n* rows and S has *m* rows, the result will have $n \times m$ rows. The number of columns in the result equals the number in R plus the number in S.
- The cross product, when used in isolation, is generally not meaningful. For example,

  $Books \times OrderInfo$

  will produce a table with 32 rows (based on our example tables) with most rows not being meaningful.
- Why?

Relational Algebra

Cross Product (contd.)

- The following cross product is meaningful because only matching ISBN rows appear pasted in the result:

$\sigma_{\text{OrderInfo.ISBN = Books.ISBN}}(\text{Books} \times \text{OrderInfo})$

OrderId	ISBN	Qty	Price	ISBN	Title	Price	Authors	Pages	PubDate	Qty
1	0929306279	1	29.95	0929306279	Bell Labs	29.95	Gehani	269	2003	121
1	0929306260	1	49.95	0929306260	Java	49.95	Sahni & Kumar	465	2003	35
2	0439357624	3	16.95	0439357624	Born Confused	16.95	Hidier	432	2002	11
3	0670031844	1	34.95	0670031844	White Moghuls	34.95	Dalrymple	459	2003	78
4	0929306279	1	29.95	0929306279	Bell Labs	29.95	Gehani	269	2003	121
4	0929306260	1	49.95	0929306260	Java	49.95	Sahni & Kumar	465	2003	35
4	0439357624	1	16.95	0439357624	Born Confused	16.95	Hidier	432	2002	11
4	0670031844	1	34.95	0670031844	White Moghuls	34.95	Dalrymple	459	2003	78

Relational Algebra

Join

- The combination of a selection operation with the cross product operation is defined as an operation in its own right, the join (or the inner join) operation. A *join* is basically a pasting of related rows in two tables, the relationship being defined by a Boolean condition. The join condition involves comparing values in the rows of the two tables and pasting together rows that satisfy the condition. The join operation is denoted by the $\triangleright\triangleleft$ operator and has the form
- R $\triangleright\triangleleft$ *join-condition* S
- The rows in the join table, the result of the join, are formed by pasting the rows of the tables R and S that satisfy the *join-condition*. Each row in the first relation, R, is compared to the each row of the second relation S All possible pair combinations of rows in R and S are considered to determine if the join condition is satisfied. Pairs of rows that satisfy the condition are pasted together and included in the join table. The total number of columns in the join table is sum of the number of columns in tables R and S.

Relational Algebra

Join (contd.)

- To illustrate the use of the join operator, suppose we want a table that lists the book titles along with the number of copies sold:

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

Relational Algebra

Join (contd.)

- The data is 2 tables:
  - The Info about books sold is in OrderInfo
  - But this is in ISBNs. Titles we have to extract from Books
  - We have to join these 2 tables together andn then extract the info
  - We also need to compute the sums – but we cannot do is in the relational algebra. We can do this in SQL.

Relational Algebra

Join (contd.)

OrderInfo $\triangleright\triangleleft$ OrderInfo.ISBN = Books.ISBN Books

From ORDERINFO				From BOOKS						
OrderId	ISBN	Qty	Price	ISBN	Title	Price	Authors	Pages	PubDate	Qty
1	0929306279	1	29.95	0929306279	Bell Labs	29.95	Gehani	269	2003	121
1	0929306260	1	49.95	0929306260	Java	49.95	Sahni & Kumar	465	2003	35
2	0439357624	3	16.95	0439357624	Born Confused	16.95	Hidier	432	2002	11
3	0670031844	1	34.95	0670031844	White Moghuls	34.95	Dalrymple	459	2003	78
4	0929306279	1	29.95	0929306279	Bell Labs	29.95	Gehani	269	2003	121
4	0929306260	1	49.95	0929306260	Java	49.95	Sahni & Kumar	465	2003	35
4	0439357624	1	16.95	0439357624	Born Confused	16.95	Hidier	432	2002	11
4	0670031844	1	34.95	0670031844	White Moghuls	34.95	Dalrymple	459	2003	78

Relational Algebra

Join (contd.)

- Many columns in the previous table are irrelevant. We will use projection:

π OrderId, Title, OrderInfo.Qty (OrderInfo $\triangleright\triangleleft$ OrderInfo.ISBN = Books.ISBN Books)

.OrderId	Title	Qty
1	Bell Labs	1
1	Java	1
2	Born Confused	3
3	White Moghuls	1
4	Bell Labs	1
4	Java	1
4	Born Confused	1
4	White Moghuls	1

Relational Algebra

EquiJoin + Natural Join

EquiJoin

- An *equijoin* is a join operation that joins rows (of two tables) with equal values for a common set of columns. Leads to one or more pairs of columns with identical values.

Natural Join

- A *natural join* is the same as equijoin with the following exception. The join is performed on columns with the same name in the two tables and only one of each identical pair of columns is retained. Notation example:

OrderInfo ▷◁ Books

Relational Algebra

Set Operations

- Union
- Intersection
- Difference
- etc.

Database Design

- Designing a database is critical to the correct and efficient functioning of a database.
  - In a relational database, this means defining the tables and the information that should be stored in them.
  - It may be also appropriate to define data structures, called indexes, to facilitate fast execution of queries.
- The design of the database must reflect user needs.
- A database designer must understand how the database will be used.
- The database designer must determine items such as
  - the data that needs to be stored,
  - the operations that will be performed on the stored data,
  - the frequency of the operations that will be performed,
  - the constraints that are to be imposed on the data, etc.
- Database modeling tools are often used to capture database requirements and the resulting database model then used to design the database.

ER Model

- A popular model used for modeling databases is the Entity-Relationship (ER) model
  - Data is described in terms of "entities", "entity sets," "attributes," and "relationships.".
- An *entity* is an object that can be distinguished from other objects.
  - Examples: a book is an entity, a customer is an entity.
- Similar entities form *entity sets*.
  - Examples: `Books` and `Customers` are entity sets if they represent a set of books and customers, respectively.
- Entities are characterized by properties called *attributes*.
  - the price of a book is an attribute of the entity book
  - An entity must have one or more attributes.
  - All entities in an entity set have the same attributes.
- Attributes have *values* associated with them.
  - Example: attribute price of a book may have the value $29.95.
  - Different attribute values distinguish similar entitles from each other.
  - Many entities can have the same attribute value.
- Values of a subset of these attributes are used to distinguish entities in an entity set from each other.
  - This set of attributes is called the *key* of the entity set.

ER Model -- Relationships

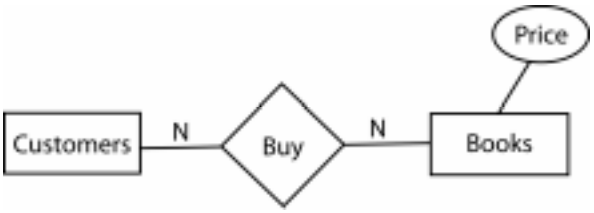
- Relationship
  - A *relationship instance* specifies an association between entities.
  - A *relationship set* specifies a relationship between two entity sets.
  - We will informally use the term relationship to refer to both of the above
  - A relationship *maps* or *relates* entities in one entity set to entities in another entity set.
- Types of relationships:
  - *One-to-one*
  - *One-to-many*
  - *Many-to-one*
  - *Many-to-many*
- Example: relationship **`Buy`** between entity sets **`Customers`** and **`Books`** is many-to-many.
- Relationships can also have attributes.
  - They are used to give information about the relationship.
  - For example, the **`PurchaseDate`** attribute in the **`Buy`** relationship can be used to describe when a book was purchased.
  - Each book identified by the relationship **`Buy`** will have its own **`PurchaseDate`** value.

ER Model
Graphical Representation

- The ER model allows the database design to be represented graphically:
  - An entity set is represented by a rectangle.
  - A relationship is represented by a diamond with lines connecting it to the two entity sets that it relates.
    - A “many” relationship is indicated by a “N” next to the line.
    - A “one” relationship is indicated by a “1” next to the line.
  - An attribute is represented by an oval connected by a solid line to the entity set rectangle or to the relationship diamond.

ER Model (Contd.)

- As an example, the following ER diagram models the fact that a customer can buy many books and many customers can buy the same book.
  - A deeper look into the operations of the Everest Books will lead us to modeling book buying as a 2-step process with each customer placing an order to buy books and Everest Books shipping books specified in the order.



Modeling Everest Books' Database

- We will now develop a reasonably complete ER model for the Everest Books database.
- By understanding data needs of Everest Books, one can conclude that

  Everest Books' business revolves around customers placing orders for books and Everest books shipping the books specified in the orders.

- This leads us to model the Everest Books database as consisting of
  - three entity sets, **`Customers`**, **`Orders`**, and **`Books`**, and
  - two relationships, **`PlaceOrder`** between **`Customers`** and **`Orders`**, and **`OrderInfo`** between **`Orders`** and **`Books`**.

Customers Entity Set

- Everest Books database needs to store the customer's company, name, and contact information. This leads to the following attributes:
  - **`Company`**,
  - **`Last`** (last name),
  - **`First`** (first name),
  - **`Street`**,
  - **`City`**,
  - **`State`**,
  - **`Zip`**, and
  - **`Tel`** (telephone number).
- Note that the name and address of a customer should be stored in component form to facilitate querying and report generation.
- The attributes listed above may not uniquely identify a customer
  - several customers can belong the same company,
  - several customers can have the same name,
  - a family can share an address.
- Consequently, we will add an attribute
  - **`Id`** (unique customer id for each customer)

Orders Entity Set

- Entity set **`Orders`** needs to have the following attributes:
  - **`OrderId`** (a unique id that identifies each order),
  - **`CustomerId`** (stores customer Id values),
  - **`OrderDate`**,
  - **`ShipDate`**,
  - **`Shipping`** (shipping charge), and
  - **`SalesTax`**.
- The above attributes do not address the books in each order.
  - An order can have a variable number of books.
  - Specifying a variable number of attributes is not possible
  - An one-to-many relationship between **`Orders`** and **`Books`** will allow us to specify the books in each order.
  - Information about the books ordered will be stored as attribute values of the **`OrderInfo`** relationship that will map each order in **`Orders`** to the books in the order, the books being members of the entity set **`Books`**.
  - There will be one set of attribute values for each book in the order.
- Order information in a database designed for commercial use will include other information such as purchase order number.

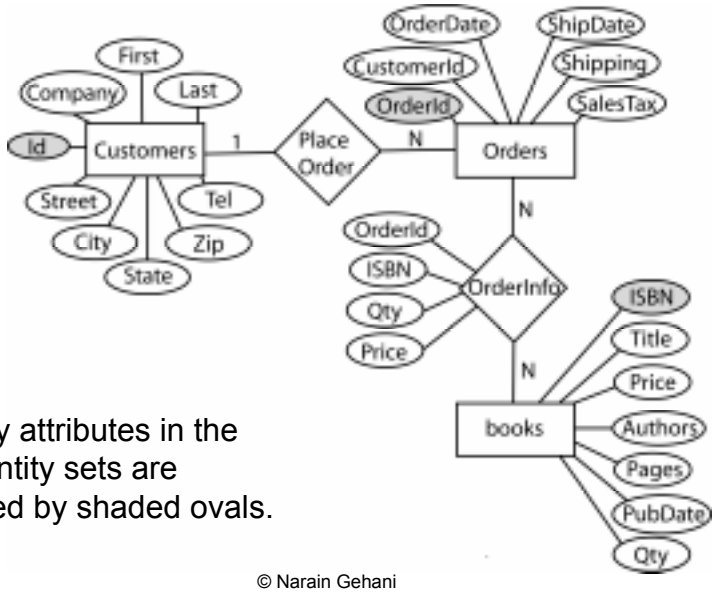
Books Entity Set

- Information to be stored about each book leads us to the attributes
  - **`ISBN`** (unique value),
  - **`Title`**,
  - **`Price`** (current price of the book),
  - **`Authors`** (comma separated list of last names),
  - **`PubDate`** (publishing date), and
  - **`Qty`** (quantity of each book in stock).
- Better to list each author individually, author's first & last names separate.
  - Will facilitate searches and report generation.
- Also, a book can have a variable number authors.
  - As mentioned earlier, specifying a variable number of attributes is not possible.
  - Recording information about a variable number of authors can be modeled as a one-to-many relationship between **`Books`** and a new **`AuthorNames`** entity set.
- To keep the example succinct, we will store author last names in a comma separated list.

Relationships

- **PlaceOrder**
  - The relationship **`PlaceOrder`** between the entity sets **`Customers`** and **`Orders`** is a one-to-many relationship
  - A customer can place multiple orders, but an order can be associated with only one customer.
  - No attributes needed
- **OrderInfo**
  - The relationship **`OrderInfo`** between entity sets **`Orders`** and **`Books`** is a many-to-many relationship. An order can have multiple books, and a book can be in multiple orders.
  - Needs attributes to record the information about the books in an order.

Everest Books' Database Model



The key attributes in the three entity sets are identified by shaded ovals.

ER Model → Relational DB

- The ER model shown above captures the requirements of the Everest Books database system.
- The next step is to create a relational database schema that reflects this ER model
  - specify and create the tables

Entity Sets → Tables
Attributes → Columns

- Each entity set in the ER model is represented by a table in a relational database.
  - Entity sets **`Customers`**, **`Orders`**, and **`Books`** are represented as tables
- Attributes become columns
  - Values of columns **`Id`**, **`OrderId`**, and **`ISBN`** uniquely identify members of the entity sets **`Customers`**, **`Orders`**, and **`Books`**, respectively
  - Consequently, they become the **`key`** columns

Relationships → Tables
Attributes → Columns

- Each relationship is also represented by a table.
- A relationship table includes the key columns of the two entity set tables it relates.
- Relationship attributes become columns of the relationship table.

Relationships → Tables (contd.)

- Relationship **`OrderInfo`** links the entity sets **`Orders`** and **`Books`**.
- Attributes **`OrderId`** and **`ISBN`** uniquely identify members of the entity sets **`Orders`** and **`Books`**, respectively.
  - These two attributes, along with the other attributes, become columns in the relationship table **`OrderInfo`**

.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

Relationships → Tables (contd.)

- **`PlaceOrder`** relationship table will have columns **`Id`** and **`OrderId`** to enable mapping of customers to orders.
  - Given a customer id, we can determine the ids of the orders placed by the customer.

PLACEORDER

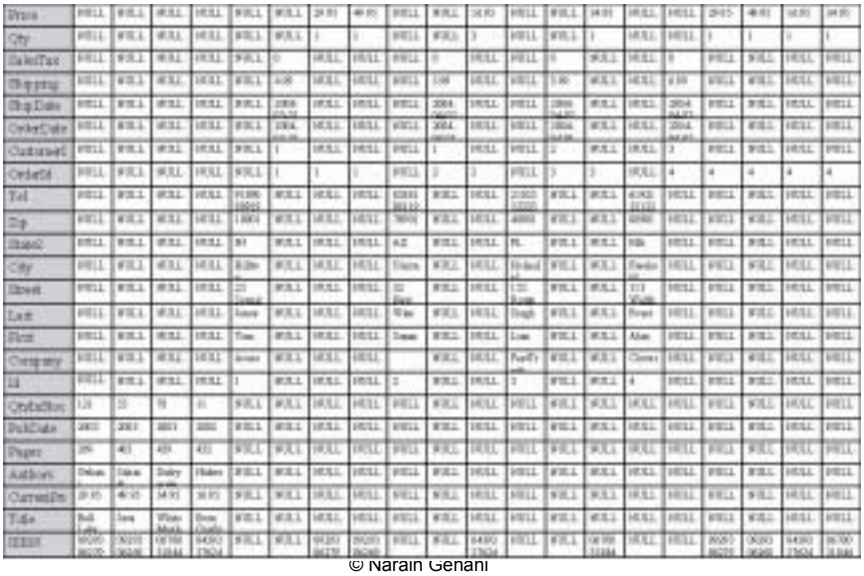
CustomerId	OrderId
1	1
1	2
2	3
3	4

- Since the customer id has been included in the entity set **`Orders`**, there is no need for the table **`PlaceOrder`**.
- It is possible to eliminate a relationship table by storing the relationship information in one of the tables of being related.
  - In case of one-to-many or one-to-one relationships, this can be done without duplicating data

Ad Hoc Database Design

- A database can be designed informally using ad hoc techniques but this can get challenging when the database is complex.
- As an example of *ad hoc* design, assume that we start off by storing all the data in one table.
  - Such a table is called the *universal* table (or relation):

Universal Table



Universal Table (contd.)

- Instead of 4 tables, we have 1 table
- Advantages
  - Simplicity of schema
  - No joins, which are expensive
- Disadvantages
  - Relationships between columns are hard to understand.
  - Lots of columns, some not needed for each row – wasted space
  - Large table, will require lot of disk accesses
  - Data redundancy
  - Less concurrency

Ad Hoc Database Design (contd.)

- Seeing many disadvantages of a universal table, how do we partition data into multiple tables?
- Seems reasonable to keep book data in a separate table, say Books.
- The rest of the data can be stored as follows:
  1. Customer and order information are stored in one table and order details in a separate table.
  2. Order information and order details are stored in one table, and customer information in a separate table.
  3. Customer information, order information, and order details are kept in separate tables (as shown earlier).
- Assume we start with option 2 – order information in **`OrderWithDetails.`**

CustomerId	OrderId	OrderDate	Ship Date	Shipping	Sales Tax	ISBN1	Qty1	Price1	...	ISBN4	Qty4	Price4
1	1	2004-03-31	2004-03-31	4.99	0	0929306279	1	29.95	...	NULL	NULL	NULL
1	2	2004-04-01	2004-04-02	5.99	0	0439357624	3	16.95	...	NULL	NULL	NULL
2	3	2004-04-01	2004-04-02	3.99	0	0670031844	1	34.95	...	NULL	NULL	NULL
3	4	2004-04-02	2004-04-02	6.99	0	0929306279	1	29.95	...	0670031844	1	34.95

Ad Hoc Database Design (contd.)

- **`OrdersWithDetails`** can handle only orders with at most 4 different books.
  - This arbitrary design limit is imposed because a table cannot have a variable number of columns.
  - Table **`OrdersWithDetails`** wastes storage when less than 4 books are ordered.
- A table cannot have a variable number of columns but can have a variable number of rows.
  - A separate table can be therefore be used to store information about a variable number of books – ISBNs, the number of copies, and prices.
  - To eliminate above problems, a database designer may want to split table **`OrdersWithDetails`** into multiple tables.
  - Note that in such a scenario, joins will be required to retrieve information about all the books in an order.
- Splitting **`OrdersWithDetails`** into two tables leads us back to our original tables **`Orders`** and **`OrderInfo`**.

Normal Forms

- Data normalization is the decomposition of tables into smaller tables to make databases efficient by
  - keeping logically related data in separate tables, and
  - eliminating redundancy.
- Normalized tables
  - facilitate the correct and consistent modification of data.
  - lead to update efficiency.
- Several normal forms of tables have been proposed.
  - They form a linear hierarchy with each normal form building upon the previous one.
  - We will just discuss the first three normal forms.

Normal Forms (contd.)

- A database with tables in third normal form (3NF) has desired properties:
  - ensuring that logically separate data is in separate tables
  - eliminating redundancy.
- Ensuring that the database design is good from the start is critical.
  - Hard to make changes later without serious consequences such as rewriting applications.
- Table normalization rules are guidelines for good database design.
  - At times it may be appropriate to deviate from these guidelines to improve database performance.
  - E.g., normalization typically requires splitting a table into multiple tables as a result of which queries may require joins.
  - Joins can be expensive.
  - If the majority of the queries require joins, then consider using a non-normalized database but be careful to avoid inconsistencies by ensuring, e.g., that all duplicate items are updated.

First Normal Form

Tables in the first normal form (1NF):

All columns must contain atomic data type values

Table **`Interpreter`** (below) is not in 1NF:

Id	Last Name	First Name	Languages
2136	LaPorta	Tom	English, French, Spanish, Hindi
2245	Munshi	Aparna	Bengali, French
4124	French	Susan	French, Italian, English
7832	Sequeira	Jim	Portuguese, Spanish, Italian

First Normal Form (contd.)

- The previous version of the **`Interpreter`** table is not in 1NF because column **`Languages`** consists of a repeating group.
- We can transform **`Interpreter`** to 1NF by keeping each language a separate column.

Id	LastName	FirstName	Language1	Language2	Language3	Language4
2136	LaPorta	Tom	English	French	Spanish	Hindi
2245	Munshi	Aparna	Bengali	French	NULL	NULL
4124	French	Susan	French	Italian	English	NULL
7832	Sequeira	Jim	Portuguese	Spanish	Italian	NULL

First Normal Form (contd.)

- Although **`Interpreter`** is now in 1NF, the number of languages that can be associated with an interpreter is four.
- Space is wasted, if an interpreter has less than four language skills.
- To avoid this problem, we can split the table into two tables – a revised **`Interpreter`** table

Id	LastName	FirstName
2136	LaPorta	Tom
2245	Munshi	Aparna
4124	French	Susan
7832	Sequeira	Jim

and a new table **`Languages`**:

First Normal Form (contd.)

`Languages`:

Id	Language
2136	English
2136	French
2136	Spanish
2136	Hindi
2245	English
2245	Bengali
2245	French
4124	French
4124	Italian
4124	English
7832	Portuguese
7832	Spanish
7832	Italian

First Normal Form (contd.)

- Splitting the **`Interpreter`** table (2nd version) into two ensures that the new tables are in 1NF, but now a join will be required for some queries.

E.g., list interpreters with Spanish skills:

$$\pi_{\text{FirstName, LastName, Language}} (\sigma_{\text{Language=Spanish}} (\text{Interpreter} \rhd\lhd \text{Languages}))$$

Second Normal Form

- Tables in the second normal form (2NF) satisfy the following property

The table must be in 1NF and, in addition, all non key attributes must be dependent on each candidate key

- Table **`Books`**, which we saw earlier, is in 2NF:

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

Second Normal Form (contd.)

- Suppose table **`Books`** contained a non-key column **`#BooksWritten`** listing the number of books written by the authors.
- **`#BooksWritten`** depends upon the non-key column **`Authors`** but not on the key ISBN. This version of **`Books`** table (below) will not be in 2NF.
- A 1NF table is converted to 2NF by removing all columns that are not dependent on the key columns. These other columns go into another table.

ISBN	Title	Price	Authors	#BooksWritten	...
0929306279	Bell Labs	29.95	Gehani	15	...
0929306260	Java	49.95	Sahni & Kumar	1	...
0670031844	White Moghuls	34.95	Dalrymple	4	...
0439357624	Born Confused	16.95	Hidier	2	...

Second Normal Form (contd.)

- A 1NF table is converted to 2NF by removing all columns that are not dependent on the key columns. These other columns go into another table.
- Note that column **`Authors`** is the atomic type VARCHAR(50) – string type with maximum of 50 characters.

Third Normal Form

- Tables in the third normal form (3NF) satisfy the following property:

The table must be in 2NF and, in addition, no non-key column determines another non-key column.

- Suppose the **`Books`** table contained another column listing a 20% discounted price for Premium Customers.

Third Normal Form (contd.)

- The non-key columns **`Price`** and **`pPrice`** determine each other. This version of the **`Books`** table (below) is thus not in 3NF.
- A 2NF table is converted to 3NF by removing one or more non-key columns so that it does not contain any non-key columns whose values are determined by other non-key columns.

ISBN	Title	Price	pPrice	...
0929306279	Bell Labs	29.95	23.96	...
0929306260	Java	49.95	39.96	...
0670031844	White Mughals	34.95	27.96	...
0439357624	Born Confused	16.95	13.56	...

SQL

- Standardized declarative programming language designed for interacting with relational databases.
  - Expressive programming language
  - Much database interaction can be done in one statement.
  - Provides facilities for creating the database, adding and changing information in the database, querying and viewing the information in the database, and managing information.
- SQL is a set-oriented database query language.
  - Based on relational algebra, which treats tables as mathematical objects.
  - Practical incarnation of the relational algebra.
- SQL provides facilities other than those for manipulating tables such as
  - support for database creation and administration
  - concurrency control to support simultaneously multiple users
  - security, etc
- Most SQL statements return a table as their result allowing these SQL statements to be used wherever a table can be specified.

SQL (contd.)

- Relational algebra views a relation as a set of tuples (rows).
  - No duplicate rows are allowed.
- SQL views relations as tables
  - Allows duplicate rows for practical reasons in
    - a table (but not in a table with a primary key because that will violate the definition of a primary key), and
    - a query result
  - For example, counting the number of customers by projecting the **`Customer`** relation on the state column and then counting the number of customers will yield different results based on whether or not duplicates are allowed.
  - Duplicates can be thrown away in SQL by using the **`DISTINCT`** clause in a **`SELECT`** query.
- We have been informally using relation and table interchangeably.
  - Despite this difference, we will keep doing so.

SQL (contd.)

SQL consists of three parts:

- *Data Definition Language* (DDL): Provides facilities for defining the database structure, such as tables, and for controlling the database.
- *Data Manipulation Language* (DML): Provides facilities for interacting with the database
- *Data Control Language*: Provides facilities for managing the database. Typically these deal with creating views and indexes, security, concurrency control, and transactions.

SQL Basics

- One language for both data definition and data manipulation (and managing the database).
- Each data manipulation operation takes one or more tables as operands.
- Each query returns a table as its result which can be used with other operations.
- Case insensitive.
- Comment lines in MySQL begin with **#.** Or they begin with **/*** and are terminated by */. Standard SQL uses -- for comment lines.
- Database objects such as databases, tables, and columns are identified using identifiers (names) – there is one exception rows are identified using unique values, i.e.,*keys*. An *identifier* is a sequence of letters and digits that must begin with a letter.
  - Names may or not may be case sensitive depending upon the underlying operating system. Safe to assume that names are case insensitive.

SQL Column Types

Column Type	Comments
BOOLEAN	True or false.
CHAR	Single character.
CHAR(*n*)	String of size *n*.
VARCHAR(*n*)	Variable length string with a maximum size equal to *n*.
BLOB	Large binary objects such as images.
TEXT	Same as BLOB, but the sorting is performed on a case-insensitive basis.
INTEGER	Normal-size integer.
INT	Normal-size integer.
SMALLINT	Small integer.
BIGINT	Big integer.
DECIMAL(*precision*, *scale*)	Decimal number; *precision* is the display-width and *scale* is the number of fractional digits – when representing money use a scale of 2.
FLOAT	Single precision floating-point number.
DOUBLE	Double precision floating-point number.
REAL	Synonym for DOUBLE.
DATE	Date in YYYY-MM-DD format.
YEAR	Year in YYYY format.
TIME	Time in HH:MM:SS format.
DATETIME	Time in YYYY-MM-DD HH:MN:SS format.

SQL– Strings

Special character	Meaning when used within a string
\'	single quote character
\"	double quote character
\b	backspace character
\n	new line character
\r	carriage return character
\t	tab character
\\	backslash character
%	without a preceding backslash, the percent character is interpreted by the string operators LIKE and NOT LIKE as a wildcard character matching zero or more characters.
_	without a preceding backslash, the underscore can be interpreted by the string operators LIKE and NOT LIKE as a wildcard character matching a single character.

SQL Operators

OR, XOR
AND
NOT
BETWEEN
=, >=, >, <=, <, <>, !=, LIKE, NOT LIKE, IN
-, +
*, /, DIV, %, MOD
^
– (unary minus)

- BETWEEN *min* AND *max*
- *string* LIKE *pattern-string*
- *string* NOT LIKE *pattern-string*
  - % matches substrings and _ matches arbitrary characters.

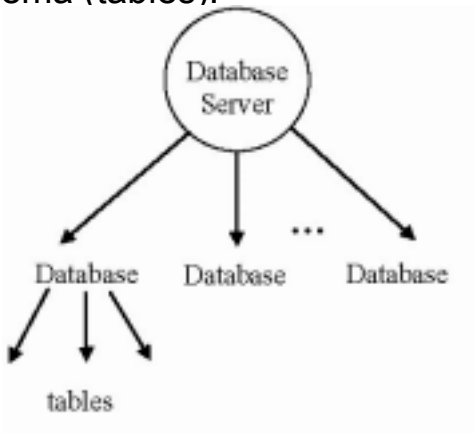
SQL
Data Definition Language

- The data definition language provides facilities for
  - creating and deleting databases,
  - creating, deleting, and modifying tables, and
  - creating and deleting views, etc.
- Data definition language commands are commands of the form:

Action DataItem Details

MySQL

- Tables reside in a database object. A MySQL server handles many databases, each with its own schema (tables):



SQL
Database Creation

1. **CREATE DATABASE** *databaseName*;
2. **CREATE DATABASE IF NOT EXISTS** *databaseName*;

```
mysql> CREATE DATABASE everest;
ERROR 1007 (HY000): Can't create database
      'everest'; database exists
mysql> CREATE DATABASE IF NOT EXISTS NewDB;
Query OK, 1 row affected (0.03 sec)
mysql> CREATE DATABASE IF NOT EXISTS NewDB;
Query OK, 0 rows affected (0.01 sec)
```


SQL
Database Deletion / Specifying Default

- **DROP DATABASE** *databaseName*;

```
mysql> DROP DATABASE NewDB;
Query OK, 1 row affected (0.73 sec)
```

- **USE** *databaseName*;

```
mysql> USE everest;
Database changed
```

SQL
Information About the Database

```
mysql> SHOW TABLES;


+--------------------+
| Tables_in_everest  |
+--------------------+
| books              |
| customers          |
| orderinfo          |
| orders             |
+--------------------+
4 rows in set (0.01 sec)
mysql>
```


SQL
Creating Tables

```
CREATE TABLE tableName (
    one or more column definitions
    zero or more column properties
) ENGINE = InnoDB;
```

- Specifying table type as InnoDB is a MySQL (not SQL) feature needed for transaction-safe tables.
- Each column definition has the form

columnName type properties

- Properties involving more than one column are specified separately from the column definition.

SQL
Creating Tables Example

```
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;
```

- MySQL uses, by default, the MyISAM storage engine which is efficient in storage but not transaction-safe.

SQL
Table Description

```
mysql> DESCRIBE Books;
```

Field	Type	Null	Key	Default	Extra
ISBN	varchar(10)		PRI		
Title	varchar(50)	YES		NULL	
Price	decimal(5,2)	YES		NULL	
Authors	varchar(50)	YES		NULL	
Pages	int(11)	YES		NULL	
PubDate	year(4)	YES		NULL	
Qty	int(11)	YES		NULL	

SQL
Inserting Data In Tables

```
INSERT INTO Books
VALUES('0929306279', 'Bell Labs', 29.95,
          'Gehani', 269, '2003', 121);
INSERT INTO Books
  VALUES('0929306260', 'Java', 49.95,
         'Sahni & Kumar', 465, '2003', 35);
INSERT INTO Books
  VALUES('0670031844', 'White Moghuls',
         34.95, 'Dalrymple', 459, '2003', 78);
INSERT INTO Books
  VALUES('0439357624', 'Born Confused',
         16.95, 'Hidier', 432, '2002', 11);
```


SQL
Table – After Data is Inserted

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

SQL
Table Definition – Customers

```
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;
```


SQL
Table Definition (Contd.)

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

SQL
Table Definition

```
CREATE TABLE OrderInfo (
  OrderId INT(8),
  ISBN CHAR(10),
  Qty INT,
  Price DECIMAL(5,2)
) ENGINE = InnoDB;
```


SQL
Column Properties

- In addition to the column type, SQL allows users to specify other column properties.
  - Default Values
  - Key Specification
  - Constraints
  - Triggers

Default Values

DEFAULT *default-value*

- Exception – primary key columns cannot have default values.

SQL

Column Property – Keys

Type of Key (Key Attribute)	Comments
PRIMARY KEY	A table can have many keys but only one key can be selected as the primary key.
UNIQUE	Column (or columns) can be constrained to have unique values with one exception – duplicate NULL values are allowed. A unique column (or set of columns) is a secondary key, a key other than the primary key.
FOREIGN KEY	Foreign keys are used to implement the referential integrity constraint. The foreign key attribute imposes the restriction on a column value requiring it to be equal to a key in another table.

SQL

Column Property – Keys (contd.)

Primary Key

- Single Column or Multiple Columns

```
PRIMARY KEY
```

or

```
PRIMARY KEY(col1, col2, …)
```

Secondary Key or Unique

- Single Column or Multiple Columns

```
UNIQUE
```

or

```
UNIQUE(col1, col2, …)
```

SQL

Column Property – Constraints

Constraints specify restrictions on values that can be stored

- *Required Values*
  - A column (field) value can be required when data (a row) is inserted or updated with the property

    ```
    NOT NULL
    ```

    By default, field values can be omitted (except the primary key).
- *Check Constraint*
  - Ensures that the values inserted in a table satisfy a specified condition:

    ```
    CHECK(condition)
    ```


SQL

Column Property – Constraints (contd.)

- *Foreign Keys*
  - Implement the referential integrity constraint.
  - define relationships between rows in referencing & referenced tables.
  - Impose restrictions on column values which must equal primary key values in the "foreign" (referenced) table.
  - Called constraint because it imposes a restriction.

    Single Column Foreign Key

    ```
    REFERENCES table(column)
    ```

    Multiple Column Foreign Key

    Must be specified separately as an property

    ```
    FOREIGN KEY(columns) REFERENCES table(columns)
    ```

  - Constraint violation, because of update or delete, causes an action (*trigger*), if specified, to be executed.
  - Resetting to default values can also be specified to address the violation.
  - Otherwise, the *transaction* is aborted (happens also if trigger does not fix violation).

SQL
Triggers

- Actions that are executed by the database server automatically whenever the specified trigger condition is satisfied.
- We will discuss triggers in detail later

SQL
Modifying Table Definitions

- Table definitions can be altered and deleted.
- Note existing queries & applications may be affected & may not work.
- The **`ALTER TABLE`** command can be used to
  - add/delete columns,
  - add/delete indexes
  - change column types
  - rename columns, and so forth.
- E.g. the following command alters the table **`Books`** by adding column **`Binding`** with type **`VARCHAR(32)`** after column **`Pages`**:

```
ALTER TABLE Books
      ADD Binding VARCHAR(32)
      AFTER Pages;
```

SQL
Modifying Table Definitions

- After the execution of this command, MySQL can be asked to display the new description of the table Books with the command
- mysql> DESCRIBE Books;
- MySQL displays the following information which now includes information about the column Binding:

Field	Type	Null	Key	Default	Extra
ISBN	varchar(10)		PRI		
Title	varchar(50)	YES		NULL	
Price	decimal(5,2)	YES		NULL	
Authors	varchar(50)	YES		NULL	
Pages	int(11)	YES		NULL	
Binding	varchar(32)	YES		NULL	
PubDate	year(4)	YES		NULL	
Qty	int(11)	YES		NULL	

SQL
Deleting / Renaming Tables

DROP TABLE *tableName;*

RENAME TABLE *oldName* TO *newName;*

SQL – Data Manipulation

SQL provides facilities for retrieving and modifying data.

- The most powerful is the **SELECT** statement
  - used for extracting data from the database, i.e., extracting rows or portions of rows from one or more tables.
- SQL embeds many capabilities in the **SELECT** statement
  - projection,
  - selection, and
  - join.

SQL – Data Manipulation (contd.)

- Informal description of the simple **SELECT** statement

```
SELECT column names or expressions
FROM tables
WHERE search condition
```

- column names listed specify projection (column selection),
- search condition specifies row selection
- listing multiple tables in the **FROM** clause and specifying the search condition to find associations between values in these tables specifies the join operation.
- column expressions are used to specify computations (discussed in the next section).

A single SELECT statement is often used to perform projection, selection, and join operations together.

SQL – Projection

- The following **`SELECT`** statement generates a list of all the books along with their titles, ISBNs, and prices (but no other information)

```
SELECT Title, ISBN, Price
FROM Books;
```

Title	ISBN	Price
Bell Labs	0929306279	29.95
Java	0929306260	49.95
White Moghuls	0670031844	34.95
Born Confused	0439357624	16.95

SQL – Projection (contd.)

- A `SELECT` statement listing the appropriate fields for all the rows in a table is fine when the table is small.
- However, typically when the table is large, a search condition is used to restrict the size of the result to data of interest.
- Incidentally, the asterisk symbol **`*`** can be used as a short hand for listing all the columns.

SQL – Computations

- Along with projection, SQL allows the user to perform computations in the SELECT statement.
- Suppose we want to prepare a table with book titles, ISBNs, along with prices that are discounted 20%. This can be easily accomplished with the following SQL statement:

```
SELECT Title, ISBN, Price*0.8
FROM Books;
```

which produces the table

SQL – Computations (contd.)

Title	ISBN	Price*0.8
Bell Labs	0929306279	23.96
Java	0929306260	39.96
White Mughals	0670031844	27.96
Born Confused	0439357624	13.56

SQL – Renaming Columns

- Columns in a result table produced by a **SELECT** statement can be renamed. E.g., to change the column name in the previous example from

  Price*0.8

  to

  20% Discounted Price

  the renaming clause

  AS *newName*

  can be used as follows

```
SELECT Title, ISBN,
       Price*0.8 AS '20% Discounted Price'
FROM Books;
```

  producing the table

SQL – Renaming Columns (contd.)

Title	ISBN	20% Discounted Price
Bell Labs	0929306279	23.96
Java	0929306260	39.96
White Moghuls	0670031844	27.96
Born Confused	0439357624	13.56

SQL - Selection

- Selection is the extraction of data (rows) that satisfy specified criteria. from a table. The **SELECT** statement

```
SELECT *
FROM tables
WHERE search condition
```

yields a table with rows that satisfy the *search condition.* ***** is a shorthand for all the columns of the tables in the FROM clause.

- The **SELECT** statement

```
SELECT *
FROM Books
WHERE Price BETWEEN 15.0 AND 30.0;
```

takes the **Books** table and yields

SQL – Selection (contd.)

ISBN	Title	Price	Authors	Pages	PubDate	Qty
0929306279	Bell Labs	29.95	Gehani	269	2003	121
0439357624	Born Confused	16.95	Hidier	432	2002	11

SQL – Selection (contd.)

- To get book titles and their price, then we can combine projection with selection

```
SELECT Title, Price
FROM Books
WHERE Price BETWEEN
            15.0 AND 30.0;
```

to get the table

SQL – Selection (contd.)

Title	Price
Bell Labs	29.95
Born Confused	16.95

SQL – WHERE Clause

- The search condition in the **`WHERE`** clause is an expression that filters out rows for which the expression is not satisfied.
- The search condition is used for
  - comparing values (using operators such as **=**, **<>**, **<**, **>**, **<=**, **>=**, and **`BETWEEN`**),
  - checking for **`NULL`** values,
  - pattern matching, etc,
- Complex conditions can be formed using the **`AND`**, **`OR`**, and **`NOT`** operators.

SQL – Joins

- Data is typically stored in multiple tables to take advantage of data normalization.
- But it would be easier to write queries if all the data was in one *universal* table.
  - Also users will not have to determine which tables contain what information.
- Tables may have to joined to get the needed data.
- Suppose we want to print the titles of the books sold along with the total quantities sold. The information, is split between two tables, **`Books`** and **`OrderInfo`**.
- We need to
  - join these two tables,
  - extract the needed information, and then
  - do some addition (aggregation).

SQL – Joins (Contd.)

- Consider the following two equivalent queries:

```
SELECT *
FROM OrderInfo, Books
WHERE Books.ISBN =
      OrderInfo.ISBN;
```

and

```
SELECT OrderInfo.*, Books.*
FROM OrderInfo, Books
WHERE Books.ISBN =
      OrderInfo.ISBN;
```


SQL – Joins (Contd.)

- Both queries join the two tables **`Books`** and **`OrderInfo`** by pasting the rows that have matching ISBNs.
  - The **`Books`** table has unique ISBNs but the **`OrderInfo`** table can have multiple rows with the same ISBN.
  - Consequently, each row in the **`Books`** table may be pasted to zero or more rows from the table **`OrderInfo`**.
- These queries differ only in how the columns to be displayed are specified.
  - The first query specifies that all the columns in the two tables (denoted by are to be shown **`*`**) are to be shown in the result.
  - The second specifies that only the columns in **`OrderInfo`** (denoted by **`OrderInfo.*`**) and **`Books`** (denoted by **`Books.*`**) are to be shown in the result.
  - The columns that will be displayed are the same in either case.

SQL – Joins (Contd.)

FROM ORDERINFO				FROM BOOKS						
OrderId	ISBN	Qty	Price	ISBN	Title	Price	Authors	Pages	PubDate	Qty
1	0929306279	1	29.95	0929306279	Bell Labs	29.95	Gehani	269	2003	121
1	0929306260	1	49.95	0929306260	Java	49.95	Sahni & Kumar	465	2003	35
2	0439357624	3	16.95	0439357624	Born Confused	16.95	Hidier	432	2002	11
3	0670031844	1	34.95	0670031844	White Moghuls	34.95	Dalrymple	459	2003	78
4	0929306279	1	29.95	0929306279	Bell Labs	29.95	Gehani	269	2003	121
4	0929306260	1	49.95	0929306260	Java	49.95	Sahni & Kumar	465	2003	35
4	0439357624	1	16.95	0439357624	Born Confused	16.95	Hidier	432	2002	11
4	0670031844	1	34.95	0670031844	White Moghuls	34.95	Dalrymple	459	2003	78

SQL – Joins (Contd.)

- There are 2 columns labeled **`Price`** and 2 columns named **`Qty`**.
- Each table contributes one of the duplicate columns.
- To avoid confusion, one duplicate column in each case needs to renamed (requires explicitly listing the columns of at least one table):

```
SELECT OrderInfo.*,
       Title, Books.Price AS CurrentPrice,
       Authors, Pages, PubDate,
       Books.Qty AS Stock
FROM OrderInfo, Books
WHERE Books.ISBN = OrderInfo.ISBN;
```

- Columns **`Price`** & **`Qty`** had to be qualified by the table name to avoid ambiguity.

SQL – Joins (Contd.)

- The result table has extra columns. These columns are easily discarded by simply listing the columns needed:

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

The result of this query is the table

Title	Qty
Bell Labs	1
Bell Labs	1
Born Confused	3
Born Confused	1
White Moghuls	1
White Moghuls	1
Java	1
Java	1

SQL – Joins (Contd.)

- By default, SQL does not eliminate duplicates in query results.
- Eliminating duplicates in the previous result will lead to a loss of information – some books sold.
- Sometimes we may not want duplicates. Suppose we want to list the titles that have sold so far. The **SELECT** statement

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

Title
Bell Labs
Bell Labs
Born Confused
Born Confused
White Moghuls
White Moghuls
Java
Java

SQL – Joins (Contd.)

- Duplicates are not needed in the previous query result.
- To eliminate duplicates, SQL provides the **`DISTINCT`** option:

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

Title
Bell Labs
Born Confused
White Moghuls
Java

SQL – Joins (Contd.)

- Back to the table with titles and quantities.
  - We still need to “aggregate” the quantities sold for each of the different books instead of having this information distributed in multiple rows.

SQL – Aggregation

- Our earlier query

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

yields a result table with information that we need but the information is not in an appropriate form because the number of books sold for each title is not totaled & presented in one line.

SQL – Aggregation (Contd.)

Title	Qty
Bell Labs	1
Bell Labs	1
Born Confused	3
Born Confused	1
White Moghuls	1
White Moghuls	1
Java	1
Java	1

We need to

- group identical titles together, and
- count the total number of books per group.

SQL – Aggregation (Contd.)

- Instead of using loops as in traditional programming languages, SQL provides a high level "aggregation" facility to perform the addition as shown in the following query:

```
SELECT Title, SUM(OrderInfo.Qty)
FROM OrderInfo, Books
WHERE Books.ISBN = OrderInfo.ISBN
GROUP BY Title;
```

- There are two parts of this query that are new.
  - The **`GROUP BY`** clause specifies that all rows with the same **`Title`** value are to be grouped into one row.
  - The **`SUM`** aggregation function specifies that the **`Qty`** value from the **`OrderInfo`** table is to be aggregated (totaled) for each group and associated with the group.

SQL – Aggregation (Contd.)

- The table produced by the above query is almost what we need – with one exception.
  - The name of the second column is not pleasing – it needs a new name.

Title	SUM(OrderInfo.Qty)
Bell Labs	2
Born Confused	4
White Moghuls	2
Java	2

SQL – Aggregation (Contd.)

- Renaming is easily done using the AS clause:

```
SELECT Title,
     SUM(OrderInfo.Qty) AS 'Copies Sold'
FROM OrderInfo, Books
WHERE Books.ISBN = OrderInfo.ISBN
GROUP BY Title;
```

- We now have the table we want:

Title	Copies Sold
Bell Labs	2
Born Confused	4
White Moghuls	2
Java	2

Common Aggregation Functions

Aggregation Function	Computes
AVG (*column*)	Average column value
COUNT (*)	Number of rows
COUNT (*column*)	Number of non-null values
MIN (*column*)	Minimum value in a column
MAX (*column*)	Maximum value in a column
SUM (*column*)	Total of the values in the column

- *column* can be an expression that evaluates to a column.
- Functions **COUNT**, **SUM**, & **AVG** take into account duplicates. Duplicate values are eliminated using the keyword **DISTINCT** as
  - **COUNT(DISTINCT *column*)**,
  - **AVG(DISTINCT *column*)**, and
  - **SUM(DISTINCT *column*)**.

Joins of More Than 2 tables

- Suppose we need to print a report that lists
  - customer ids,
  - the order ids associated with them, and
  - the number of Java books in each order.
- Requires joining 3 tables – **Orders**, **OrderInfo**, **Books** :

```
SELECT CustomerId, Orders.OrderId, Title,
       OrderInfo.Qty
FROM Orders, OrderInfo, Books
WHERE Orders.OrderId = OrderInfo.OrderId
      AND OrderInfo.ISBN = Books.ISBN
      AND Title = 'Java';
```

CustomerId	OrderId	Title	Qty
1	1	Java	1
3	4	Java	1

More About Joins

- SQL provides an explicit **JOIN** operator. Query

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

  can be written using an explicit **JOIN** operator as :

```
SELECT DISTINCT Title
FROM OrderInfo JOIN Books
ON Books.ISBN = OrderInfo.ISBN;
```

- Using the operator **JOIN** is equivalent to using **INNER JOIN**.

More About Joins

- If the names of the columns of the two tables being joined are the same and the join condition is equality, then the **`USING`** clause can be used:

```
SELECT DISTINCT Title
FROM OrderInfo JOIN Books
USING (ISBN);
```

- SQL supports several different types of joins in addition to the default inner join.

More About Joins
Left Outer Join

- We want a list customers, along with order dates.
- The **`Customers`** table includes persons who never placed an order.
- A left outer join on **`Customers`** and **`Orders`** tables allows us to generate such a list
  - For rows that match, a left outer join works like an inner join.
  - For rows in the left table without a matching row in the right table, it appends **`NULL`** values for the columns from the right table.
  - The inner join ignores such rows.
- Here is the query that produces the customer list we need:

```
SELECT First, Last, Company, OrderDate
FROM Customers LEFT OUTER JOIN Orders
ON Id = CustomerId;
```

More About Joins
Left Outer Join (Contd.)

First	Last	Company	OrderDate
Tom	Jones	Acme	2004-03-31
Tom	Jones	Acme	2004-04-01
Susan	Wise		2004-04-01
Liza	Singh	FastTrack	2004-04-02
Alan	Feuer	Clover	NULL
Vinod	Oza	TechSmart	NULL

More About Joins
Left Outer Join (Contd.)

- If we change the order of tables in the left outer join, then we will not get the same result.
  - Result will be different because each row in the left table (**`Orders`**) will have a matching a row in the right table (**`Customers`**).
  - The left outer join will not pick up customers who have not placed an order.

```
SELECT First, Last, Company, OrderDate
FROM Orders LEFT OUTER JOIN Customers
ON Id = CustomerId;
```

More About Joins
Left Outer Join (Contd.)

First	Last	Company	OrderDate
Tom	Jones	Acme	2004-03-31
Tom	Jones	Acme	2004-04-01
Susan	Wise		2004-04-01
Liza	Singh	FastTrack	2004-04-02

More About Joins
Right Outer Join

- The right outer join is like the left outer join except that the focus is on the right table.
- The following query

```
SELECT First, Last, Company, OrderDate
FROM Customers RIGHT OUTER JOIN Orders
ON Id = CustomerId;
```

produces as its result the same table as the rearranged left outer join query.

Nested Queries

- A nested query (subquery) is a query that is embedded in another query
- Nested queries are often used in the **WHERE** clause. The value returned can be
  - of a basic type -- can be compared using the comparison operators,
  - a column – can be compared using the comparison operators followed by the operators **ANY** or **ALL**,
  - tested for membership in the set of values returned (using **IN** or **NOT IN**), and
  - checked to see if a null or a non-null value is returned (operators **EXIST** and **NOT EXIST** are used).
- The nested query must always be enclosed in parentheses.

Nested Queries – IN Operator

- The following query prints names of companies that ordered books in 2004.

```
mysql> SELECT Company
    -> FROM Customers
    -> WHERE Id IN
    ->       (SELECT CustomerId
    ->        FROM Orders
    ->        WHERE OrderDate BETWEEN
    ->        '2004-01-1' AND '2004-12-31');
+-----------+
| Company   |
+-----------+
| Acme      |
|           |
| FastTrack |
+-----------+
4 rows in set (0.00 sec)
mysql>
```

- The blank row – one customer did not have company in the address.

Nested Queries – `EXISTS` Operator

- List titles of all books for which more than one copy has been ordered (on a "line item" basis) in a single order:

```
SELECT Title
FROM Books
WHERE EXISTS
    (SELECT *
      FROM OrderInfo
     WHERE OrderInfo.Qty > 1 AND
           Books.ISBN = OrderInfo.ISBN);
```

- For each book (identified by its ISBN in the nested query) in **`Books`**, its title is printed only if the result of the nested query contains one or more rows
- The **`SELECT`** list in the nested query is not used for anything, so typically **`*`** is used.

Nested Queries – `ALL` Operator

- List the titles of the books with the highest quantity ordered in a single order (on a line item basis):

```
SELECT Title
FROM Books, OrderInfo
WHERE Books.ISBN = OrderInfo.ISBN AND
      OrderInfo.Qty >= ALL (SELECT Qty
                            FROM OrderInfo
                           );
```

- The nested query returns a table with the quantities ordered for every book (on a line item).
- The number of copies of every book ordered (on a line item basis) is then compared with the values in this table, and if this number is greater than or equal to all the values in the table, the comparison evaluates to true.

Nested Queries – Advantages

- Allow the partitioning of a complex query into simpler queries
- Make queries more readable

SQL – Inserting Data into Tables

- We have seen several examples of the **INSERT** statement. Simple version:

```
INSERT INTO table
VALUES(list of values for all columns);
```

- Here is another example:

```
INSERT INTO Customers
VALUES(1, 'Acme', 'Tom',
 'Jones', '25 Scenic Road', 'Hilltop',
 'NJ', '18901', '9189088919');
```

- This form requires values for all the fields.
- Keyword **DEFAULT** sets a field to its default value (otherwise, field will be set to the **NULL** value).

SQL – Inserting Data into Tables (contd.)

- If all field values are not going to be supplied, then the following version of the **INSERT** can be used:

```
INSERT INTO table(columns)
VALUES(values for the columns listed);
```

- The **INSERT** statement

```
INSERT INTO Customers(Id, Company, Zip)
VALUES(22, 'Lucent', '07974');
```

- will insert **NULL** values for the omitted columns.

Id	Company	First	Last	Street	City	State2	Zip	Tel
1	Acme	Tom	Jones	25 Scenic Road	Hilltop	NJ	18901	9189088919
...								
22	Lucent	NULL	NULL	NULL	NULL	NULL	07974	NULL

SQL – Deleting Data

- Rows that match specified criterion can be deleted with a single **DELETE** statement. Simple form:

```
DELETE FROM table
WHERE expression;
```

- Example:

```
DELETE FROM Customers
WHERE Id=22 OR Id=23;
```

will delete the rows with Id equal to 22 or 23.

SQL – Modifying Data

- Values in a table are modified using the **`UPDATE`** statement. Simple form:

```
UPDATE table SET column = expression1
WHERE expression2;
```

  which sets the values of *column* elements to *expression*1 for all the rows that satisfy *expression*2.
- If the **`WHERE`** clause is left out, then the values of all the rows will be updated.
- The following **`UPDATE`** statement will increase, by 10%, the price of all the books in the **`Books`** table:

```
UPDATE Books
SET Price = Price * 1.1;
```

  The **`Books`** table will change from

SQL – Modifying Data (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

To

ISBN	Title	Price	Authors	Pages	PubDate	Qty
0929306279	Bell Labs	32.95	Gehani	269	2003	121
0929306260	Java	54.95	Sahni & Kumar	465	2003	35
0670031844	White Moghuls	38.45	Dalrymple	459	2003	78
0439357624	Born Confused	18.65	Hidier	432	2002	11

SQL – Data Control Language

- The data control language part of SQL relates to facilities for managing the database. Typically these deal with
  - views,
  - triggers,
  - indexes,
  - security,
  - concurrency control,
  - transactions etc.
- We will be discussing these facilities in depth in the ensuing chapters.
- **Note:** Indexes are not part of standard SQL because they relate to the physical, not logical, organization of the data.
  - They are used for improving query access times.
  - They used to be part of SQL but were removed from SQL.
  - Most database specific SQLs provide facilities for indexes because access speed is a critical in database use.

Stored Procedures

- Stored routines allow a set of SQL commands to be compiled and stored on the database server.
  - They can then executed by referencing their names.
  - Stored routines are more efficient than executing the same SQL commands because they do not have to be transmitted to the server or compiled every time.
  - Stored procedures allow parameterization of the SQL commands.
  - They also allow an expert to write complex queries for use by others.
- Two kinds of stored routines:
  - Procedures
  - Function (not discussed)
- Procedures definitions use syntax of the form

```
CREATE PROCEDURE pname(parameter declarations)
        statement;
```

- If the procedure consists of multiple statements, then **BEGIN ATOMIC** / **END** must be used.
- A procedure is executing by referencing it using a statement of the form

```
CALL procedureName(arguments);
```

Stored Procedures (contd.)

```
SELECT OrderInfo.ISBN, Title,
       SUM(OrderInfo.Qty) AS Quantity,
       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 <= '2004-04-02'
GROUP BY OrderInfo.ISBN, Title;
```

- Above query produces the following table

ISBN	Title	Quantity	Sales
0439357624	BornConfused	4	67.80
0670031844	White Moghuls	2	69.90
0929306260	Java	1	49.95
0929306279	Bell Labs	1	29.95

Stored Procedure Definition

```
CREATE PROCEDURE Sales(S Date, E Date)
BEGIN ATOMIC
    SELECT OrderInfo.ISBN, Title,
       SUM(OrderInfo.Qty) AS Quantity,
       SUM(OrderInfo.Qty*OrderInfo.Price)AS Sales
    FROM OrderInfo, Orders, Books
    WHERE OrderInfo.ISBN = Books.ISBN AND
       OrderInfo.OrderId = Orders.OrderId AND
       Orders.ShipDate >= S AND
       Orders.ShipDate <= E
    GROUP BY OrderInfo.ISBN, Title;
END;
```

Stored Procedures in MySQL

```
mysql> delimiter //
mysql> CREATE PROCEDURE Sales(S Date,E Date)
    -> BEGIN
    -> SELECT OrderInfo.ISBN, Title,
    ->    SUM(OrderInfo.Qty) AS Quantity,
    ->    SUM(OrderInfo.Qty*OrderInfo.Price)
    ->        AS Sales
    -> FROM OrderInfo, Orders, Books
    -> WHERE OrderInfo.ISBN = Books.ISBN AND
    ->    OrderInfo.OrderId = Orders.OrderId
    ->                                    AND
    ->    Orders.ShipDate >= S AND
    ->    Orders.ShipDate <= E
    -> GROUP BY OrderInfo.ISBN, Title;
    -> END;
    -> //
Query OK, 0 rows affected (0.00 sec)
mysql> delimiter ;
```


Stored Procedures in MySQL

- To get the sales information for the first two days of April, we can now simply call procedure **`Sales`** as follows:

```
CALL Sales('2004-04-01','2004-04-02');
```

Everest Books

Orders, Invoices & Reports

- We will now illustrate the use of SQL to
  - enter an order, and
  - generate invoices and reports.
- SQL does not provides much in the way of formatting facilities.
- In case of complex applications and those that require non-trivial formatting facilities
  - SQL is typically used from within the context of a host programming language such as Java using classes that allow users to intermix Java and SQL.
  - But we will use SQL directly (not from within a host language) using its limited data formatting.

Everest Books

Orders, Invoices & Reports (contd.)

- Order Leading to an Invoice
  - Updating the database to record an order and then generating the corresponding invoice is a multi-step process involving
  - entering the customer information, if needed, into the database,
  - entering the order information after checking to ensure that the books are in the inventory and that they are available for this order (they are not allocated to another customer order), and
  - generation of the actual invoice.
- In practice, all the above steps should happen as a single atomic action.
- The invoice generation process should not stop in the middle nor should it be affected by other orders.
- We will not worry about ensuring atomicity of the multiple steps – we will be discussing that later.

Invoice

EVEREST BOOKS
2300 GREAT SCENIC VIEW
SUMMIT TOP, VT 08211

Ship Date 2/4/04 Invoice # 004 Customer Id # 003
Order Date 2/4/04

Liza Singh
FastTrack
155 Route 133
Holmdel, FL 48901

ISBN	Title	Qty	Unit Price	Total
0929306279	Bell Labs	1	29.95	29.95
0929306260	Java	1	49.95	49.95
0439357624	Born Confused	1	16.95	16.95
0670031844	White Moghuls	1	34.95	34.95
			Subtotal	131.80
			Sales Tax	0.00
			Shipping	6.99
			TOTAL DUE	138.79

Inovice

- We will not be able to print such a nice looking invoice by directly using SQL. As mentioned earlier, data formatting typically requires the use of SQL from within a host language.
- The sales report to be generated is

Sales Report

EVEREST BOOKS
SALES REPORT
4/1/04 to 4/2/04

ISBN	Title	Qty	Book Sales
0929306279	Bell Labs	1	29.95
0929306260	Java	1	49.95
0439357624	Born Confused	4	67.80
0670031844	White Moghuls	2	69.90
	TOTAL for period		$217.60

Recording the Order
(Shown in the Invoice)

- The customer is

```
Liza Singh
FastTrack
155 Route 133
Holmdel, FL 48901
Tel no: 218-555-2223
```

- Steps in inserting customer information in database:
  - Check if customer is in database
  - if yes, find the customer's id.
  - If no, assign a new id to the customer.
  - Insert customer information in the database.
- Query checking to see if customer exists:

```
SELECT Id
FROM Customers
WHERE Company = 'FastTrack'
        AND First = 'Liza'
        AND Last = 'Singh';
```

Liza is not in the database. Customer ids are assigned sequentially starting from 1.

Recording the Order (contd.)

- Determine largest id value been assigned and use this plus1 as new id
- The largest id assigned is determined by the query

```
SELECT MAX(Id)
FROM Customers;
```

- Assume at this time that there are only two customers in the database.
  - The id to be assigned for the next customer is 3:

```
INSERT INTO Customers
VALUES(3, 'FastTrack', 'Liza', 'Singh',
       '155 Route 133', 'Holmdel', 'FL',
       '48901','2185552223');
```

- Assigning a id to a new customer can be automated using user variables (not standard SQL). Alternatively, if SQL is being used from within a host language such as Java, then Java facilities can be used.
- MySQL user variables have the form
  `@variableName`
- They are assigned values using the **SET** statement e.g.,
  `SET @newid = 0;`

Recording the Order (contd.)

- User Variables can also be assigned values within other statements by using the assignment operator := in places where expressions are allowed. For example:

```
SELECT @newid := MAX(Id) + 1
FROM Customers;

INSERT INTO Customers
VALUES(@newid, "FastTrack",
     "Liza", "Singh",
     "155 Route 133", "Holmdel", "FL",
     "48901", "2185552223");
```

Recording the Order (contd.)

- We also need to assign a new id to the order for the above customer. We determine the largest order id used so far as

```
SELECT MAX(OrderId)
FROM Orders;
```

- The largest order id is 3. The new order id will be 4.
- We insert order information in the tables **Orders** and **OrderInfo**. First in **Orders**:

```
INSERT INTO Orders
VALUES(4,3,'2004-04-2','2004-04-2',0.0,0.0);
```

- Computing new customer and order ids is painful.
- Fortunately, the MySQL column property **AUTO_INCREMENT** can be used to automatically supply a new value, one more than the last highest value used in the column, for a new row. E.g., if the **OrderId** column of **Orders** is defined with the **AUTO_INCREMENT** property:

Recording the Order (contd.)

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

- With **AUTO_INCREMENT** we can now omit the **OrderId** value and let MySQL do the computing as in

```
INSERT INTO Orders(CustomerId, OrderDate,
                   ShipDate,Shipping, SalesTax)
VALUES(3,'2004-04-2','2004-04-2',0.0,0.0);
```

Recording the Order (contd.)

- Using `AUTO_INCREMENT`, the user or application will not have to worry about computing a new value for the `OrderId`. MySQL will automatically provide a new value for `OrderId` (starts with 1). Incidentally, function `LAST_INSERT_ID()` can be used to retrieve the last `OrderId` inserted, e.g.,

```
SELECT LAST_INSERT_ID();
```

- Continuing with our invoice example, the order details are inserted into table `OrderInfo` as follows:

```
INSERT INTO OrderInfo
       VALUES(4, '0929306279', 1, 29.95);
INSERT INTO OrderInfo
       VALUES(4, '0929306260', 1, 49.95);
INSERT INTO OrderInfo
       VALUES(4, '0439357624', 1, 16.95);
INSERT INTO OrderInfo
       VALUES(4, '0670031844', 1, 34.95);
```


Recording the Order (contd.)

- The price info & availability information was found using the `Books` table.
- The inventory can be updated as follows:

```
UPDATE Books
SET Qty = Qty - 1
WHERE ISBN = '0929306279';

UPDATE Books
SET Qty = Qty - 1
WHERE ISBN = '0929306260';

UPDATE Books
SET Qty = Qty - 1
WHERE ISBN = '0439357624';

UPDATE Books
SET Qty = Qty - 1
WHERE ISBN = '0670031844';
```

Recording the Order (contd.)

- The shipping amount is reflected in the table **`Orders`** after determining the number of books being shipped. For four books, the shipping charge is $6.99 ($3.99 for the first book and $1 for each additional book):

```
UPDATE Orders
SET Shipping = 6.99
WHERE OrderId = 4;
```

- Note that determining the order and customer ids, computing the shipping info, updating the inventory, determining the price, etc. could all be automated using MySQL facilities or by using host language facilities if SQL is being used from within a host language. And from a user perspective, a GUI needs to be provided to enter the data.
- The order is now in the Everest database. We still have to print the invoice using the information in the database – our next step

Printing the Invoice

- We will extract the information in the order needed for the invoice.
  - This information will need to be assembled and formatted properly to print the invoice.
  - We will not do this here, because SQL does not provide the needed formatting capabilities.
- Extracting the invoice information will be based primarily on the order id. The query

```
SELECT ShipDate,OrderId,CustomerId,OrderDate
FROM Orders
WHERE OrderId = 4;
```

- extracts the following information needed for the top part of the invoice (following the Everest Books address whose location is fixed to be on top of the invoice):

ShipDate	OrderId	CustomerId	OrderDate
2004-04-02	4	3	2004-04-02

Printing the Invoice (contd.)

- Now we need to extract the customer information.
  - We can do this using the order id which will requires determining the customer id and using it to extract the customer information.
  - If the customer id is known, as should be the case, it can be used directly

```
SELECT First, Last, Company, Street,
       City, State2, Zip
FROM Customers
WHERE Id = 3;
```

extracts the customer information:

First	Last	Company	Street	City	State2	Zip	Tel
Liza	Singh	FastTrack	155 Route 133	Holmdel	FL	48901	2185552223

Printing the Invoice (contd.)

- Now here is the query extracting information about the books ordered (based on the order id):

```
SELECT Books.ISBN, Books.Title,
  OrderInfo.Qty, OrderInfo.Price,
  OrderInfo.Price*OrderInfo.Qty
FROM Books, OrderInfo
WHERE Books.ISBN = OrderInfo.ISBN
      AND OrderId = 4;
```

ISBN	Title	Qty	Price	OrderInfo.Price*OrderInfo.Qty
0929306279	Bell Labs	1	29.95	29.95
0929306260	Java	1	49.95	49.95
0439357624	Born Confused	1	16.95	16.95
0670031844	White Moghuls	1	34.95	34.95

Printing the Invoice (contd.)

- Now we determine the cost for the books (subtotal) & the sales tax, extract the shipping cost, & compute the grand total. First the subtotal & sales tax:

```
SELECT SUM(OrderInfo.Price*OrderInfo.Qty)
          AS SubTotal,
       SUM(OrderInfo.Price*OrderInfo.Qty)
          * SalesTax AS SalesTaxAmount
FROM  OrderInfo, Orders
WHERE OrderInfo.OrderId =
            Orders.OrderId AND
            Orders.OrderId = 4;
```

SubTotal	SalesTaxAmount
131.80	0

Printing the Invoice (contd.)

- The shipping cost is extracted from the **`Orders`** table using the query

```
SELECT Shipping
FROM Orders
WHERE OrderId = 4;
```

which produces

Shipping
6.99

Printing the Invoice (contd.)

- The total cost of the order is computed using the query

```
SELECT SUM(OrderInfo.Price*OrderInfo.Qty)*
       (1+SalesTax) + Shipping AS Total
FROM  OrderInfo, Orders
WHERE OrderInfo.OrderId = Orders.OrderId
GROUP BY Orders.OrderId
HAVING Orders.OrderId = 4;
```

which yields the following information:

Total
138.75

Printing the Invoice (contd.)

- We can compute the subtotal, sales tax, shipping cost, and total in one SQL statement

```
SELECT SUM(OrderInfo.Price*OrderInfo.Qty)
           AS SubTotal,
       SUM(OrderInfo.Price*OrderInfo.Qty) *
           SalesTax AS SalesTax,
       Shipping,
       SUM(OrderInfo.Price*OrderInfo.Qty) *
        (1+SalesTax) + Shipping AS Total
FROM  OrderInfo, Orders
WHERE OrderInfo.OrderId = Orders.OrderId
GROUP By OrderInfo.OrderId
HAVING Orders.OrderId = 4;
```

The above query produces the following table:

SubTotal	SalesTax	Shipping	Total
131.80	0	6.99	138.79

Sales Report

- The Sales report shown is for the first two days of April 2004.
- Two dates are recorded in the Everest Books database.
  - The order date and the ship date.
- Because we are not told which date to use, we use the ship date.
- The sales report lists the sales period. We will leave printing the sales period to an application – printing headings, text, etc., is not part of SQL.
- Query for first part of Sales report (excludes total sales for period). We use the stored procedure **`Sales`** Defined earlier

Sales Report (contd.)

```
CALL Sales('2004-04-01','2004-04-02');
```

ISBN	Title	Quantity	Sales
0439357624	BornConfused	4	67.80
0670031844	White Moghuls	2	69.90
0929306260	Java	1	49.95
0929306279	Bell Labs	1	29.95

Sales Report (contd.)

- This query computes total sales in the first 2 days of April 2004:

```
SELECT SUM(OrderInfo.Qty*OrderInfo.Price)
          AS 'Total'
FROM OrderInfo, Orders
WHERE OrderInfo.OrderId = Orders.OrderId
  AND
     ShipDate >= '2004-04-01' AND
     ShipDate <= '2004-04-02';
```

Total
217.60

We now have information for the report, but not in a nice format.

Sales Report (contd.)

- SQL does not provide facilities for generating nicely formatted reports.
- One has to go beyond SQL. E.g.,
  - SQL tables and results can be saved into files and then manipulated by other tools
  - Using SQL from within a host language and using the host language facilities for formatting.

Transactions

- A *consistent* (or *valid*) database is one whose data satisfies all the constraints specified in the database.
- Examples of constraints are
  - specifying a column to have non null values,
  - specifying a column as a primary key (no duplicate or null values), or
  - specifying the range of allowed column values.

Transactions

- Most database systems ensure that users accessing a database and manipulating data do not make the database inconsistent.
- Databases accomplish this by using a *transaction – defined as*

  an exchange or transfer of goods, services, or funds ("electronic" transactions)
- Strict rules are enforced that
  - govern how multiple simultaneous transactions update databases and
  - control how transactions read updates made by transactions that have not completed execution.

Transactions (contd.)

- Some examples of common database transactions:
  - A stock trade.
  - A deposit in a bank.
  - A hotel reservation.
- A transaction can consist of many actions.
- E.g., executing an order for an Everest Books customer involves executing the following group of statements:
  - check if the book is available,
  - if yes, then enter the
    - customer information,
    - order information,
    - payment information, etc.

Transactions (contd.)

- A transaction takes a database from one consistent state to another.
- If a transaction tries to take the database to an inconsistent state, then the database system will “kill” (abort) the transaction and undo its changes, if any.

Transaction Correctness

- Besides being consistent, a database must also be “correct.”
- A *correct* database is one that is consistent and satisfies “external” correctness properties.
  - External because the database system does not know about them and they cannot be checked or be enforced by it.
  - For example, if the Everest Books database contains incorrect book prices, the database system cannot do anything about them since it has no knowledge about correct book prices.
- Ideally, a transaction should take a database from one correct state to another.
  - Will happen only if the transaction is written correctly.
- Since a database system does not know about correctness, it can only guarantee that a transaction will take a database from one consistent state to another.
  - Database systems ensure this by aborting transactions that violate constraints.

Transaction Correctness (contd.)

- Proving that a transaction is written correctly is a non trivial task, especially for complex transactions.
- Consequently, most programmers test programs (such as transactions) for “correctness.”
- Unfortunately, from a practical perspective, testing cannot be used to prove the correctness of programs.
  - Testing demonstrates the presence of errors but not their absence.
  - Only by exhaustive testing (using all possible inputs) can a program be guaranteed to be correct.
  - In most cases, exhaustive testing is not a realistic option because of the amount of testing required.
- In lieu of being able to prove programs correct, most programmers build confidence in the correctness of their programs by
  - understanding the code,
  - testing as much as is reasonable, and
  - having others look at and test their code.

Transaction Properties

- A database transaction is an action that takes a database from one “consistent” (valid) state to another.
- A transaction cannot be executed partially – it is either executed in its entirety or not at all.
- Transactions also allow a group of statements to be executed as one logical “atomic” action.
- Transactions allow multiple users to simultaneous access and update the database while guaranteeing that transactions will not interfere with each other.
  - In there is potential of interference, the system may delay execution of some transactions (or even abort them).

Transactions Properties (contd.)

- Simultaneously execution of multiple transactions can lead to higher throughput & faster response times compared to executing them serially.
- Each transaction gets the illusion that it is operating in isolation, i.e., in single-user mode.
- Database systems guarantee that simultaneous execution of multiple transactions will not cause the database to become inconsistent
  - by ensuring that such execution corresponds to some serial (sequential) execution of these transactions.

Transactions Properties (contd.)

- Suppose there is only one copy of a book in the Everest Books inventory.
  - Two customer agents should **NOT** be able to sell the one copy to their customer.
  - Only one agent should be able to see this information and the other agent forced to wait until the first agent is done.
  - The second agent will then see that there is either one copy or none in stock.
- To increase concurrency, some database systems may
  - allow the two agents to see that one copy is available,
  - but will allow only one of them to complete the sale
  - the other agent's transaction will be aborted.
  - In this case, the agent can deduce that some other agent made the sale first thus weakening/eliminating the single-user mode illusion.

Transactions in SQL

- SQL statements execute as transactions in one of two modes.
  - *auto (implicit) commit* mode in which each SQL statement is executed as a transaction – default mode.
  - *explicit commit* mode in which a group of SQL statements can be executed as one transaction.
- Explicit Commit Transaction syntax:

```
START TRANSACTION;
      SQL statements
COMMIT;
```

- The **ROLLBACK** statement can be used to abort a transaction:

```
START TRANSACTION;
      SQL statements
ROLLBACK;
```

Transactions Example

- Suppose we want to change the order with OrderId equal to 4 by
  - deleting the book with ISBN 0670031844
  - reduce the shipping charge by $1.00.
  - Note that the invoice total is not stored but will have to be calculated on demand based on the information stored.
- This requires two changes:
  - deleting one row in table OrderInfo
  - updating the shipping charge in table Orders.
- Both these changes must occur together or not at all. Otherwise, the database will be inconsistent.

Transactions Example (Contd.)

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

Transaction Example (contd.)

```
START TRANSACTION;
    DELETE FROM OrderInfo
    WHERE OrderId = 4 AND
          ISBN = '0670031844';

    UPDATE Orders
    SET Shipping = Shipping - 1
    WHERE OrderId = 4;
COMMIT;
```


Transactions
Informal Definition

- A database *transaction* is one logical action
  - that consists of one or more component actions – collectively executed as a single action;
  - that is executed in effective isolation even in the presence of other transactions running simultaneously; the database system ensures no "conflicts" occur (see next slide);
  - whose execution either happens in its entirety or does not happen
    - all or nothing semantics, no partial execution;
  - whose changes become permanent after it *commits*
    - a transaction is said to *commit* after it has successfully executed and changes made by it have been written to the log (on disk);
  - that leaves the database in a consistent state even if the system crashes;
  - whose, if aborted or rolled back before it commits, changes are effectively not applied to the database (if applied they are undone).

Transaction Conflict

- Two simultaneously executing transactions are said to *conflict* if they access the same data item with one of them updating the item.
- A conflict makes the order of execution of the two simultaneously executing transactions significant
- One transaction may
  - be required to commit before the other
  - have to be aborted

  to ensure that the conflict does not cause inconsistent semantics.

Transactions

Formal Definition – ACID Properties

- Database transactions are actions with the properties of
  - **a**tomicity,
  - **c**onsistency,
  - **i**solation, and
  - **d**urability.

  collectively known as the ACID model properties:

Transactions
Atomicity

- A transaction is either executed completely or not at all.
- Partial executions are not allowed
  - may cause database to become inconsistent.
- Protects against database server crashes.
- Although a transaction is atomic from the user perspective but for the database server it consists of many operations.

Transactions
Consistency

- A “correctly written” transaction operating on a consistent database will leave it in a consistent state.
- Database constraints satisfied before the execution of the transaction must be satisfied after its execution
  - even though during the execution of the transaction they may temporarily not be satisfied.

Transactions
Isolation

- A transaction must execute as if it was executing alone even though there may be other simultaneously executing transactions.

Transactions
Durability

- Updates of a transaction that has successfully executed must be permanently reflected in the database.
- It is the responsibility of the database system to ensure that the updates will be permanent.
- A transaction is said to have successfully executed when the changes made by it have been are recorded in a log (on disk). The changes are applied to the database after writing to the log.
- Recording the changes in the log is what makes the transaction's changes permanent even if the computer system crashes before the changes are written to the database.
  - Upon system recovery, the log will be examined for changes that need to be reflected in the database.
  - If there are such changes, they will be applied to the database.

Transactions Example

- Consider a money "transfer" transaction in a bank that transfers money from one account.
  - We want to transfer $100 from account number 1001 to account number1005.
  - If account 1001 does not have at least $100, the transaction does nothing to the database.

Transactions Example (Contd.)

- **`Accounts`** Table – Before & After Transfer (**`ActNum`** is the primary key)

ActNum	First	Last	Balance
1001	Mary	Brown	2300.03
1002	Jim	Black	300.00
1004	Susan	Greene	1300.00
1005	Arti	White	900.00

ActNum	First	Last	Balance
1001	Mary	Brown	2200.03
1002	Jim	Black	300.00
1004	Susan	Greene	1300.00
1005	Arti	White	1000.00

Transactions Example (contd.)

- First, the transaction needs to see if there is at least $100 in the account.

```
START TRANSACTION;
        SELECT ActNum, Balance
        FROM Accounts
        WHERE ActNum = 1001 OR ActNum = 1005;
```

- If the result is less than 100, then the transaction should be aborted by executing

```
ROLLBACK;
```

- Otherwise, the transfer can proceed by executing the statements:

```
        UPDATE Accounts
        SET Balance = Balance – 100
        WHERE ActNum = 1001;
        UPDATE Accounts
        SET Balance = Balance + 100
        WHERE ActNum = 1005;
COMMIT;
```


Transactions Example (contd.)

- Ensuring that there is enough money in the account is part of the transaction:
  - between the **BEGIN** and **COMMIT** statements
  - or, between the **BEGIN** and **ROLLBACK** statements
  - otherwise, another transaction can possibly change the amount in the account before the transfer takes place.
- The money transfer can be made conditional in SQL.
  - SQL-99 has the **IF-THEN-ELSE** conditional statement.
- The transaction code can be embedded in an application written, for example, in Java using JDBC to connect to the database.
  - In such a case, a Java conditional statement can be used to determine whether the transaction should be committed or rolled back.

Transactions Example & ACID Properties

- The transfer transaction has three parts.
  - It first checks the account balance.
  - Then if the balance is less than $100, the transaction is aborted.
  - Otherwise, the money is transferred and the transaction committed.
- ACID Guarantees:
  - **Atomicity**: The amount withdrawn from one account and its deposit to the other either succeeds or fails but there is no partial execution.
  - **Consistency**: No constraints will be violated (only constraint is the primary key column `ActNum` which is not impacted by the transaction).
  - **Isolation**: The transfer transaction will correctly move $100 from one account to another even in the presence of other simultaneously executing transactions that may be interested in modifying the same accounts.
  - **Durability**: Once the transaction has successfully executed, its effects will be reflected in the database, i.e., they will become permanent.

Transactions Serializability

- Database systems allow simultaneous execution of multiple transactions for better performance.
- To show that a set of simultaneously executing transactions do not interfere or conflict with each other, it suffices to show that their execution is equivalent to some serial (sequential) execution.
  - Simultaneous execution of transactions on different parts of a database trivially corresponds to a sequential execution.
  - The same applies for read-only transactions.
  - Interesting cases arise when simultaneously executing transactions read and update common objects.

Transactions Serializability (contd.)

- Suppose simultaneously executing transactions T1 and T2 both want to access item **`Qty`** of the book B.
- It is the database system's responsibility to ensure that the simultaneous execution of T1 and T2 is equivalent to either
  - T1 executes before T2 or
  - T2 executes before T1.
- Here are the different possibilities:
  - T1 and T2 both read **`Qty`** from the customer database but do not change it. This simultaneous execution is equivalent to either of the two orders, T1 before T2, or T2 before T1.
  - T1 and T2 read **`Qty`**, then T2 changes **`Qty`**.
    - In this case, there is only one order that is equivalent to the simultaneous execution and that is T1 before T2.
  - T1 and T2 both read **`Qty`**, both change **`Qty`**. In this case, the simultaneous execution cannot be equivalent to any sequential order.
    - Therefore one of the two transactions must be aborted and restarted so that it reads the new value.

Ensuring Serializability of Transactions

- Serializability of transactions can be ensured by allowing
  - only one transaction at a time,
  - many simultaneous read transactions but no update transactions,
  - many simultaneous transactions as long as they operate on different parts of the database, or
  - many simultaneous transactions and preventing conflicts by delaying some transactions and aborting others.
- Allowing multiple transactions to execute simultaneously while ensuring serializability
  - complicates the implementation of a database system but
  - it does reduce response time and maximize throughput
- In case of Everest Books database, there are many opportunities for conflicts.
  - For example, there will be conflicts between order transactions for the same book – lots of people may want to order the same book in a short span of time, say soon after a book has won a prestigious award like the Pulitzer Prize.
  - The database item over which the conflict occurs is the number of copies Qty of the book in stock.

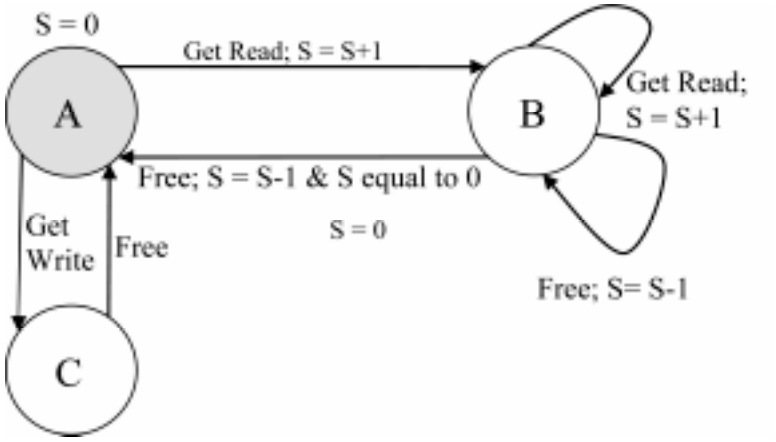
Locks

- Conflicts between transactions that simultaneously want to access the same database items are prevented by using ***locks.***
  - they are variables used as flags or semaphores.
- Locks are associated with database items such as tables and rows.
- A lock indicates the sharing status of a database item – if it is being used by a transaction and whether the transaction will just be reading its value or updating it.
- The simplest lock is a ***binary lock***.
  - Two states indicating that the associated database item is either available for reading or writing or not.
  - Such locks do not allow multiple transactions to read the same database item simultaneously.
  - Variations of the binary lock with additional states allow multiple read transactions but only one update transaction.
- A transaction requests read permission by requesting a "read" lock and update permission by requesting a "write" lock:
- A read or write lock must be acquired by a transaction before it is allowed to read or update a database item, respectively.

Locks (contd.)

- Locks are implemented using variables that can be updated by only one transaction at a time.
- To understand how locks are implemented, one needs to understand the states of a lock.
- A lock's state indicates whether or not the associated database item is
  - free for reading or updating,
  - free for reading, or
  - not free (either for reading or updating)
- The following state diagram illustrates how the state of a lock changes it accepts read requests (Read) and write requests (Write), and when a transaction that is done with a lock frees the lock (Free):

Locks (contd.)



Locks – SQL

- SQL users do not have to worry about locking because it is done implicitly.
- SQL code, when translated, contains code to acquire and release locks.
- Consider the following SQL query that computes the total sales in April 2004.

```
START TRANSACTION;
    SELECT SUM(OrderInfo.Qty*OrderInfo.Price)
    FROM OrderInfo, Orders
    WHERE OrderInfo.OrderId = Orders.OrderId
          AND ShipDate >= '2004-04-01'
          AND ShipDate <= '2004-04-30';
COMMIT;
```

Locks (contd.)

- This transaction needs to access the tables **`OrderInfo`** and **`Orders`**. These two tables should not be updated while the query is computing the total sales.
- Here is what some of the generated code may look like:

```
get_read_lock("OrderInfo");
get_read_lock("Orders");
   Code for computing the total sales
free_read_lock("OrderInfo");
free_read_lock("Orders");
```


Lock Granularity

Lock Granularity	Comments
Database	The whole database is locked. This scheme allows one transaction at a time to update the database, but multiple transactions can read from the database. It reduces concurrency but makes lock implementation very simple and reduces the overhead of managing locks. If the update transactions are very small, fast, and few, then such a scenario could lead to good performance.
Table	The whole table is locked. Provides much more concurrency than database level locking.

Lock Granularity (contd.)

Row	A row of a specific table is locked. Increases concurrency but complicates lock management.
Field	A specific field of a row is locked. Offers a high degree of concurrency, but complicates lock management significantly.
Predicate	Rows satisfying a predicate (a Boolean expression) are locked. Predicate locks offer a lot of flexibility but are hard to implement efficiently. Rows are not individually locked – they are locked as a set.

Pros of Row Locking

- Good results for databases subject to large number of transactions and high throughput.
  - Fewer lock conflicts when accessing different rows in different transactions.
  - Fewer changes for rollbacks because the rows that may have changed are easy to identify.
  - A single row can be locked for a long time without affecting most other transactions – a table locked for a long time will slow the system because it locks all the rows in a table.

Cons of Row Locking

- Row locking takes more memory than table locking
  - each row locked requires a lock
- Row locking takes longer than table locking
  - many more locks are required
- Row locking will be definitely much worse than table locking if a scan of the whole table is required (as in aggregation).
- Row locking can lead to the “phantom” problem
  - a new row satisfying the locking criteria is inserted into a table after the rows have been locked (discussed later)

Locks Required for a Query

- Consider the query we saw earlier, i.e.,

```
START TRANSACTION;
SELECT
  SUM(OrderInfo.Qty*OrderInfo.Price)
  FROM OrderInfo, Orders
  WHERE OrderInfo.OrderId =
        Orders.OrderId
   AND ShipDate >= '2004-04-01'
   AND ShipDate <= '2004-04-30';
COMMIT;
```

Locks Required for Query (contd.)

Lock Granularity	Locks Acquired
database	Read lock for the whole Everest Books database.
table	Read locks for the `OrderInfo` and `Orders` tables.
row	Read locks for all the rows of the `Orders` table with appropriate `ShipDate` values and for the rows in the `OrderInfo` table with `OrderId` values matching those of the rows selected from the `Orders` table.
field	Read locks for the fields `OrderId` and `ShipDate` of table `Orders` and the fields `OrderId`, `Qty`, and `Price` of table `OrderInfo` – only for the rows selected for row locking (above).

Phantom Problem – Row Locking

- Consider a transaction T that queries only one table.
- Before T is executed, all the rows in the table satisfying the **`WHERE`** expression of the **`SELECT`** statement in T are locked.
- Then, but before T has committed, another transaction arrives, adds a new row that also happens to satisfy the **`WHERE`** expression of T, and commits.
- T will not "see" the new row even though it will be in the table before T commits. The new row, called the "phantom" row because it was not present initially, should be part of the T's computation.

Phantom Problem & Serialization

- Consider transaction T1 that retrieves all orders of a customer whose customer id is 91:

```
SELECT *
FROM Orders
WHERE CustomerId = 91;
```

- And a simultaneous transaction T2 that wants to insert another order for the customer with id equal to 91.

Phantom Problem & Serialization Locking Options

1. Table **`Orders`** locked
   - T2 must wait for T1 to complete and commit.
2. Rows in **`Orders`** with customer id equal to 91 locked
   - T2 does not have to wait for T1 to commit before inserting a new order in the **`Orders`** table.

- Assume T1 starts before T2. Concurrent execution of T1 and T2 must be equivalent to some serial order:
  - If T1 commits before T2 commits, then the existence of an equivalent order is trivially obvious.
  - If T1 commits after T2 commits, then there is a problem. T1 should have seen the order inserted by T2. But this is not the case if T1 locks (and reads) the rows of **`Orders`** table before the row inserted by T2.

Phantom Problem (contd.)

- T1 committing after T2 means that T1 did not execute in isolation because it did not read the new row inserted by T2.
- To avoid this problem, T1 must commit before T2.
- Under row locking semantics, this cannot be guaranteed.
  - No guarantee on serialization since if T1 commits after T2 there will be a conflict over a customer order (row) that did not exist when T1 started.
- Solution of the phantom row problem requires preventing future insertions of rows that match the criteria used by T1 to select rows which it locked – until after it commits.

Phantom Problem (contd.)
Solution

- `Orders` table is locked (lock released after T1 commits).
  - T2 will not be able to insert the order until after T1 has committed.
  - Inefficient since it forces all transactions, even those that do not conflict with T1 (say those inserting orders for different customers) to be delayed until after T1 commits.
- Predicate locks are used to lock the set of rows of the customer with customer id equal 91.
  - Predicate locking does not suffer from the phantom row problem because predicate lock checking is dynamic.
  - The predicate lock, in our example, will checked before the insertion of every row into the `Orders` table.
  - The attempt by T2 to add a row with customer id equal to 91 will be delayed until after T1 commits.

Phantom Problem (contd.)

- Predicate locks are a good conceptual tool but they are expensive to implement
  - they must be evaluated for every row insertion.
- Databases such as MySQL lock ***indexes*** (data structures for fast table access) using a technique called *next-key* or *index record* locking, that produces results similar to row locking but without the phantom problem.
  - Instead of locking the rows directly, portions of the index that point to the rows are locked.
  - Index record locking requires an index on the search field.
- Of course, all this locking happens behind the scenes.

Users do not have to worry about locks.

Starvation

- Multiple transactions can simultaneously hold a read lock but only one transaction can have a write lock.
  - Transactions requesting read locks automatically get preferential treatment since they do not have to wait for the read locks to be released.
  - A transaction requesting a write lock for an item has to wait until all the read locks on the item are released.
- It is possible that a transaction requesting a write lock will never get it
  - because of a steady arrival of transactions requesting read locks.
- A transaction requesting a write lock can be "starved" by being forced to wait forever.

Such a transaction is said to be *starved*.

Starvation

- To avoid starvation of a transaction requesting a write lock:
  - Suspend granting read locks when a write lock request is pending.
  - However, with this strategy, transactions requesting read locks can starve.
- Refine strategy by
  - honoring a read lock request, if read lock requests are pending, every time between two successive requests for write locks.

Once again, all this all happens behind the scenes.

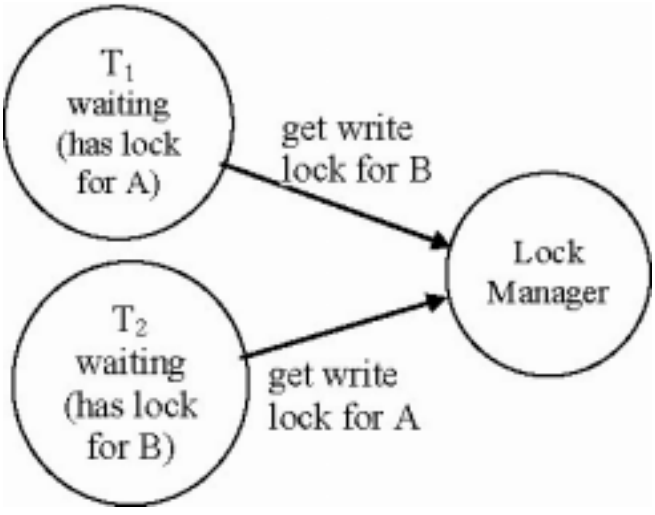
Deadlocks

- Locks can cause a ***deadlock,*** a state in which
  - two or more transactions are each waiting for the other(s) to release a lock and
  - are unable to make progress until this happens.

Deadlocks

- Consider two transactions T1 and T2
  - each wants to update the same two database items, A and B
  - but they request locks in different orders
  - T1 wants write locks, first for the database item A and later one for B.
  - T2 wants write locks, first for the database item B and later one for A.
- The database system's lock manager grants locks as follows:
  - T1 requests a write lock for A and gets it.
  - T2 requests a write lock for B and gets it.
  - T1 requests a write lock for B and is told to wait until it becomes available.
  - T2 requests a write lock for A and is told to wait until it becomes available.

Deadlocks (contd.)



Deadlocks (contd.)

- At this point, transactions T1 and T2 are said to be in deadlock.
  - T1 is waiting for T2 to release the lock for B
  - T2 is waiting for T1 to release the lock for A
- No progress will occur unless some radical action is taken.
- Databases resolve deadlocks by aborting one or more of deadlocked transactions so that the remaining transactions can proceed.
- To break the deadlock one of T1 or T2 has to be aborted
  - Databases automatically detects a deadlock and aborts or rolls back a transaction.
  - When aborted, a transaction releases the locks its holding.
  - So if T1 is aborted, it will release the lock for A that it holds
  - T2 will then be able to get the lock allowing it to proceed.
  - After is aborted, T1 is rescheduled for execution.

Deadlocks (contd.)

- There are several schemes for preventing deadlocks, E.g.
- Scheme 1
  1. Transactions get all the locks they need at the very beginning.
  2. Transactions must release all the locks immediately if they do not get the locks they want and try again.
  3. Potential problem
     - A transaction may not be able to get the locks it wants because every time it tries to get the locks, one or more locks may not be available.
     - Sophisticated algorithms are used to avoid this problem.
- Scheme 2
  1. The database items are linearly ordered.
  2. Locks for these items must be requested in increasing order.
  3. If a lock is not available, the transaction is forced to wait until the lock is free.

Deadlocks (contd.)

Back to the Example

- Scheme 1
  - T1 and T2 will have to request locks for both A and B at the beginning.
  - One of T1 and T2 will get the locks while the other is forced to wait until the first transaction finishes.
- Scheme 2
  - Assume that A precedes B in the linear ordering specified for lock acquisition.
  - This will force both T1 and T2 to first request the lock for A and then the lock for B.
  - One of T1 and T2 will get the lock for A and the other will be forced to wait until the lock is free.
  - The successful transaction can then request the lock for B.

Deadlock is avoided in both cases.

Locks & Serializability

- Using locks to control access to resources does not by itself guarantee that concurrent execution of transactions will correspond to some serial execution.
- To ensure serializability
  - Transactions must follow some protocols when acquiring and releasing locks.
  - These protocols guarantees serializability but they do reduce concurrency.
  - One such protocol known as the *two-phase locking protocol*.
  - The *two-phase* refers to the lock acquisition and release phases that a transaction must follow.
- We will discuss three variations of the two-phase protocols and how they lead to serializability.

Locks & Serializability

Basic 2-Phase Locking

- Ensures serializability of concurrently executing transactions by requiring every transaction to
  - acquire locks it needs as it proceeds (acquisition or growth phase),
  - release locks when done using the associated database items (shrinking or release phase),
  - perform all lock acquisitions before any lock release.
- The two-phase locking protocol suffers from
  - *Deadlocks*
    - Since a transaction acquires locks as it needs them, it may end up waiting for a lock held by a 2nd transaction which is holding a lock wanted by the 1st transaction.
  - *Cascading Aborts*
    - A transaction T releases locks when it is done using the associated database items but before it commits or is aborted.
    - Other transactions are free to get the released locks and access the associated database items.
    - These transactions may have be to be aborted if T is aborted – to prevent dirty reads.

Locks & Serializability

Basic 2-Phase Locking – Dirty Read Example

- Transaction T1
  - locks table **`Books`**,
  - updates the table,
  - releases the lock.
- Then, before T1 commits or aborts, transaction T2
  - locks **`Books`**,
  - reads data from it,
  - releases the lock.
- Although T2 has completed execution
  - it cannot commit until after T1 commits because it is relying on T1's update to **`Books`**.
  - In case T1 aborts or is aborted, T2 will have to be aborted since the updates made by T1 will no longer be valid.

Locks & Serializability

Basic 2-Phase Locking Example (contd.)

- Informally, we can see that the basic 2-phase locking protocol will ensure that the execution of transactions T1 and T2 is serializable.
  - If T2 uses the updates made by T1, then the database system will ensure that T2 commits after T1.
  - And if T1 aborts, then T2 will also be aborted.
- There may be a series of aborts.
  - Just like T2's fate depends upon that of T1, there may be other transactions whose fate depends upon that of T1 (if they are also executing simultaneously with T1 and depend upon T1) or on the fate of T2, and so on.
  - These transactions will also need to be aborted if T1 is aborted.

Locks & Serializability

Conservative 2-Phase Locking

- Guarantees serializability without deadlocks & cascading aborts.
- Requires that a transaction
  - acquire all locks before it starts;
    - if all locks are not available, transaction releases all locks and waits until they are available;
  - Deadlocks do not occur because all locks are acquired in the beginning.
  - There is no partial lock acquisition and waiting -- scenario that leads to deadlocks.
- Aborts can occur because transactions will be able to read updates made by uncommitted transactions.
  - Transactions will be able to perform dirty reads.
- Reduces concurrency by requiring a transaction to acquire all locks at the beginning even though it may not need them until much later.
  - May increase execution time of transactions as it may take them longer to get locks.

Locks & Serializability

Strict 2-Phase Locking

To increase concurrency, a transaction can acquire locks as it needs them
  - can lead to deadlocks.
- Eliminates cascading aborts by requiring a transaction to release locks only when it commits or aborts (as part of the commit or abort)
  - prevents dirty reads.

Strict two-phase locking protocol is commonly used.

Locks & Serializability

Example

Consider the following 2 transactions that execute simultaneously:

- Transaction **`NewOrder`**
  - updates tables **`Orders`**, **`OrderInfo`**, and **`Books`** to place an order for an existing customer.
- Transaction **`CancelOrder`**
  - update tables **`OrderInfo`**, **`Orders`**, and **`Books to`** cancel an order

Locks & Serializability

Example – Basic 2 Phase Locking

- Transactions acquire locks as needed but release them only after no more locks are to be acquired and when the locks are no longer needed.

Deadlock

- **`NewOrder`** needs write locks for the tables: **`Books, Orders, OrderInfo`**
- **`CancelOrder`** needs write locks for the following tables: **`Orders, OrderInfo, Books`**
- These two transactions happen to execute as follows
  - **`NewOrder`**: Get lock for table **`Books`**
  - **`CancelOrder`**: Get lock for table **`Orders`**
  - **`NewOrder`**: Get lock for table **`Orders`** ... waiting
  - **`CancelOrder`**: Get lock for table **`Books`** ... waiting

One transaction must be aborted!

Locks & Serializability

Example – Basic 2 Phase Locking

Dirty Read

- Another execution scenario:
  - **`NewOrder`**: Get locks for **`Books`**, **`Orders`**, & **`OrderInfo`**
  - **`NewOrder`**: Make updates & complete execution
  - **`NewOrder`**: Release locks (before committing)
  - **`CancelOrder`**: Get locks for **`Books`**, **`Orders`**, & **`OrderInfo`**
  - **`CancelOrder`**: Make updates & complete execution
  - **`CancelOrder`**: release locks (before committing)
  - **`CancelOrder`**: Commit ... forced to wait

Locks & Serializability

Example – Basic 2 Phase Locking

Dirty Read (contd.)

- Going forward there are two scenarios:
  - **`NewOrder`**: Aborts
  - **`CancelOrder`**: Aborted (relied on updates made by **`NewOrder`** which will be undone because **`NewOrder`** aborted)

  or
  - **`NewOrder`**: Commits
  - **`CancelOrder`**: Commits

Locks & Serializability

Examples (contd.)

Conservative Two-Phase Locking

- Transactions **`NewOrder`** and **`CancelOrder`** will not deadlock
- Cascading aborts are possible.

Locks & Serializability

Examples (contd.)

Strict Two-Phase Locking

- Transactions **`NewOrder`** and **`CancelOrder`** may deadlock
  - transactions are allowed to acquire locks as they need them
- No cascading aborts.

Locks & Serializability

Multi-Version Locking

- Database system keeps multiple versions of database items (typically rows) to increase concurrency by allowing a transaction to read one version while another version is being updated.
- The cost of this concurrency is more storage.
- Transactions may see an older version of a database item
  - Database will guarantee that the transaction sees a consistent view of the database by guaranteeing serializability.

Locks & Serializability

Two-Version Locking

Most common version of multi-version locking.

- When transaction T acquires a write lock on a database item, a new version of the item is created.
- T works on the new version.
- Meanwhile, other transactions can read the old version.
- When a write lock is outstanding on a database item, read locks are allowed but on the old version only.
- When T is ready to commit, there must be no other transactions with read locks on items for which T has the write locks.
  - All transactions with read locks on the old versions of the database items write locked by T must have committed or aborted.
  - Otherwise, T must wait until the read locks are freed.

SQL Isolation Levels

- Concurrent transactions can be made to execute completely independently, that is, in **isolation**
  - by ensuring that they do not "conflict" with each other.
- A transaction does not conflict with another concurrently executing transaction if it
  - does not or cannot read updates made by the other transaction
  - update database items being read by the other transaction.
- In this scenario, transactions are executing in complete isolation
- Such execution is said to be "serializable"
  - there exists an equivalent serial execution producing the same results.
- Executing transactions in serializable mode is the safest mode
  - results are guaranteed to be equivalent to a serial execution.

SQL Isolation Levels (contd.)

- In serializable mode
  - a transaction waits to access the database items it needs until the transaction that preceded it in locking these items commits or aborts.
  - waiting reduces concurrency.
- To minimize or eliminate waiting & increase throughput (amount of work per time unit), database systems offer a choice of less than complete isolation.
  - leads to non-serializable transactions,
  - results may not be repeatable.
- Relaxing the complete isolation requirement of the serializable mode can lead to
  - dirty reads
  - non-repeatable reads
  - phantom" problem

SQL Isolation Levels (contd.)

- To minimize or eliminate waiting & increase throughput, SQL allows
  - transactions to read the changes made by a transaction that has not committed.
    - Values read are "dirty" since they are not guaranteed to be final values because they may be changed again or the changes undone if the transaction aborts.
- SQL specifies 4 levels of transaction isolation reflecting different levels of correctness of the values read.

The degree of isolation of a transaction is called its *isolation level.*

SQL Isolation Level 1
READ UNCOMMITED

- No isolation between transactions.
- Reads are non-locking reads
  - database items are read without locking.
- Reads can thus be dirty reads, and may not be consistent
  - they may not come from the same “snapshot” of the database
  - snapshot is a consistent view of the database, that is, one that contains only changes made by committed transactions.

SQL Isolation Level 2
READ COMMITTED

- Same as Level 1 isolation but without dirty reads.
- Reads are consistent
  - the value read reflects changes made only by committed transactions.
- However, the reads are not repeatable because between two reads
  - the database item read may have been changed by a transaction that commits its changes.
- Consider, for example, a transaction T1 that reads a row from a table.
  - Another transaction comes along, updates the row, and commits.
  - T1 will see the updated value when it reads the row again.

SQL Isolation Level 3
REPEATABLE READ

- The reads are repeatable with transactions seeing a consistent view of the database.
- To ensure that the reads are repeatable, the database item to be read is locked for reading until the transaction commits.
- Transactions see changes made by transactions that committed before the item was locked and their own changes but do not see changes made by later transactions or uncommitted transactions.
- Level 3 isolation does not suffer from dirty or inconsistent reads (from different snapshots) but it does suffer from the phantom problem.

SQL Isolation Level 4
SERIALIZABLE

- Supports serial execution of transactions.
- Transactions see a consistent view of the database.
- Like the REPEATABLE READ isolation level except phantoms are prevented!
- Transactions may be aborted and applications or users must be prepared to re-execute them.

SQL Isolation Levels Examples
Isolation Levels 3 and 2

- Consider two users, U1 and U2, interacting with a MySQL database server to insert into and query records from a table **TEAM** defined as follows:

```
CREATE TABLE TEAM(
    First VARCHAR(30),
    Last VARCHAR(30)
) ENGINE = InnoDB;
```


SQL Isolation Levels Example
Repeatable Read – Level 3 (MySQL Default)

	U_1 Input	U_1 Output	U_2 Input
1	START TRANSACTION;		
2			START TRANSACTION;
3	SELECT * FROM Team;	*empty*	
4			INSERT INTO Team VALUES(Tom, Gere);
5			INSERT INTO Team VALUES(Bill, Clay);
6	SELECT * FROM Team;	*empty*	
7			COMMIT;
8	SELECT * FROM Team;	*empty*	
9	COMMIT;		
10	SELECT * FROM Team;	Tom Gere Bill Clay	

SQL Isolation Levels Example

Repeatable Read – Level 3 (MySQL Default)

- U1 sees the empty table **`Team`** even after U2 has inserted a pair of rows into **`Team`** and committed.
- Only after U1 commits, can U1 see the rows inserted by U2. This ensures repeatable reads.
- If is important to read the latest committed value of a database item, then in MySQL, the **`SELECT`** statement can be used in locking mode, e.g.,

```
SELECT *
FROM Team IN LOCK SHARE MODE;
```

as shown below in row number 8 (grayed):

SQL Isolation Levels Example

Repeatable Read–Level 3 (MySQL Lock Share Mode)

	U_1 Input	U_1 Output	U_2 Input
1	START TRANSACTION;		
2			START TRANSACTION;
3	SELECT * FROM Team;	*empty*	
4			INSERT INTO Team VALUES (Tom, Gere);
5			INSERT INTO Team VALUES (Bill, Clay);
6	SELECT * FROM Team;	*empty*	
7			COMMIT;
8	SELECT * FROM Team LOCK IN SHARE MODE;	Tom Gere Bill Clay	
9	SELECT * FROM Team;	*empty*	
10	COMMIT;		
11	SELECT * FROM Team;	Tom Gere Bill Clay	

SQL Isolation Levels Example

Repeatable Read – Level 3 – LOCK SHARE MODE

- In **LOCK SHARE MODE**, the **SELECT** statement reads the latest values of the specified database items.
- Repeating such reads will not necessarily yield the same values.
- In case, the database items reflect changes made by as yet uncommitted transactions, then the **SELECT** statement will be forced to wait until the transactions commit.

SQL Isolation Levels Example

Read Committed – Level 2

	U_1 Input	U_1 Output	U_2 Input
0	SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;		SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;;
1	START TRANSACTION;		
2			START TRANSACTION;
3	SELECT * FROM Team;	*empty*	
4			INSERT INTO Team VALUES (Tom, Gere);
5			INSERT INTO Team VALUES (Bill, Clay);
6	SELECT * FROM Team;	*empty*	
7			COMMIT;
8	SELECT * FROM Team;	Tom Gere Bill Clay	
9	COMMIT;		
10	SELECT * FROM Team;	Tom Gere Bill Clay	

Transactions in MySQL

- MySQL, by default, operates in the *auto commit* mode
  - Each SQL statement is treated as a transaction.
  - Multiple SQL statements can be grouped together into a single transaction using the brackets

```
START TRANSACTION;
…
COMMIT;
```

or

```
BEGIN;
…
COMMIT;
```

- To abort a transaction, instead of the `COMMIT` use the statement

```
ROLLBACK;
```


Locking in MySQL/ InnoDB

- Uses row-level multi-version two-phase locking.
- Presents a consistent version of the database, a single snapshot of the database, for reading to the transaction.
  - Transaction will see changes made by other transactions that committed before the snapshot was taken
  - Transaction will not see changes made after the snapshot was taken.

Specifying Isolations Levels

- MySQL supports all the four SQL isolation levels .
  - The default Isolation is **REPEATABLE READ**.
- The isolation level can be changed using the using the **SET TRANSACTION** statement

```
SET [SESSION | GLOBAL]
     TRANSACTION ISOLATION LEVEL
     {READ UNCOMMITTED | READ COMMITTED
     | REPEATABLE READ | SERIALIZABLE};
```

- Users can specify the isolation level for the current session using the keyword **SESSION** or for all future sessions using **GLOBAL**.
  - By default the isolation level is set for the next statement.