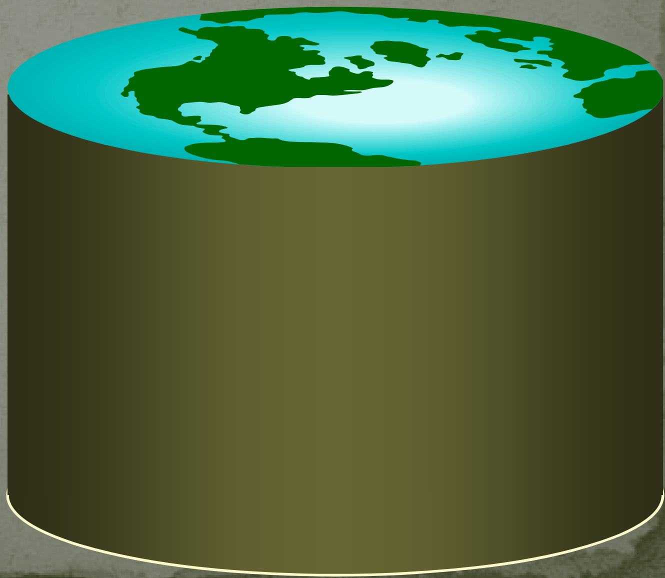
CS 505: Intermediate Topics to Database Systems

Instructor: Jinze Liu

Fall 2008



GROUP BY

- `SELECT … FROM … WHERE …`
  `GROUP BY` *list_of_columns*`;`

- Example: find the average GPA for each age group
  - ```
    SELECT age, AVG(GPA)
    FROM Student
    GROUP BY age;
    ```
    ```

Operational semantics of GROUP BY

`SELECT … FROM … WHERE … GROUP BY …;`

- Compute FROM
- Compute WHERE
- Compute GROUP BY: group rows according to the values of GROUP BY columns
- Compute SELECT for each group
  - For aggregation functions with DISTINCT inputs, first eliminate duplicates within the group
  - ☞ Number of groups = number of rows in the final output

Example of computing GROUP BY

```
SELECT age, AVG(GPA) FROM Student GROUP BY
```

sid	name	age	gpa
1234	John Smith	21	3.5
1123	Mary Carter	19	3.8
1011	Bob Lee	22	2.6
1204	Susan Wong	22	3.4
1306	Kevin Kim	19	2.9

Compute `GROUP BY`: group rows according to the values of `GROUP BY` columns

sid	name	age	gpa
1234	John Smith	21	3.5
1123	Mary Carter	19	3.8
1306	Kevin Kim	19	2.9
1011	Bob Lee	22	2.6
1204	Susan Wong	22	3.4

Compute `SELECT` for each group

age	gpa
21	3.5
19	3.35
22	3.0

Aggregates with no GROUP BY

- An aggregate query with no GROUP BY clause represent a special case where all rows go into one group

Compute aggregate over the group

```
SELECT AVG(GPA) FROM Student;
```

sid	name	age	gpa
1234	John Smith	21	3.5
1123	Mary Carter	19	3.8
1011	Bob Lee	22	2.6
1204	Susan Wong	22	3.4
1306	Kevin Kim	19	2.9

sid	name	age	gpa
1234	John Smith	21	3.5
1123	Mary Carter	19	3.8
1011	Bob Lee	22	2.6
1204	Susan Wong	22	3.4
1306	Kevin Kim	19	2.9

Group all rows into one group

gpa
3.24

Restriction on `SELECT`

- If a query uses aggregation/group by, then every column referenced in `SELECT` must be either
  - Aggregated, or
  - A `GROUP BY` column

☞This restriction ensures that any `SELECT` expression produces only *one* value for each group

Examples of invalid queries

- `SELECT SID, age FROM Student GROUP BY age;`
  - Recall there is one output row per group
  - There can be multiple SID values per group
- `SELECT ~~SID,~~ MAX(GPA) FROM Student;`
  - Recall there is only one group for an aggregate query with no `GROUP BY` clause
  - There can be multiple SID values
  - Wishful thinking (that the output SID value is the one associated with the highest GPA) does NOT work

HAVING

- Used to filter groups based on the group properties (e.g., aggregate values, `GROUP BY` column values)
- `SELECT … FROM … WHERE … GROUP BY … HAVING` *`condition`*`;`
  - Compute `FROM`
  - Compute `WHERE`
  - Compute `GROUP BY`: group rows according to the values of `GROUP BY` columns
  - Compute `HAVING` (another selection over the groups)
  - Compute `SELECT` for each group that passes `HAVING`

HAVING examples

- Find the average GPA for each age group over 10
  - ```
    SELECT age, AVG(GPA)
    FROM Student
    GROUP BY age
    HAVING age > 10;
    ```
  - Can be written using WHERE without table expressions
- List the average GPA for each age group with more than a hundred students
  - ```
    SELECT age, AVG(GPA)
    FROM Student
    GROUP BY age
    HAVING COUNT(*) > 100;
    ```
  - Can be written using WHERE and table expressions

Table expression

- Use query result as a table
  - In set and bag operations, FROM clauses, etc.
  - A way to "nest" queries
- Example: names of students who are in more clubs than classes

```
SELECT DISTINCT name
FROM Student,
     (SELECT SID FROM ClubMember)
     EXCEPT ALL
     (SELECT SID FROM Enroll) ) AS S
WHERE Student.SID = S.SID;
```

Scalar subqueries

- A query that returns a single row can be used as a value in `WHERE`, `SELECT`, etc.
- Example: students at the same age as Bart

What's Bart's age?

```
SELECT *
FROM Student
WHERE age = ( SELECT age
              FROM Student
              WHERE name = 'Bart');
```

- Runtime error if subquery returns more than one row
  - Under what condition will this runtime error never occur?
    - *name* is a key of *Student*
- What if subquery returns no rows?
  - The value returned is a special `NULL` value, and the comparison fails

`IN` subqueries

- `x IN (`*subquery*`)` checks if *x* is in the result of *subquery*
- Example: students at the same age as (some) Bart

What's Bart's age?

```
SELECT *
FROM Student
WHERE age IN (SELECT age
              FROM Student
              WHERE name = 'Bart'
             );
```

`EXISTS` subqueries

- `EXISTS (`*subquery*`)` checks if the result of *subquery* is non-empty
- Example: students at the same age as (some) Bart
  - ```
    SELECT *
    FROM Student AS s
    WHERE EXISTS (SELECT * FROM Student
                  WHERE name = 'Bart'
                  AND age = s.age);
    ```
  - This happens to be a correlated subquery—a subquery that references tuple variables in surrounding queries
    ```

Operational semantics of subqueries

- ```
  SELECT *
  FROM Student AS s
  WHERE EXISTS (SELECT * FROM Student
                WHERE name = 'Bart'
                AND age = s.age);
  ```
- For each row `s` in `Student`
  - Evaluate the subquery with the appropriate value of `s.age`
  - If the result of the subquery is not empty, output `s.*`
- The DBMS query optimizer may choose to process the query in an equivalent, but more efficient way (example?)
  ```

Next Topic

- Functional Dependency.
- Normalization
- Decomposition
- BCNF

Motivation

- How do we tell if a design is bad, e.g., *WorkOn(<u>EID</u>, Ename, <u>PID</u>, Pname, Hours)*?
  - This design has *redundancy*, because the name of an employee is recorded multiple times, once for each project the employee is taking

EID	PID	Ename	Pname	Hours
1234	10	John Smith	B2B platform	10
1123	9	Ben Liu	CRM	40
1234	9	John Smith	CRM	30
1023	10	Susan Sidhuk	B2B platform	40

Why redundancy is bad?

- Waste disk space.
- What if we want to perform update operations to the relation
  - INSERT an new project that no employee has been assigned to it yet.
  - UPDATE the name of “John Smith” to “John L. Smith”
  - DELETE the last employee who works for a certain project

EID	PID	Ename	Pname	Hours
1234	10	John Smith	B2B platform	10
1123	9	Ben Liu	CRM	40
1234	9	John Smith	CRM	30
1023	10	Susan Sidhuk	B2B platform	40

Functional dependencies

- A functional dependency (FD) has the form *X* -> *Y*, where *X* and *Y* are sets of attributes in a relation *R*
- *X* -> *Y* means that whenever two tuples in *R* agree on all the attributes in *X*, they must also agree on all attributes in *Y*
  - $t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$

X	Y	Z
a	b	c
a	b	d

Must be "b"

Could be anything, e.g. d

FD examples

Address (street_address, city, state, zip)

- *street_address, city, state -> zip*
- *zip -> city, state*
- *zip, state -> zip*?
  - This is a trivial FD
  - Trivial FD: LHS ⊇ RHS
- *zip -> state, zip*?
  - This is non-trivial, but not completely non-trivial
  - Completely non-trivial FD: LHS ∩ RHS = ?

Keys redefined using FD's

Let *attr*(*R*) be the set of all attributes of *R*, a set of attributes *K* is a (candidate) key for a relation *R* if

- *K* **->** attr(*R)* - *K*, and
  - That is, *K* is a "super key"
- No proper subset of *K* satisfies the above condition
  - That is, *K* is minimal (full functional dependent)
- *Address* (*street_address*, *city*, *state*, *zip*)
  - {street_address, city, state, zip} — Super key
  - {street_address, city, zip} — Super key
  - {street_address, zip} — Key
  - {zip} — Non-key

Reasoning with FD's

Given a relation *R* and a set of FD's F

- Does another FD follow from F?
  - Are some of the FD's in F redundant (i.e., they follow from the others)?
- Is *K* a key of *R*?
  - What are all the keys of *R*?

Attribute closure

- Given *R*, a set of FD's F that hold in *R*, and a set of attributes *Z* in *R*:
  The closure of *Z* (denoted Z^+) with respect to F is the set of all attributes $\{A_1, A_2, \ldots\}$ functionally determined by *Z* (that is, Z -> $A_1 A_2 \ldots$)
- Algorithm for computing the closure
  - Start with closure = *Z*
  - If *X* -> *Y* is in F and *X* is already in the closure, then also add *Y* to the closure
  - Repeat until no more attributes can be added

A more complex example

WorkOn(<u>EID</u>, Ename, email, <u>PID</u>, Pname, Hours)

- *EID -> Ename, email*
- *email -> EID*
- *PID -> Pname*
- *EID, PID -> Hours*

(Not a good design, and we will see why later)

Example of computing closure

- F includes:
  - *EID* -> *Ename, email*
  - *email* -> *EID*
  - *PID* -> *Pname*
  - *EID, PID* -> *Hours*
- { *PID, email* }$^+$ = ?
- closure = { *PID, email* }
- *email* -> *EID*
  - Add *EID*; closure is now { *PID, email, EID* }
- *EID* -> *Ename, email*
  - Add *Ename, email*; closure is now { *PID, email, EID, Ename* }
- *PID* -> *Pname*
  - Add *Pname;* close is now { *PID, Pname, email, EID, Ename* }
- *EID, PID* -> *hours*
  - Add *hours*; closure is now all the attributes in *WorksOn*

Using attribute closure

Given a relation R and set of FD's F

- Does another FD X -> Y follow from F?
  - Compute X^+ with respect to F
  - If $Y \subset X^+$, then X -> Y follow from F
- Is K a super key of R?
  - Compute K^+ with respect to F
  - If K^+ contains all the attributes of R, K is a super key
- Is a super key K a key of R?
  - Test where $K' = K - \{ a \mid a \in K\}$ is a superkey of R for all possible a

Rules of FD's

- Armstrong's axioms
  - Reflexivity: If $Y \subset X$, then $X \rightarrow Y$
  - Augmentation: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
  - Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
- Rules derived from axioms
  - Splitting: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
  - Combining: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$

Using rules of FD's

Given a relation *R* and set of FD's F

- Does another FD *X* -> *Y* follow from F?
  - Use the rules to come up with a proof
- Example:
  - F includes:
    EID -> *Ename, email*; *email* -> *EID*; *EID, PID* -> *Hours, Pid* -> *Pname*
  - *PID, email* ->*hours*?
    email -> *EID* (given in F)
    PID, email -> *PID, EID* (augmentation)
    PID, EID -> *hours* (given in F)
    PID, email -> *hours* (transitivity)

Example of redundancy

- *WorkOn* (*EID*, *Ename*, *email*, *PID*, *hour*)
- We say *X* -> *Y* is a *partial dependency* if there exist a *X'* ⊂ *X* such that *X'* -> *Y*
  - *e.g. EID, email-> Ename, email*
- Otherwise, *X* -> *Y* is a *full dependency*
  - *e.g. EID, PID -> hours*

EID	PID	Ename	email	Pname	Hours
1234	10	John Smith	jsmith@ac.com	B2B platform	10
1123	9	Ben Liu	bliu@ac.com	CRM	40
1234	9	John Smith	jsmith@ac.com	CRM	30
1023	10	Susan Sidhuk	ssidhuk@ac.com	B2B platform	40

Normalization

- A *normalization* is the process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations
- A *normal form* is a certification that tells whether a relation schema is in a particular state

2^{nd} Normal Form

- An attribute *A* of a relation *R* is a *nonprimary attribute* if it is not part of any key in *R*, otherwise, *A* is a *primary attribute*.
- *R* is in (general) 2^{nd} normal form if every nonprimary attribute *A* in *R* is not partially functionally dependent on *any* key of *R*

X	Y	Z	W
a	b	c	e
b	b	c	f
c	b	c	g

$X , Y \rightarrow Z, W$ $\quad\Rightarrow\quad$ (X , Y , W)

$Y \rightarrow Z$ $\quad\quad\quad\quad$ (Y , Z)

2^{nd} Normal Form

- Note about 2^{nd} Normal Form
  - by definition, every nonprimary attribute is functionally dependent on every key of *R*
  - In other words, *R* is in its 2^{nd} normal form if we could not find a partial dependency of a nonprimary key to a key in *R*.

Decomposition

EID	PID	Ename	email	Pname	Hours
1234	10	John Smith	jsmith@ac.com	B2B platform	10
1123	9	Ben Liu	bliu@ac.com	CRM	40
1234	9	John Smith	jsmith@ac.com	CRM	30
1023	10	Susan Sidhuk	ssidhuk@ac.com	B2B platform	40

Decomposition

Foreign key

EID	Ename	email
1234	John Smith	jsmith@ac.com
1123	Ben Liu	bliu@ac.com
1023	Susan Sidhuk	ssidhuk@ac.com

EID	PID	Pname	Hours
1234	10	B2B platform	10
1123	9	CRM	40
1234	9	CRM	30
1023	10	B2B platform	40

- Decomposition eliminates redundancy
- To get back to the original relation:

Decomposition

- Decomposition may be applied recursively

EID	PID	Pname	Hours
1234	10	B2B platform	10
1123	9	CRM	40
1234	9	CRM	30
1023	10	B2B platform	40

PID	Pname
10	B2B platform
9	CRM

EID	PID	Hours
1234	10	10
1123	9	40
1234	9	30
1023	10	40

Unnecessary decomposition

EID	Ename	email
1234	John Smith	jsmith@ac.com
1123	Ben Liu	bliu@ac.com
1023	Susan Sidhuk	ssidhuk@ac.com

EID	Ename
1234	John Smith
1123	Ben Liu
1023	Susan Sidhuk

EID	email
1234	jsmith@ac.com
1123	bliu@ac.com
1023	ssidhuk@ac.com

- Fine: join returns the original relation
- Unnecessary: no redundancy is removed, and now *EID* is stored twice->

Bad decomposition

EID	PID	Hours
1234	10	10
1123	9	40
1234	9	30
1023	10	40

EID	PID
1234	10
1123	9
1234	9
1023	10

EID	Hours
1234	10
1123	40
1234	30
1023	40

- Association between *PID* and *hours* is lost
- Join returns more rows than the original relation

Lossless join decomposition

- Decompose relation R into relations S and T
  - $attrs(R) = attrs(S) \cup attrs(T)$
  - $S = \pi_{attrs(S)} (R)$
  - $T = \pi_{attrs(T)} (R)$
- The decomposition is a lossless join decomposition if, given known *constraints* such as FD's, we can guarantee that $R = S * T$
- Any decomposition gives $R \subseteq S \bowtie T$ (why?)
  - A *lossy* decomposition is one with $R \subset S \bowtie T$

Loss? But I got more rows->

- "Loss" refers not to the loss of tuples, but to the loss of information
  - Or, the ability to distinguish different original tuples

EID	PID	Hours
1234	10	10
1123	9	40
1234	9	30
1023	10	40

EID	PID
1234	10
1123	9
1234	9
1023	10

EID	Hours
1234	10
1123	40
1234	30
1023	40

Questions about decomposition

- When to decompose
- How to come up with a correct decomposition (i.e., lossless join decomposition)

Non-key FD's

- Consider a non-trivial FD *X* -> *Y* where *X* is not a super key
  - Since *X* is not a super key, there are some attributes (say *Z*) that are not functionally determined by *X*

X	Y	Z
a	b	c
a	b	d

That *b* is always associated with *a* is recorded by multiple rows: redundancy, update anomaly, deletion anomaly

Dealing with Nonkey Dependency: BCNF

- A relation *R* is in Boyce-Codd Normal Form if
  - For every non-trivial FD *X* -> *Y* in *R*, *X* is a super key
  - That is, all FDs follow from "key -> other attributes"

- When to decompose
  - As long as some relation is not in BCNF
- How to come up with a correct decomposition
  - Always decompose on a BCNF violation (details next)
  - ☞ Then it is guaranteed to be a lossless join decomposition->

BCNF decomposition algorithm

- Find a BCNF violation
  - That is, a non-trivial FD X -> Y in R where X is not a super key of R
- Decompose R into R_1 and R_2, where
  - R_1 has attributes $X \cup Y$
  - R_2 has attributes $X \cup Z$, where Z contains all attributes of R that are in neither X nor Y (i.e. $Z = attr(R) - X - Y$)
- Repeat until all relations are in BCNF

BCNF decomposition example

WorkOn (*EID*, *Ename*, *email*, *PID*, *hours*)

BCNF violation: *EID* -> *Ename*, *email*

Student (*EID*, *Ename*, *email*)

BCNF

Grade (*EID*, *PID*, *hours*)

BCNF

Another example

WorkOn (*EID*, *Ename*, *email*, *PID*, *hours*)

BCNF violation: *email* -> *EID*

StudentID (*email*, *EID*)

BCNF

StudentGrade' (*email*, *Ename*, *PID*, *hours*)

BCNF violation: *email* -> *Ename*

StudentName (*email*, *Ename*)

BCNF

Grade (*email*, *PID*, *hours*)

BCNF

Exercise

- *Property(Property_id#, County_name, Lot#, Area, Price, Tax_rate)*
  - Property_id#-> *County_name, Lot#, Area, Price, Tax_rate*
  - *County_name, Lot#* -> Property_id#, *Area, Price, Tax_rate*
  - *County_name* -> *Tax_rate*
  - *area* -> *Price*

Exercise

Property(Property_id#, County_name, Lot#, Area, Price, Tax_rate)

BCNF violation: County_name -> Tax_rate

LOTS1 (County_name, Tax_rate)

BCNF

LOTS2 (Property_id#, County_name, Lot#, Area, Price)

BCNF violation: Area -> Price

LOTS2A (Area, Price)

BCNF

LOTS2B (Property_id#, County_name, Lot#, Area)

BCNF

Why is BCNF decomposition lossless

Given non-trivial $X \rightarrow Y$ in R where X is not a super key of R, need to prove:

- Anything we project always comes back in the join:
  $R \subseteq \pi_{XY}(R) \bowtie \pi_{XZ}(R)$
  - Sure; and it doesn't depend on the FD
- Anything that comes back in the join must be in the original relation:
  $R \supseteq \pi_{XY}(R) \bowtie \pi_{XZ}(R)$
  - Proof makes use of the fact that $X \rightarrow Y$

Recap

- Functional dependencies: a generalization of the key concept
- Partial dependencies: a source of redundancy
  - Use 2nd Normal form to remove partial dependency
- Non-key functional dependencies: a source of redundancy
- BCNF decomposition: a method for removing ALL functional dependency related redundancies
  - Plus, BCNF decomposition is a lossless join decomposition

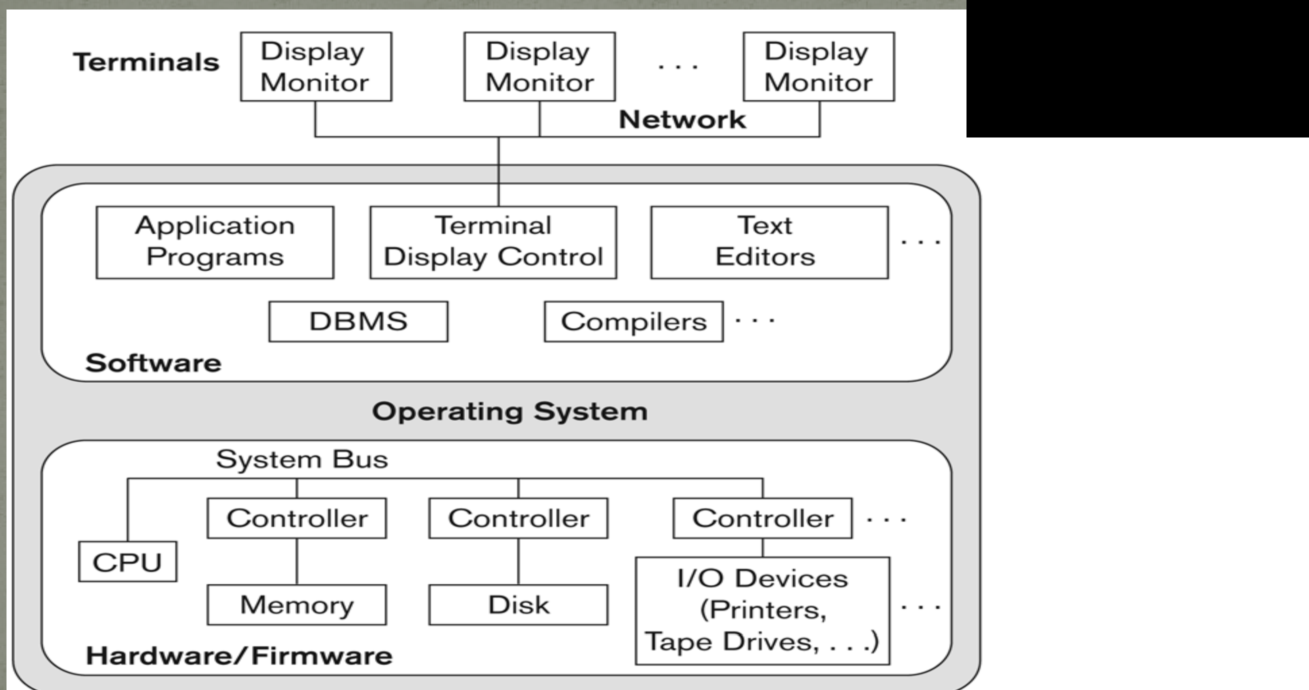
Today's Topic

- Database Architecture
- Database programming

Centralized Architectures

- Centralized DBMS: combines everything into single system including- DBMS software, hardware, application programs and user interface processing software.

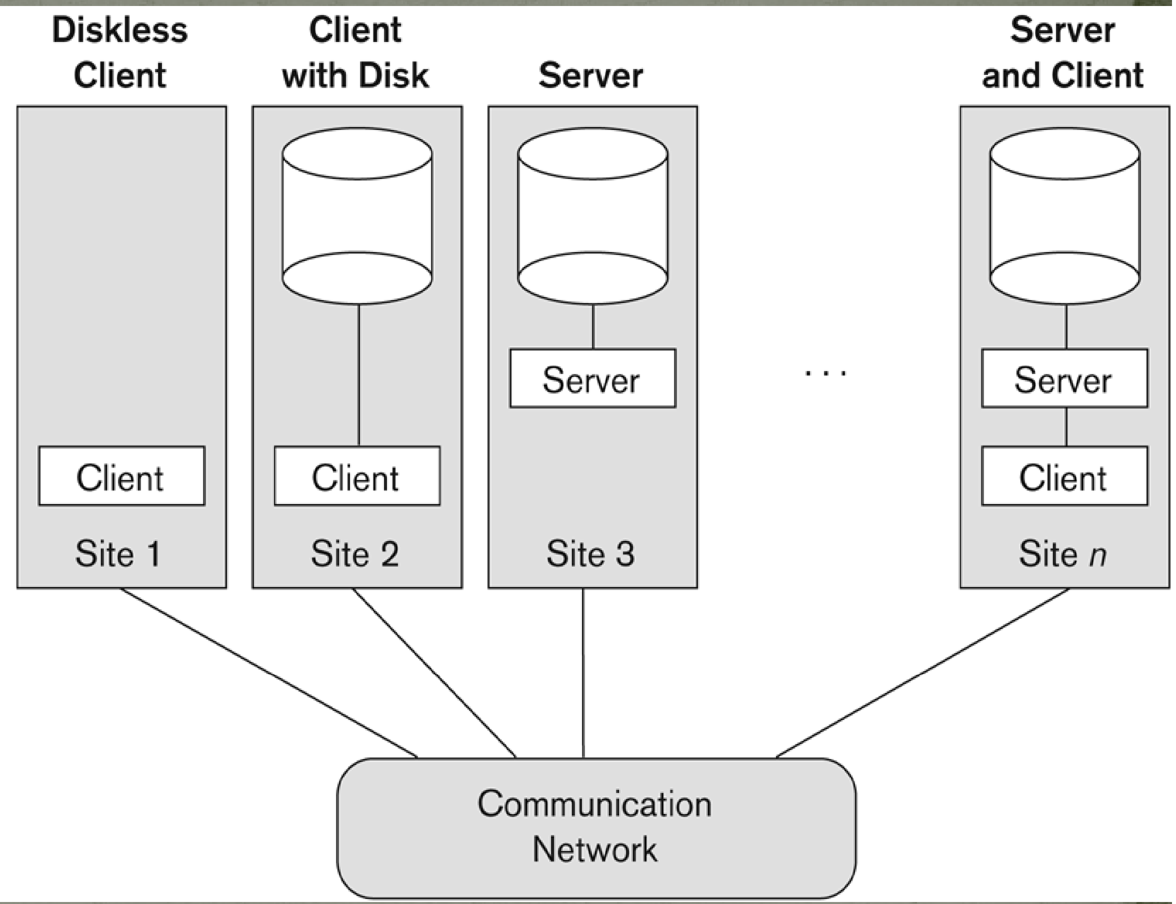
[redacted]



Two Tier Client-Server Architectures

- Server: provides query and transac services to client
- Client: provide appropriate interf server.
  - Run User Inte Programs and Application Pr
  - Connect to se network.

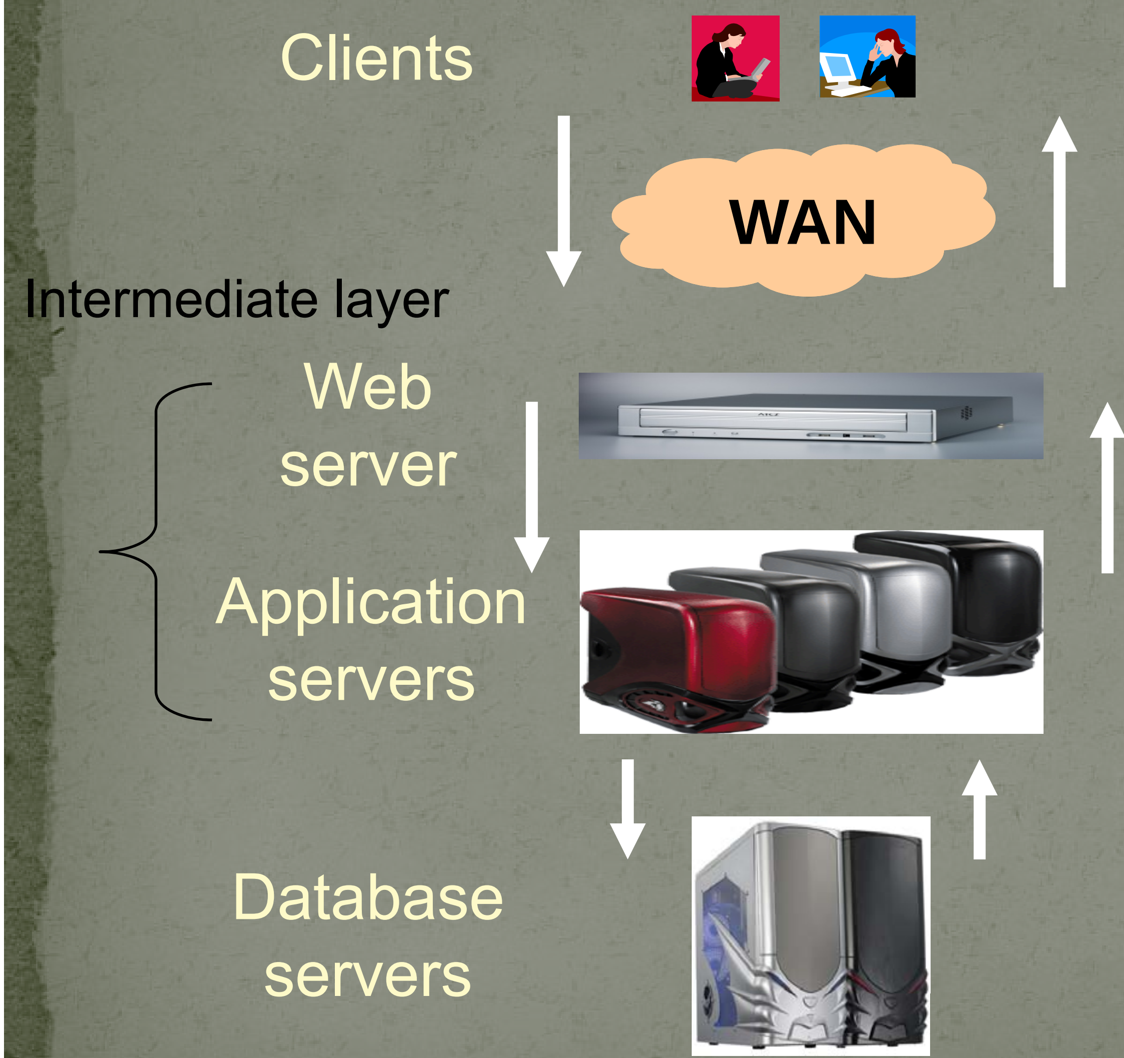
Figure 2.6
Physical two-tier client/server architecture.



Client-Server Interface

- The interface between a server and a client is commonly specified by ODBC (Open Database Connectivity)
  - Provides an Application program interface (API)
  - Allow client side programs to call the DBMS.

Three (n) Tier Client-Server Architecture



- The intermediate layer is called Application Server or Web Server, or both:
- Stores the web connectivity software and business logic for applications
- Acts like a conduit for sending partially processed data between the database server and the client.
- Additional Features
  - Security: encrypt the data at the server and client before transmission

Database Programming: Overview

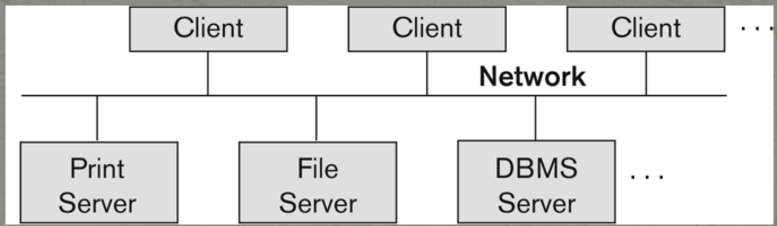
- Pros and cons of SQL
  - Very high-level, possible to optimize
  - Specifically designed for databases and is called *data sublanguage*
  - Not intended for general-purpose computation, which is usually done by a *host language*
- Solutions
  - Augment SQL with constructs from general-purpose programming languages (SQL/PSM)
  - Use SQL together with general-purpose programming languages
    - Database APIs, embedded SQL, JDBC, etc.

Clarification of Terms

- John has a mySQL database server installed in his laptop. He wrote a perl script to connect to the local mySQL database, retrieve data, and print out reports about his house innovation plan.
  - Client-server model
  - Use APIs provided by mySQL to access the database
  - Perl supports mySQL API

Clarification of Terms (cont.)

- John went to his office. He has a JAVA program, which connects to a SqlServer database in his company's intranet. He use the program to retrieve data and print out reports for his business partner.
  - Client-server model
  - Use APIs provided by SqlServer to access the database
  - Java supports SqlServer API using JDBC



Clarification of Terms (cont.)

- After job, John went to youtube.com, searched for a video of Thomas train for his children, and downloaded one
  - Client-mediate level-sever model
  - “SQL experience a plus” from a job ad linked from youtube’s web site.





Impedance mismatch and a solution

- SQL operates on a set of records at a time
- Typical low-level general-purpose programming languages operates on one record at a time

☞ Solution: cursor

  - Open (a result table): position the cursor before the first row
  - Get next: move the cursor to the next row and return that row; raise a flag if there is no such row
  - Close: clean up and release DBMS resources

  ☞ Found in virtually every database language/API

    - With slightly different syntaxes

A Typical Flow of Interactions

- A client (user interface, web server, application server) opens a connection to a database server
- A client interact with the database server to perform query, update, or other operations.
- A client terminate the connection

Interfacing SQL with another language

- API approach
  - SQL commands are sent to the DBMS at runtime
  - Examples: JDBC, ODBC (for C/C++/VB), Perl DBI
  - These API's are all based on the SQL/CLI (Call-Level Interface) standard
- Embedded SQL approach
  - SQL commands are embedded in application code
  - A precompiler checks these commands at compile-time and converts them into DBMS-specific API calls
  - Examples: embedded SQL for C/C++, SQLJ (for Java)

Example API: JDBC

- JDBC (Java DataBase Connectivity) is an API that allows a Java program to access databases

```
// Use the JDBC package:
import java.sql.*;
…
public class … {
    …
    static {
        // Load the JDBC driver:
        try {
Class.forName("oracle.jdbc.driver.OracleDriver");
        } catch (ClassNotFoundException e) {
            …
        }
    }
    …
}
```

Connections

```
// Connection URL is a DBMS-specific string:
String url =
    "jdbc:oracle:thin:@oracle.cs.uky.edu:1521:orcl";

// Making a connection:
 conn
=DriverManager.getConnection(url,username,password)
…
// Closing a connection:
con.close();
```

For clarity we are ignoring exception handling for now

Statements

```
//  Create an object for sending SQL statements:
Statement stmt = con.createStatement();

//  Execute a query and get its results:
ResultSet rs =
    stmt.executeQuery(”SELECT name, passwd FROM
regiusers”);

//  Work on the results:
…
//  Execute a modification (returns the number of rows affected):
int rowsUpdated =
    stmt.executeUpdate
    (”UPDATE regiusers SET passwd = ’1234’ WHERE name =
‘sjohn’ ”);
//  Close the statement:
stmt.close();
```

Query results

```
//  Execute a query and get its results:
ResultSet rs =
    stmt.executeQuery(”SELECT name, passwd FROM regiusers”);

//  Loop through all result rows:
while (rs.next()) {
    //  Get column values:
    String name = rs.string(1);
    String passwd = rs.getString(2);
    //  Work on sid and name:
    …
}

//  Close the ResultSet:
rs.close();
```

Other `ResultSet` features

- Move the cursor (pointing to the current row) backwards and forwards, or position it anywhere within the `ResultSet`
- Update/delete the database row corresponding to the current result row
  - Analogous to the view update problem
- Insert a row into the database
  - Analogous to the view update problem

Prepared statements: motivation

```
Statement stmt = con.createStatement();
for (int age=0; age<100; age+=10) {
    ResultSet rs = stmt.executeQuery
        ("SELECT AVG(GPA) FROM Student" +
         " WHERE age >= " + age + " AND age < " + (age+10));
    // Work on the results:
    …
}
```

- Every time an SQL string is sent to the DBMS, the DBMS must perform parsing, semantic analysis, optimization, compilation, and then finally execution
- These costs are incurred 10 times in the above example
- A typical application issues many queries with a small number of patterns (with different parameter values)

Transaction processing

- Set isolation level for the current transaction
  - `con.setTransactionIsolationLevel(l);`
  - Where *l* is one of `TRANSACTION_SERIALIZABLE` (default), `TRANSACTION_REPEATABLE_READ`, `TRANSACTION_READ_COMITTED`, and `TRANSACTION_READ_UNCOMMITTED`
- Set the transaction to be read-only or read/write (default)
  - `con.setReadOnly(true|false);`
- Turn on/off `AUTOCOMMIT` (commits every single statement)
  - `con.setAutoCommit(true|false);`
- Commit/rollback the current transaction (when `AUTOCOMMIT` is off)
  - `con.commit();`
  - `con.rollback();`