ICS 421 Spring 2010
SQL & Application Programming

Asst. Prof. Lipyeow Lim

Information & Computer Science Department

University of Hawaii at Manoa

Nested Queries

Q1 : Find the names of sailors who have reserved boat 103

```
SELECT S.sname
FROM   Sailors S, Reserves R
WHERE  S.sid=R.sid AND bid=103
```

```
SELECT S.sname
FROM   Sailors S
WHERE  S.sid IN ( SELECT R.sid
                  FROM Reserves R
                  WHERE R.bid=103 )
```

- A *nested query* is a query that has another query, called a *subquery,* embedded within it.
- Subqueries can appear in WHERE, FROM, HAVING clauses

Conceptual Evaluation Strategy for Nested Queries

1. Compute the cross-product of *relation-list.*
   - If there is a subquery, recursively (re-)compute the subquery using this conceptual evaluation strategy
   - Compute the cross-product over the results of the subquery.
2. Discard resulting tuples if they fail *qualifications.*
   - If there is a subquery, recursively (re-)compute the subquery using this conceptual evaluation strategy
   - Evaluate the qualification condition that depends on the subquery
3. Delete attributes that are not in *target-list.*
4. If DISTINCT is specified, eliminate duplicate rows.

Correlated Nested Queries

Q1: Find the names of sailors who’ve reserved boat #103

```
SELECT S.sname
FROM    Sailors S
WHERE  EXISTS ( SELECT *
                FROM Reserves R
                WHERE R.bid = 103 AND R.sid=S.sid
```

- EXISTS is another set comparison operator, like *IN*.
- If UNIQUE is used, and * is replaced by R.bid, finds sailors with at most one reservation for boat #103. (UNIQUE checks for duplicate tuples; * denotes all attributes. Why do we have to replace * by R.bid?)
- Illustrates why, in general, subquery must be re-computed for each Sailors tuple.

Aggregate Operators

- SQL supports 5 aggregation operators on a column, say A,
  1. COUNT (*), COUNT ([DISTINCT] A)
  2. SUM ([DISTINCT] A)
  3. AVG ([DISTINCT] A)
  4. MAX (A)
  5. MIN (A)

Q27: Find the name and age of the oldest sailor

```
SELECT S.sname, MAX (S.age)
FROM    Sailors S
```

```
SELECT S.sname, S.age
FROM    Sailors S
WHERE S.age = ( SELECT MAX(S2.age)
                FROM Sailors S2 )
```

- If there is an aggregation operator in the SELECT clause, then it can only have aggregation operators unless the query has a GROUP BY clause -- first query is illegal.

Queries with GROUP BY and HAVING

```
SELECT      [DISTINCT]  target-list
FROM        relation-list
WHERE       qualification
GROUP BY    grouping-list
HAVING      group-qualification
```

- The *target-list* contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (*S.age*)).
  - The list of attribute names in (i) must be a subset of *grouping-list*.
  - Intuitively, each answer tuple corresponds to a *group*, and these attributes must have a single value per group.
  - A *group* is a set of tuples that have the same value for all attributes in *grouping-list*.

Conceptual Evaluation Strategy with GROUP BY and HAVING

- [Same as before] The cross-product of *relation-list* is computed, tuples that fail *qualification* are discarded, `*unnecessary*' fields are deleted
- The remaining tuples are partitioned into groups by the value of attributes in *grouping-list*.
- The *group-qualification* is then applied to eliminate some groups. Expressions in *group-qualification* must have a *single value per group*!
  - In effect, an attribute in *group-qualification* that is not an argument of an aggregate op also appears in *grouping-list*. (SQL does not exploit primary key semantics here!)
- Aggregations in *target-list* are computed for each group
- One answer tuple is generated per qualifying group

Q32: Find age of the youngest sailor with age >= 18, for each rating with at least 2 such sailors

```
SELECT  S.rating,
        MIN(S.age) AS minage
FROM  Sailors S
WHERE  S.age >= 18
GROUP BY  S.rating
HAVING  COUNT (*) > 1
```

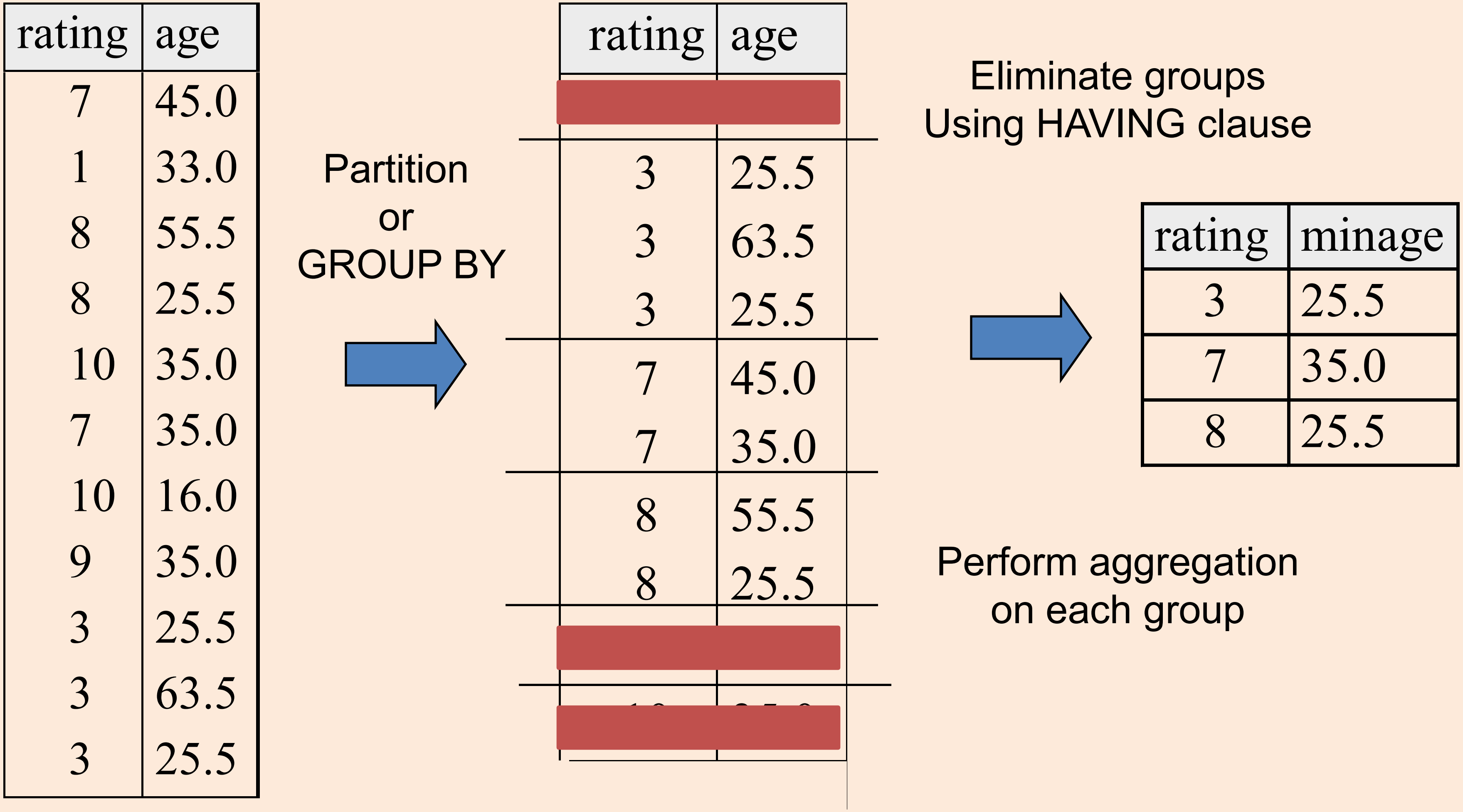
Sailors instance:

sid	sname	rating	age
22	dustin	7	45.0
29	brutus	1	33.0
31	lubber	8	55.5
32	andy	8	25.5
58	rusty	10	35.0
64	horatio	7	35.0
71	zorba	10	16.0
74	horatio	9	35.0
85	art	3	25.5
95	bob	3	63.5
96	frodo	3	25.5

Answer relation:

rating	minage
3	25.5
7	35.0
8	25.5

Conceptual Evaluation for Q32



rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5

Partition
or
GROUP BY

rating	age
[redacted]	[redacted]
3	25.5
3	63.5
3	25.5
7	45.0
7	35.0
8	55.5
8	25.5
[redacted]	[redacted]
[redacted]	[redacted]

Eliminate groups
Using HAVING clause

rating	minage
3	25.5
7	35.0
8	25.5

Perform aggregation
on each group

EVERY and ANY in HAVING clauses

```
SELECT  S.rating, MIN(S.age) AS minage
FROM  Sailors S
WHERE  S.age >= 18
GROUP BY  S.rating
HAVING  COUNT (*) > 1 AND EVERY ( S.age <=60 )
```

- EVERY: every row in the group must satisfy the attached condition
- ANY: at least one row in the group need to satisfy the condition

Conceptual Evaluation with EVERY

HAVING COUNT (*) > 1 AND EVERY (S.age <=60)

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5

Partition
or
GROUP BY

rating	age
[redacted]	[redacted]
[redacted]	[redacted]
7	45.0
7	35.0
8	55.5
8	25.5
[redacted]	[redacted]
[redacted]	[redacted]

Eliminate groups
Using HAVING clause

rating	minage
7	35.0
8	25.5

Perform aggregation
on each group

What is the result of changing EVERY to ANY?

Find age of the youngest sailor with age 18, for each rating with at least 2 sailors between 18 and 60

```
SELECT  S.rating,
        MIN (S.age) AS minage
FROM  Sailors S
WHERE  S.age >= 18 AND S.age <= 60
GROUP BY  S.rating
HAVING  COUNT (*) > 1
```

Sailors instance:

sid	sname	rating	age
22	dustin	7	45.0
29	brutus	1	33.0
31	lubber	8	55.5
32	andy	8	25.5
58	rusty	10	35.0
64	horatio	7	35.0
71	zorba	10	16.0
74	horatio	9	35.0
85	art	3	25.5
95	bob	3	63.5
96	frodo	3	25.5

Answer relation:

rating	minage
3	25.5
7	35.0
8	25.5

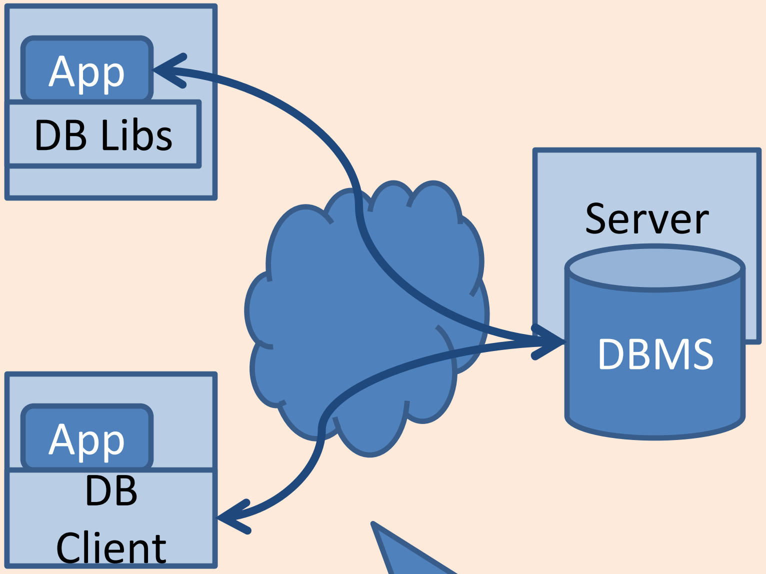
SQL & Other Programming Languages

Two extremes of the integration spectrum:

- Highly integrated eg. Microsoft linq
  - Compiler checking of database operations
- Loosely integrated eg. ODBC & JDBC
  - Provides a way to call SQL from host language
  - Host language compiler doesn't understand database operations.
- Requirements:
  - Perform DB operations from host language
  - DB operations need to access variables in host language

Remote Client Access

- Applications run on a machine that is separate from the DB server
- DBMS “thin” client
  - Libraries to link your app to
  - App needs to know how to talk to DBMS server via network
- DBMS “full” client layer
  - Need to pre-configure the thick client layer to talk to DBMS server
  - Your app talks to a DBMS client layer as if it is talking to the server



What information is needed for 2 machines to talk over a network ?

Configuring DBMS Client Layer

- Tell the client where to find the server

```
db2 CATALOG TCPIP NODE mydbsrv
   REMOTE 123.3.4.12 SERVER 50001
```

Give a name for this node

Specify the IP address/hostname and the port number of the DB server machine

- Tell the client where to find the server

```
db2 CATALOG DATABASE bookdb AS
   mybookdb AT NODE mydbsrv
```

Specify the name of the database on the server

Give a local alias for the database

Specify the name of the node that is associated with this database

Static vs Dynamic SQL

- Static SQL refers to SQL queries that are completely specified at compile time. Eg.

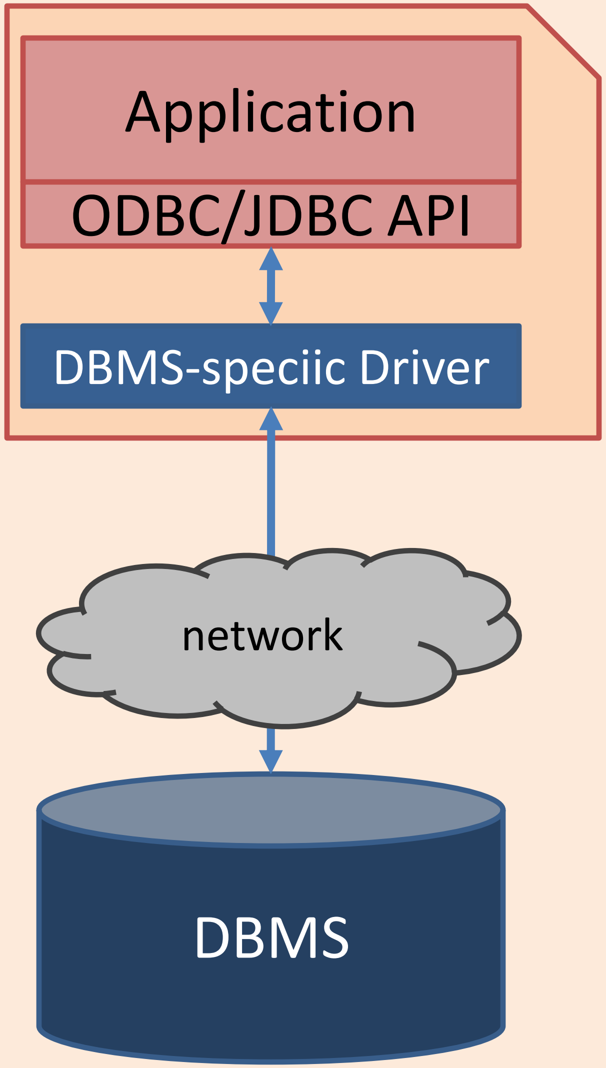
```
// Declare A Static Cursor
EXEC SQL DECLARE C1 CURSOR FOR
SELECT EMPNO, LASTNAME,
    DOUBLE(SALARY)
FROM EMPLOYEE
WHERE JOB = 'DESIGNER';
```

- Dynamic SQL refers to SQL queries that are note completely specified at compile time. Eg.

```
strcpy(SQLStmt, “SELECT * FROM
    EMPLOYEE WHERE JOB=");
strcat(SQLStmt,  argv[1]);
EXEC SQL PREPARE SQL_STMT FROM
    :SQLStmt;
EXEC SQL EXECUTE SQL_STMT;
```

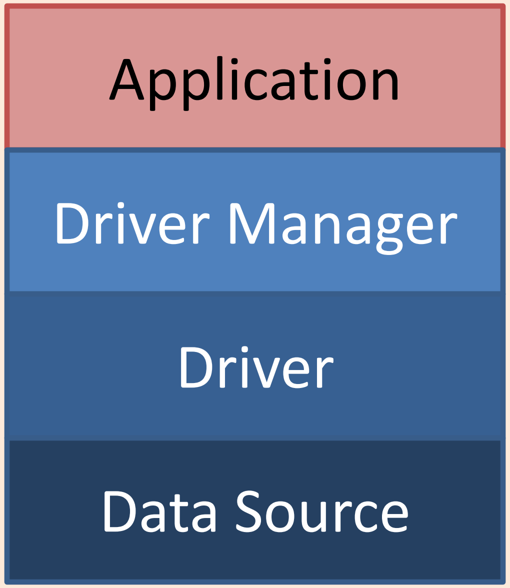
Alternative to Embedded SQL

- What if we want to compile an application without the need for a DBMS-specific pre-compiler ?
- Use a library of database calls
  - Standardized (non-DBMS-specific) API
  - Pass SQL-strings from host language and presents result sets in a language friendly way
  - Eg. ODBC for C/C++ and JDBC for Java
  - DBMS-neutral
    - A driver traps the calls and translates them into DBMS-specific code



ODBC/JDBC Architecture

- Application
  - Initiates connections
  - Submits SQL statements
  - Terminates connections
- Driver Manager
  - Loads the right JDBC driver
- Driver
  - Connects to the data source,
  - Transmit requests,
  - Returns results and error codes
- Data Source
  - DBMS



4 Types of Drivers

- Type I: Bridge
  - Translate SQL commands to non-native API
  - eg. JDBC-ODBC bridge. JDBC is translated to ODBC to access an ODBC compliant data source.
- Type II: Direct Translation to native API via non-Java driver
  - Translates SQL to native API of data source.
  - Needs DBMS-specific library on each client.
- Type III: Network bridge
  - SQL stmts sent a middleware server that talks to the data source. Hence small JDBC driver at each client
- Type IV: Direct Translation to native API via Java driver
  - Converts JDBC calls to network protocol used by DBMS.
  - Needs DBMS-specific Java driver at each client.

High Level Steps

1. Load the ODBC/JDBC driver
2. Connect to the data source
3. [optional] Prepare the SQL statements
4. Execute the SQL statements
5. Iterate over the resultset
6. Close the connection

Prepare Statement or Not ?

```
String sql="SELECT * FROM books WHERE price < ?";
PreparedStatement pstmt = conn.prepareStatement(sql);
Pstmt.setFloat(1, usermaxprice);
Pstmt.executeUpdate();
```

- Executing without preparing statement
  - After DBMS receives SQL statement,
    - The SQL is compiled,
    - An execution plan is chosen by the optimizer,
    - The execution plan is evaluated by the DBMS engine
    - The results are returned
- conn.prepareStatement
  - Compiles and picks an execution plan
- pstmt.executeUpdate
  - Evaluates the execution plan with the parameters and gets the results

cf. Static vs Dynamic SQL

ResultSet

```
ResultSet rs = stmt.executeQuery(sqlstr);
while( rs.next() ){
		col1val = rs.getString(1); …
}
```

- Iterate over the results of a SQL statement -- cf. cursor
- Note that types of column values do not need to be known at compile time

SQL Type	Java Class	accessor
BIT	Boolean	getBoolean
CHAR, VARCHAR	String	getString
DOUBLE, FLOAT	Double	getDouble
INTEGER	Integer	getInt
REAL	Double	getFloat
DATE	Java.sql.Date	getDate
TIME	Java.sql.Time	getTime
TIMESTAMP	Java.sql.TimeStamp	getTimestamp

RowSet

- When inserting lots of data, calling an execute statement for each row can be inefficient
  - A message is sent for each execute
- Many APIs provide a rowset implementation
  - A set of rows is maintained in-memory on the client
  - A single execute will then insert the set of rows in a single message
- Pros: high performance
- Cons: data can be lost if client crashes.
- Analogous rowset for reads (ie. ResultSet) also available