Logistics

• Get a CD while you can
• DeZign for Databases

The SQL Query Language

• Developed by IBM (system R) in the 1970s
• Need for a standard since it is used by many vendors
• Standards:
  • SQL-86
  • SQL-89 (minor revision)
  • SQL-92 (major revision)
  • SQL-99 (major extensions, current standard)
Example Instances

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

- We will use these instances of the Sailors and Reserves relations in our examples.
- If the key for the Reserves relation contained only the attributes sid and bid, how would the semantics differ?

Basic SQL Query

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
```

- **relation-list:** A list of relation names (possibly with a range-variable after each name).
- **target-list:** A list of attributes of relations in relation-list
- **qualification:**
  - Comparisons: Attr op const or Attr1 op Attr2, where op is one of the following: 
    - $\leq$, $\geq$, $\neq$
    - combined using AND, OR and NOT.
  - **DISTINCT** is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are not eliminated!

Conceptual Evaluation Strategy

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of relation-list.
  - Discard resulting tuples if they fail qualifications.
  - Delete attributes that are not in target-list.
  - If **DISTINCT** is specified, eliminate duplicate rows.
  - This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute the same answers.
### Example of Conceptual Evaluation

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

<table>
<thead>
<tr>
<th>(sid)</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>(sid)</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

### A Note on Range Variables

- Really needed only if the same relation appears twice in the FROM clause. The previous query can also be written as:

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

OR

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

It is good style, however, to use range variables always!

### Find sailors who have reserved at least one boat

```sql
SELECT  S.sid
FROM     Sailors S, Reserves R
WHERE S.sid=R.sid
```

- Would adding **DISTINCT** to this query make a difference?
- What is the effect of replacing S.sid by S.sname in the SELECT clause? Would adding **DISTINCT** to this variant of the query make a difference?
Expressions and Strings

SELECT S.age, age1=S.age-5, 2*S.age AS age2
FROM Sailors S
WHERE S.sname LIKE ‘B_%B’

- Illustrates use of arithmetic expressions and string pattern matching: Find triples (of ages of sailors and two fields defined by expressions) for sailors whose names begin and end with B and contain at least three characters.
- AS and = are two ways to name fields in result.
- LIKE is used for string matching. ‘_’ stands for any one character and ‘%’ stands for 0 or more arbitrary characters.

Find sid’s of sailors who’ve reserved a red or a green boat

- UNION: Can be used to compute the union of any two union-compatible sets of tuples (which are themselves the result of SQL queries).
- If we replace OR by AND in the first version, what do we get?
- Also available: EXCEPT (What do we get if we replace UNION by EXCEPT?)

Find sid’s of sailors who’ve reserved a red and a green boat

- INTERSECT: Can be used to compute the intersection of any two union-compatible sets of tuples.
- Included in the SQL/92 standard, but some systems don’t support it.
- Contrast symmetry of the UNION and INTERSECT queries with how much the other versions differ.
In-Class Exercise

Suppliers(sid: integer, sname:string, address:string)
Parts(pid: integer, pname:string, color:string)
Catalog(sid: integer, pid: integer, cost: real)

- Find the *pnames* of parts for which there is some supplier.
- Find the *sids* of suppliers who supply a red part and a green part.
- Find the *sids* of suppliers who supply a red part or a green part.

Back to SQL: Nested Queries

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

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

- A very powerful feature of SQL: a WHERE clause can itself contain an SQL query! (Actually, so can FROM and HAVING clauses.)
- To find sailors who’ve not reserved #103, use NOT IN.
- To understand semantics of nested queries, think of a nested loops evaluation: For each Sailors tuple, check the qualification by computing the subquery.

Nested Queries with Correlation

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

```sql
SELECT S.sname
FROM Sailors S
WHERE EXISTS (SELECT *
               FROM Reserves R
               WHERE R.bid=103 AND S.sid=R.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.
More on Set-Comparison Operators

- We’ve already seen IN, EXISTS and UNIQUE. Can also use NOT IN, NOT EXISTS and NOT UNIQUE.
- Also available: \( \text{op ANY, op ALL, op IN} \)
- Find sailors whose rating is greater than that of some sailor called Horatio:
  
  ```sql
  SELECT *
  FROM Sailors S
  WHERE S.rating > ANY (SELECT S2.rating
  FROM Sailors S2
  WHERE S2.sname='Horatio')
  ```

Rewriting INTERSECT Queries Using IN

- Similarly, EXCEPT queries re-written using NOT IN.
- To find names (not sid’s) of Sailors who’ve reserved both red and green boats, just replace \( S.sid \) by \( S.sname \) in SELECT clause. (What about INTERSECT query?)

Division

- Not supported as a primitive operator, but useful for expressing queries like:
  - Find sailors who have reserved all boats.
- Let \( A \) have 2 fields, \( x \) and \( y \); \( B \) have only field \( y \):
  - \( A/B = \{ (x) \mid \exists (x,y) \in A \land (y) \in B \} \)
  - i.e., \( A/B \) contains all \( x \) tuples (sailors) such that for every \( y \) tuple (boat) in \( B \), there is an \( xy \) tuple in \( A \).
- Or: If the set of \( y \) values (boats) associated with an \( x \) value (sailor) in \( A \) contains all \( y \) values in \( B \), the \( x \) value is in \( A/B \).
- In general, \( x \) and \( y \) can be any lists of fields; \( y \) is the list of fields in \( B \); and \( x \land y \) is the list of fields of \( A \).
Examples of Division A/B

<table>
<thead>
<tr>
<th>sno</th>
<th>pno</th>
<th>pno</th>
<th>pno</th>
<th>pno</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>p1</td>
<td>p2</td>
<td>p1</td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td>p2</td>
<td>p2</td>
<td>p2</td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td>p3</td>
<td>p4</td>
<td>p4</td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td>p4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>p1</td>
<td>sno</td>
<td>s1</td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>p2</td>
<td>s1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>p2</td>
<td>s1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s4</td>
<td>p2</td>
<td>s1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s4</td>
<td>p4</td>
<td>s1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A  A/B1  A/B2  A/B3

Expressing A/B Using Basic Operators

- Division is not essential op; just a useful shorthand.
  - (Also true of joins, but joins are so common that systems implement joins specially.)
- **Idea** For A/B, compute all x values that are not ‘disqualified’ by some y value in B.
  - x value is disqualified if by attaching y value from B, we obtain an xy tuple that is not in A.

Disqualified x values: \( \pi_x((\pi_x(A) \times B) - A) \)

A/B: \( \pi_x(A) \) – all disqualified tuples

Find the names of sailors who’ve reserved all boats

- Uses division; schemas of the input relations to / must be carefully chosen:
  - \( \rho(Tempsids, (\pi_{sid,bid} Reserves) / (\pi_{bid} Boats)) \)
  - \( \pi_{sname}(Tempsids \bowtie Sailors) \)
- To find sailors who’ve reserved all ‘Interlake’ boats:
  - \( \ldots / \pi_{bid}(\sigma_{bname = 'Interlake'} Boats) \)
### Division in SQL

Find sailors who’ve reserved all boats:

- Let’s do it the hard way, without EXCEPT:

\[
\begin{align*}
(1) & \quad \text{SELECT } S\text{.sname} \\
\text{FROM} & \quad \text{Sailors } S \\
\text{WHERE} & \quad \text{NOT EXISTS} \\
\left( & \quad \text{SELECT } B\text{.bid} \\
\text{FROM} & \quad \text{Boats } B \\
\text{EXCEPT} & \quad \text{SELECT } R\text{.bid} \\
\text{FROM} & \quad \text{Reserves } R \\
\text{WHERE} & \quad R\text{.sid}=S\text{.sid})
\end{align*}
\]

\[
\begin{align*}
(2) & \quad \text{SELECT } S\text{.sname} \\
\text{FROM} & \quad \text{Sailors } S \\
\text{WHERE} & \quad \text{NOT EXISTS} \\
\left( & \quad \text{SELECT } B\text{.bid} \\
\text{FROM} & \quad \text{Boats } B \\
\text{WHERE} & \quad \text{NOT EXISTS} \\
\left( & \quad \text{SELECT } R\text{.bid} \\
\text{FROM} & \quad \text{Reserves } R \\
\text{WHERE} & \quad R\text{.bid}=B\text{.bid} \\
\text{AND} & \quad R\text{.sid}=S\text{.sid})
\right)
\end{align*}
\]

Sailors S such that ...
there is no boat B without ...
a Reserves tuple showing S reserved B

### In-Class Exercise

**Suppliers**
- sid: integer
  - sname: string
  - address: string

**Parts**
- pid: integer
  - pname: string
  - color: string

**Catalog**
- sid: integer
  - pid: integer
  - cost: real

- Find the *snames* of suppliers who supply every part.
- Find the *snames* of suppliers who supply every red part.
- Find the *pnames* of parts supplied by Acme Widget Suppliers and no one else.

### Aggregate Operators

**Significant extension of relational algebra.**

\[
\begin{align*}
\text{SELECT} & \quad \text{COUNT}(*) \\
\text{FROM} & \quad \text{Sailors } S
\end{align*}
\]

\[
\begin{align*}
\text{SELECT} & \quad \text{S\text{.sname}} \\
\text{FROM} & \quad \text{Sailors } S \\
\text{WHERE} & \quad \text{S\text{.rating}=10}
\end{align*}
\]

\[
\begin{align*}
\text{SELECT} & \quad \text{COUNT(DISTINCT} S\text{.rating)} \\
\text{FROM} & \quad \text{Sailors } S \\
\text{WHERE} & \quad \text{S\text{.sname} = ‘Bob’}
\end{align*}
\]

\[
\begin{align*}
\text{SELECT} & \quad \text{AVG(DISTINCT} S\text{.age)} \\
\text{FROM} & \quad \text{Sailors } S \\
\text{WHERE} & \quad \text{S\text{.rating}=10}
\end{align*}
\]
Find name and age of the oldest sailor(s)

- The first query is illegal! (We’ll look into the reason a bit later, when we discuss GROUP BY.)
- The third query is equivalent to the second query, and is allowed in the SQL/92 standard, but is not supported in some systems.

GROUP BY and HAVING

- So far, we’ve applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several groups of tuples.
- Consider: Find the age of the youngest sailor for each rating level.
  - In general, we don’t know how many rating levels exist, and what the rating values for these levels are!
  - Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (!):

  For $i = 1, 2, \ldots, 10$:
  
  ```sql
  SELECT MIN(S.age)
  FROM Sailors S
  WHERE S.rating = i
  ```

Queries With GROUP BY and HAVING

- The target-list contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (S.age)).
- The attribute list (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

- The cross-product of relation-list is computed, tuples that fail qualification are discarded, 'unnecessary' fields are deleted, and 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!)
- One answer tuple is generated per qualifying group.

Find the age of the youngest sailor with age ≥ 18, for each rating with at least 2 such sailors

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

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>71</td>
<td>zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

* Only S.rating and S.age are mentioned in the SELECT, GROUP BY or HAVING clauses; other attributes 'unnecessary'.
* 2nd column of result is unnamed. (Use AS to name it.)

Answer relation

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

For each red boat, find the number of reservations for this boat

```
SELECT B.bid, COUNT(*) AS scount
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid AND B.color='red'
GROUP BY B.bid
```

- Grouping over a join of three relations.
- What do we get if we remove B.color='red' from the WHERE clause and add a HAVING clause with this condition?
- What if we drop Sailors and the condition involving S.sid?
Find the age of the youngest sailor with age > 18, for each rating with at least 2 sailors (of any age)

```sql
SELECT S.rating, MIN(S.age)
FROM Sailors S
WHERE S.age > 18
GROUP BY S.rating
HAVING 1 < (SELECT COUNT(*)
            FROM Sailors S2
            WHERE S.rating=S2.rating)
```

- Shows HAVING clause can also contain a subquery.
- Compare this with the query where we considered only ratings with 2 sailors over 18!
- What if HAVING clause is replaced by:
  - HAVING COUNT(*) >1

Find those ratings for which the average age is the minimum over all ratings

- Aggregate operations cannot be nested! **WRONG:**

```sql
SELECT S.rating
FROM Sailors S
WHERE S.age = (SELECT MIN(AVG(S2.age))
              FROM Sailors S2)
```

- Correct solution (in SQL/92):  

```sql
SELECT Temp.rating, Temp.avgage
FROM (SELECT S.rating, AVG(S.age) AS avgage
      FROM Sailors S
      GROUP BY S.rating) AS Temp
WHERE Temp.avgage = (SELECT MIN(Temp.avgage)
                      FROM Temp)
```

In-Class Exercise

```
Suppliers(sid: integer, sname: string, address: string)
Parts(pid: integer, pname: string, color: string)
Catalog(sid: integer, pid: integer, cost: real)
```

- Find the sids of suppliers who charge more for some part than the average cost of that part (averaged over all the suppliers who supply that part).
- For each part, find the sname of the supplier who charges the most for that part.
- Find the sids of suppliers who supply only red parts.
- For every supplier that only supplies green parts, print the name of the supplier and the total number of parts that she supplies.
- For every supplier that supplies a green part and a red part, print the name and price of the most expensive part that she supplies.
Null Values

- Field values in a tuple are sometimes unknown (e.g., a rating has not been assigned) or inapplicable (e.g., no spouse’s name).
- SQL provides a special value null for such situations.
- The presence of null complicates many issues. E.g.:
  - Special operators needed to check if value is/is not null.
  - Is rating > 8 true or false when rating is equal to null? What about AND, OR and NOT connectives?
  - We need a 3-valued logic (true, false and unknown).
  - Meaning of constructs must be defined carefully. (e.g., WHERE clause eliminates rows that don’t evaluate to true.)
  - New operators (in particular, outer joins) possible/needed.

Integrity Constraints (Review)

- An IC describes conditions that every legal instance of a relation must satisfy.
  - Inserts/deletes/updates that violate IC’s are disallowed.
  - Can be used to ensure application semantics (e.g., sid is a key), or prevent inconsistencies (e.g., sname has to be a string, age must be < 200)
- Types of ICs: Domain constraints, primary key constraints, foreign key constraints, general constraints.
  - Domain constraints: Field values must be of right type. Always enforced.

CREATE TABLE Sailors
    ( sid INTEGER,
      sname CHAR(10),
      rating INTEGER,
      age REAL,
      PRIMARY KEY  (sid),
      CHECK ( rating >= 1
               AND rating <= 10 ) )

CREATE TABLE Reserves
    ( sname CHAR(10),
      bid  INTEGER,
      day  DATE,
      PRIMARY KEY  (bid,day),
      CONSTRAINT noInterlakeRes
        CHECK ( 'Interlake' <> ( SELECT B.bname
                                    FROM Boats B
                                    WHERE B.bid=bid))))

General Constraints

- Useful when more general ICs than keys are involved.
- Can use queries to express constraint.
- Constraints can be named.
Constraints Over Multiple Relations

CREATE TABLE Sailors
(sid INTEGER,
sname CHAR(10),
rating INTEGER,
age REAL,
PRIMARY KEY (sid),
CHECK
(SELECT COUNT (S.sid) FROM Sailors S) + (SELECT COUNT (B.bid) FROM Boats B) < 100
)

• Awkward and wrong!
• If Sailors is empty, the number of Boats tuples can be anything!
• ASSERTION is the right solution; not associated with either table.

Triggers

• Trigger: Procedure that starts automatically if specified changes occur to the DBMS
• Three parts:
  • Event (activates the trigger)
  • Condition (tests whether the triggers should run)
  • Action (what happens if the trigger runs)

Triggers: Example (SQL:1999)

CREATE TRIGGER youngSailorUpdate
AFTER INSERT ON SAILORS
REFERENCING NEW TABLE NewSailors
FOR EACH STATEMENT
INSERT INTO YoungSailors(sid, name, age, rating)
SELECT sid, name, age, rating
FROM NewSailors N
WHERE N.age <= 18
### SQL in Application Code

- Embedded SQL
- Cursors
- Dynamic SQL
- JDBC
- SQLJ
- Stored procedures

### SQL in Application Code

- SQL commands can be called from within a host language (e.g., C++ or Java) program.
  - SQL statements can refer to host variables (including special variables used to return status).
  - Must include a statement to connect to the right database.
- Two main integration approaches:
  - Embed SQL in the host language (Embedded SQL, SQLJ)
  - Create special API to call SQL commands (JDBC)

### SQL in Application Code (Contd.)

**Impedance mismatch:**
- SQL relations are (multi-) sets of records, with no *a priori* bound on the number of records. No such data structure exist traditionally in procedural programming languages such as C++. (Though now: STL)
  - SQL supports a mechanism called a **cursor** to handle this.
Embedded SQL

• Approach: Embed SQL in the host language.
  • A preprocessor converts the SQL statements into special API calls.
  • Then a regular compiler is used to compile the code.

• Language constructs:
  • Connecting to a database: EXEC SQL CONNECT
  • Declaring variables: EXEC SQL BEGIN (END) DECLARE SECTION
  • Statements: EXEC SQL Statement

Embedded SQL: Variables

EXEC SQL BEGIN DECLARE SECTION
char c_sname[20];
long c_sid;
short c_rating;
float c_age;
EXEC SQL END DECLARE SECTION

• Two special “error” variables:
  • SQLCODE (long, is negative if an error has occurred)
  • SQLSTATE (char[6], predefined codes for common errors)

Cursors

• Can declare a cursor on a relation or query statement (which generates a relation).
• Can open a cursor, and repeatedly fetch a tuple then move the cursor, until all tuples have been retrieved.
• Can use a special clause, called ORDER BY, in queries that are accessed through a cursor, to control the order in which tuples are returned.
  • Fields in ORDER BY clause must also appear in SELECT clause.
  • The ORDER BY clause, which orders answer tuples, is only allowed in the context of a cursor.
• Can also modify/delete tuple pointed to by a cursor.
Cursor that gets names of sailors who’ve reserved a red boat, in alphabetical order

EXEC SQL DECLARE sinfo CURSOR FOR
SELECT S.sname
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid AND B.color='red'
ORDER BY S.sname

• Note that it is illegal to replace S.sname by, say, S.sid in the ORDER BY clause! (Why?)
• Can we add S.sid to the SELECT clause and replace S.sname by S.sid in the ORDER BY clause?

Embedding SQL in C: An Example

char SQLSTATE[6];
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20]; short c_minrating; float c_age;
EXEC SQL END DECLARE SECTION
c_minrating = random();
EXEC SQL DECLARE sinfo CURSOR FOR
SELECT S.sname, S.age FROM Sailors S
WHERE S.rating > c_minrating
ORDER BY S.sname;
do {
EXEC SQL FETCH sinfo INTO :c_sname, :c_age;
printf("%s is %d years old\n", c_sname, c_age);
} while (SQLSTATE != '02000');
EXEC SQL CLOSE sinfo;

Dynamic SQL

• SQL query strings are now always known at compile time (e.g., spreadsheet, graphical DBMS frontend): Allow construction of SQL statements on-the-fly

• Example:
char c_sqlstring[]="DELETE FROM Sailors WHERE rating>5";
EXEC SQL PREPARE readytogo FROM :
c_sqlstring;
EXEC SQL EXECUTE readytogo;
Database APIs: Alternative to Embedding

• Rather than modify compiler, add library with database calls (API)
• Special standardized interface: procedures/objects
• Pass SQL strings from language, presents result sets in a language-friendly way
• Sun’s JDBC: Java API
• Supposedly DBMS-neutral
  • a “driver” traps the calls and translates them into DBMS-specific code
  • database can be across a network

JDBC: Architecture

• Four architectural components:
  • Application (initiates and terminates connections, submits SQL statements)
  • Driver manager (load JDBC driver)
  • Driver (connects to data source, transmits requests and returns/translations results and error codes)
  • Data source (processes SQL statements)

JDBC Architecture (Contd.)

Four types of drivers:
Bridge:
  • Translates SQL commands into non-native API.
  Example: JDBC-ODBC bridge. Code for ODBC and JDBC driver needs to be available on each client.
Direct translation to native API, non-Java driver:
  • Translates SQL commands to native API of data source. Need OS-specific binary on each client.
Network bridge:
  • Send commands over the network to a middleware server that talks to the data source. Needs only small JDBC driver at each client.
Direction translation to native API via Java driver:
  • Converts JDBC calls directly to network protocol used by DBMS. Needs DBMS-specific Java driver at each client.
JDBC Classes and Interfaces

Steps to submit a database query:
- Load the JDBC driver
- Connect to the data source
- Execute SQL statements

JDBC Driver Management

- All drivers are managed by the DriverManager class
- Loading a JDBC driver:
  - In the Java code:
    Class.forName("oracle/jdbc.driver.OracleDriver");
  - When starting the Java application:
    -Djdbc.drivers=oracle/jdbc.driver

Connections in JDBC

We interact with a data source through sessions.
Each connection identifies a logical session.
- JDBC URL:
  jdbc:<subprotocol>:<otherParameters>

Example:
String url="jdbc:oracle:www.bookstore.com:3083";
Connection con;
try{
  con = DriverManager.getConnection(url, usedId, password);
} catch (SQLException excpt) { ...}
Connection Class Interface

- public int getTransactionIsolation() and void setTransactionIsolation(int level)
  Sets isolation level for the current connection.
- public boolean getReadOnly() and void setReadOnly(boolean b)
  Specifies whether transactions in this connection are read-only.
- public boolean getAutoCommit() and void setAutoCommit(boolean b)
  If autocommit is set, then each SQL statement is considered its own transaction. Otherwise, a transaction is committed using commit(), or aborted using rollback().
- public boolean isClosed()
  Checks whether connection is still open.

Executing SQL Statements

- Three different ways of executing SQL statements:
  - Statement (both static and dynamic SQL statements)
  - PreparedStatement (semi-static SQL statements)
  - CallableStatement (stored procedures)

- PreparedStatement class:
  Precompiled, parametrized SQL statements:
  - Structure is fixed
  - Values of parameters are determined at run-time

Executing SQL Statements (Contd.)

String sql="INSERT INTO Sailors VALUES(?,?,?,?)";
PreparedStatement pstmt=con.prepareStatement(sql);
pstmt.clearParameters();
pstmt.setInt(1,sid);
pstmt.setString(2,sname);
pstmt.setInt(3, rating);
pstmt.setFloat(4,age);

// we know that no rows are returned, thus we use executeUpdate()
int numRows = pstmt.executeUpdate();
ResultSets

- PreparedStatement.executeUpdate only returns the number of affected records
- PreparedStatement.executeQuery returns data, encapsulated in a ResultSet object (a cursor)

```java
ResultSet rs = pstmt.executeQuery(sql);
// rs is now a cursor
while (rs.next()) {
    // process the data
}
```

ResultSets (Contd.)

A ResultSet is a very powerful cursor:
- previous(): moves one row back
- absolute(int num): moves to the row with the specified number
- relative (int num): moves forward or backward
- first() and last()

Matching Java and SQL Data Types

<table>
<thead>
<tr>
<th>SQL Type</th>
<th>java class</th>
<th>ResultSet get method</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>Boolean</td>
<td>getBoolean()</td>
</tr>
<tr>
<td>CHAR</td>
<td>String</td>
<td>getString()</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>String</td>
<td>getString()</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Double</td>
<td>getDouble()</td>
</tr>
<tr>
<td>FLOAT</td>
<td>Double</td>
<td>getDouble()</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Integer</td>
<td>getInt()</td>
</tr>
<tr>
<td>REAL</td>
<td>Double</td>
<td>getFloat()</td>
</tr>
<tr>
<td>DATE</td>
<td>java.sql.Date</td>
<td>getDate()</td>
</tr>
<tr>
<td>TIME</td>
<td>java.sql.Time</td>
<td>getTime()</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>java.sql.Timestamp</td>
<td>getTimestamp()</td>
</tr>
</tbody>
</table>
JDBC: Exceptions and Warnings

- Most of java.sql can throw and SQLException if an error occurs.
- SQLWarning is a subclass of SQLException; not as severe (they are not thrown and their existence has to be explicitly tested)

Warning and Exceptions (Contd.)

```java
try {
    stmt = con.createStatement();
    warning = con.getWarnings();
    while (warning != null) {
        // handle SQLWarnings;
        warning = warning.getNextWarning();
    }
    con.clearWarnings();
    stmt.executeUpdate(queryString);
    warning = con.getWarnings();
    ...
} //end try
catch (SQLException sqle) {
    // handle the exception
}
```

Examining Database Metadata

DatabaseMetaData object gives information about the database system and the catalog.

```java
DatabaseMetaData md = con.getMetaData();
// print information about the driver:
System.out.println(
    "Name:" + md.getDriverName() +
    "version: " + md.getDriverVersion());
```
Database Metadata (Contd.)

```java
DatabaseMetaData md=con.getMetaData();
ResultSet trs=md.getTables(null,null,null,null);
String tableName;
While(trs.next()) {
    tableName = trs.getString("TABLE_NAME");
    System.out.println("Table: " + tableName);
    //print all attributes
    ResultSet crs = md.getColumns(null,null,tableName, null);
    while (crs.next()) {
        System.out.println(crs.getString("COLUMN_NAME") + ", ");
    }
}
```

A (Semi-)Complete Example

```java
Connection con = // connect
DriverManager.getConnection(url, "login", "pass");
Statement stmt = con.createStatement(); // set up stmt
String query = "SELECT name, rating FROM Sailors";
ResultSet rs = stmt.executeQuery(query);
try { // handle exceptions
    // loop through result tuples
    while (rs.next()) {
        String s = rs.getString("name");
        Int n = rs.getFloat("rating");
        System.out.println(s + "   " + n);
    }
} catch(SQLException ex) {
    System.out.println(ex.getMessage ()
                      + ex.getSQLState ()
                      + ex.getErrorCode ());
}
```

SQLJ

- Complements JDBC with a (semi-)static query model:
  - Compiler can perform syntax checks, strong type checks, consistency of the query with the schema
  - All arguments always bound to the same variable:
    ```sql
    #sql = {
        SELECT name, rating INTO :name, :rating
        FROM Books WHERE sid = :sid;
    }
    ```
  - Compare to JDBC:
    ```java
    sid=rs.getInt(1);
    if (sid==1) {sname=rs.getString(2);}
    else { sname2=rs.getString(2);}
    ```
- SQLJ (part of the SQL standard) versus embedded SQL (vendor-specific)
**SQLJ Code**

```sql
Int sid; String name; Int rating;
// named iterator
#sql iterator Sailors(Int sid, String name, Int rating);
Sailors sailors;
// assume that the application sets rating
#sailors = {
    SELECT sid, sname INTO :sid, :name
    FROM Sailors WHERE rating = :rating
};
// retrieve results
while (sailors.next()) {
    System.out.println(sailors.sid + " " + sailors.sname);
}
sailors.close();
```

**SQLJ Iterators**

Two types of iterators ("cursors"):
- Named iterator
  - Need both variable type and name, and then allows retrieval of columns by name.
  - See example on previous slide.
- Positional iterator
  - Need only variable type, and then uses FETCH .. INTO construct:
    ```sql
    #sql iterator Sailors(Int, String, Int);
    Sailors sailors;
    #sailors = ...
    while (true) {
        #sql {FETCH :sailors INTO :sid, :name} ;
        if (sailors.endFetch()) { break; }
        // process the sailor
    }
    ```

**Stored Procedures**

- What is a stored procedure:
  - Program executed through a single SQL statement
  - Executed in the process space of the server
- Advantages:
  - Can encapsulate application logic while staying "close" to the data
  - Reuse of application logic by different users
  - Avoid tuple-at-a-time return of records through cursors
Stored Procedures: Examples

CREATE PROCEDURE ShowNumReservations
    SELECT S.sid, S.sname, COUNT(*)
    FROM Sailors S, Reserves R
    WHERE S.sid = R.sid
    GROUP BY S.sid, S.sname

Stored procedures can have parameters:
- Three different modes: IN, OUT, INOUT

CREATE PROCEDURE IncreaseRating(
    IN sailor_sid INTEGER, IN increase INTEGER)
UPDATE Sailors
    SET rating = rating + increase
WHERE sid = sailor_sid

Stored Procedures: Examples (Contd.)

Stored procedure do not have to be written in SQL:

CREATE PROCEDURE TopSailors(
    IN num INTEGER)
    LANGUAGE JAVA
    EXTERNAL NAME
        "file:///c:/storedProcs/rank.jar"

Calling Stored Procedures

EXEC SQL BEGIN DECLARE SECTION
    Int sid;
    Int rating;
EXEC SQL END DECLARE DECLARE SECTION

// now increase the rating of this sailor
EXEC CALL IncreaseRating(sid,:rating);
Calling Stored Procedures (Contd.)

**JDBC:**
```
CallableStatement cstmt = con.prepareCall("call ShowSailors");
ResultSet rs = cstmt.executeQuery();
while (rs.next()) {
    ...
}
```

**SQL:**
```
#sql iterator
ShowSailors(...);
ShowSailors showsailors;
#sql showsailors={CALL ShowSailors};
while (showsailors.next()) {
    ...
}
```

---

**SQL/PSM**

Most DBMSs allow users to write stored procedures in a simple, general-purpose language (close to SQL) → SQL/PSM standard is a representative

- **Declare a stored procedure:**
  
  ```
  CREATE PROCEDURE name(p1, p2, ..., pn) 
  
  local variable declarations 
  
  procedure code; 
  ```

- **Declare a function:**
  
  ```
  CREATE FUNCTION name (p1, ..., pn) RETURNS sqlDataType 
  
  local variable declarations 
  
  function code; 
  ```

---

**Main SQL/PSM Constructs**

```
CREATE FUNCTION rate Sailor 
(IN sailorId INTEGER) 
RETURNS INTEGER 
DECLARE rating INTEGER 
DECLARE numRes INTEGER 
SET numRes = (SELECT COUNT(*) 
    FROM Reserves R 
    WHERE R.sid = sailorId) 
IF (numRes > 10) THEN rating = 1; 
ELSE rating = 0; 
END IF; 
RETURN rating; 
```
Main SQL/PSM Constructs (Contd.)

- Local variables (DECLARE)
- RETURN values for FUNCTION
- Assign variables with SET
- Branches and loops:
  - IF (condition) THEN statements;
  - ELSEIF (condition) statements;
  - ELSE statements; END IF;
  - LOOP statements; END LOOP
- Queries can be parts of expressions
- Can use cursors naturally without “EXEC SQL”

Summary

- Embedded SQL allows execution of parametrized static queries within a host language
- Dynamic SQL allows execution of completely ad-hoc queries within a host language
- Cursor mechanism allows retrieval of one record at a time and bridges impedance mismatch between host language and SQL
- APIs such as JDBC introduce a layer of abstraction between application and DBMS

Summary (Contd.)

- SQLJ: Static model, queries checked at compile-time.
- Stored procedures execute application logic directly at the server
- SQL/PSM standard for writing stored procedures