SQL: The Query Language  
Part 2  
R & G - Chapter 5  
The important thing is not to stop questioning.  
Albert Einstein  

Example Database  

<table>
<thead>
<tr>
<th>Sailors</th>
<th>Boats</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>bname</td>
</tr>
<tr>
<td>22</td>
<td>Dustin</td>
</tr>
<tr>
<td>31</td>
<td>Luader</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
</tr>
<tr>
<td>101</td>
<td>Interlake</td>
</tr>
<tr>
<td>102</td>
<td>Interlake</td>
</tr>
<tr>
<td>103</td>
<td>Clipper</td>
</tr>
<tr>
<td>104</td>
<td>Marine</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reserves</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>95</td>
</tr>
</tbody>
</table>

Conceptual SQL Evaluation  

- **SELECT** 
  - Target list 
  - FROM relation list 
  - WHERE qualification 
  - GROUP BY grouping list 
  - HAVING group qualification 

- Apply selections (eliminate rows) 
- Project away columns (just keep those used in SELECT, GBY, HAVING) 
- Relation cross-product  

Sorting the Results of a Query  

- **ORDER BY column [ASC | DESC] [ , ...]**
  
  ```sql
  SELECT S.rating, S.sname, S.age
  FROM Sailors S, Boats B, Reserves R
  WHERE S.sid = R.sid
  AND R.bid = B.bid AND B.color = 'red'
  ORDER BY S.rating, S.sname;
  ```

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

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.  

Joins  

- **SELECT (column_list)** 
  FROM table_name 
  [INNER | [LEFT | RIGHT | FULL] OUTER] JOIN table_name 
  ON qualification_list 
  WHERE ...

- Explicit join semantics needed unless it is an INNER join (INNER is default)
**Inner Join**

Only the rows that match the search conditions are returned.

```
SELECT s.sid, s.name, r.bid
FROM Sailors s INNER JOIN Reserves r
ON s.sid = r.sid
```

Returns only those sailors who have reserved boats SQL-92 also allows:

```
SELECT s.sid, s.name, r.bid
FROM Sailors s NATURAL JOIN Reserves r
```

“NATURAL” means equi-join for each pair of attributes with the same name

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>bid</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>101</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>103</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>103</td>
</tr>
</tbody>
</table>

**Left Outer Join**

Left Outer Join returns all matched rows, plus all unmatched rows from the table on the left of the join clause (use nulls in fields of non-matching tuples)

```
SELECT s.sid, s.name, r.bid
FROM Sailors s LEFT OUTER JOIN Reserves r
ON s.sid = r.sid
```

Returns all sailors & information on whether they have reserved boats

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>bid</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>101</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>103</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>103</td>
</tr>
</tbody>
</table>

**Right Outer Join**

Right Outer Join returns all matched rows, plus all unmatched rows from the table on the right of the join clause

```
SELECT r.sid, b.bid, b.name
FROM Reserves r RIGHT OUTER JOIN Boats b
ON r.bid = b.bid
```

Returns all boats & information on which ones are reserved.

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>bid</th>
<th>bname</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
<td>101</td>
<td>Interlake</td>
<td>blue</td>
</tr>
<tr>
<td>95</td>
<td>103</td>
<td>11/12/96</td>
<td>102</td>
<td>Interlake</td>
<td>red</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>103</td>
<td>Clipper</td>
<td>green</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>104</td>
<td>Marine</td>
<td>red</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>95</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
Full Outer Join

Full Outer Join returns all (matched or unmatched) rows from the tables on both sides of the join clause

```sql
SELECT r.sid, b.bid, b.name
FROM Reserves r FULL OUTER JOIN Boats b
ON r.bid = b.bid
```

Returns all boats & all information on reservations

Note: in this case it is the same as the ROJ because bid is a foreign key in reserves, so all reservations must have a corresponding tuple in boats.

Views: Defining External DB Schemas

```sql
CREATE VIEW view_name
AS select_statement
```

Makes development simpler
Often used for security
Not instantiated - makes updates tricky

```sql
CREATE VIEW Reds
AS SELECT B.bid, COUNT(*) AS scount
FROM Boats B, Reserves R
WHERE R.bid=B.bid AND B.color='red'
GROUP BY B.bid
```

Views Instead of Relations in Queries

```sql
CREATE VIEW Reds
AS SELECT B.bid, COUNT(*) AS scount
FROM Boats B, Reserves R
WHERE R.bid=B.bid AND B.color='red'
GROUP BY B.bid
```

```sql
SELECT bname, scount
FROM Reds R
WHERE R.bid=B.bid
AND scount < 10
```

Discretionary Access Control

```sql
GRANT privileges ON object TO users
[WITH GRANT OPTION]
```

- Object can be a Table or a View
- Privileges can be:
  - Select
  - Insert
  - Delete
  - References (cols) – allow to create a foreign key that references the specified column(s)
  - All
  - Can later be REVOKEd
  - Users can be single users or groups
  - See Chapter 17 for more details.

Two more important topics

- Constraints
- SQL embedded in other languages
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 IC's: Domain constraints, primary key constraints, foreign key constraints, general constraints.

- Domain constraints: Field values must be of right type. Always enforced.
- Primary key and foreign key constraints: you know them.

Types of IC's:
- Domain constraints: Field values must be of right type. Always enforced.
- Primary key and foreign key constraints: you know them.
- General Constraints: Useful when more general ICs than keys are involved.
  - Can use queries to express constraint.
  - Checked on insert or update.
  - Constraints can be named.

```sql
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))
```

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 )
```

CREATE ASSERTION smallClub
CHECK ( (SELECT COUNT(S.sid) FROM Sailors S) + (SELECT COUNT(B.bid) FROM Boats B) < 100 )
```

Writing Applications with SQL

- SQL is not a general purpose programming language.
  + Tailored for data retrieval and manipulation
  + Relatively easy to optimize and parallelize
  - Can't write entire apps in SQL alone

Options:
- Make the query language "Turing complete"
  - Avoids the "impedance mismatch"
  - But, loses advantages of relational language simplicity
- Allow SQL to be embedded in regular programming languages.
- Q: What needs to be solved to make the latter approach work?

Embedded SQL

- DBMS vendors traditionally provided "host language bindings"
  - E.g. for C or COBOL
  - Allow SQL statements to be called from within a program
  - Typically you preprocess your programs
  - Preprocessor generates calls to a proprietary DB connectivity library
- General pattern
  - One call to connect to the right database (login, etc.)
  - SQL statements can refer to host variables from the language
- Typically vendor-specific
  - We won't look at any in detail, we'll look at standard stuff
- Problem
  - SQL relations are (multi-)sets, no a priori bound on the number of records. No such data structure in C.
  - SQL supports a mechanism called a cursor to handle this.

Just to give you a flavor

```sql
EXEC SQL SELECT S.sname, S.age
INTO :c_sname, :c_age
FROM Sailors S
WHERE S.sid = :c_sid
```
Cursors

- Can declare a cursor on a relation or query
- Can open a cursor
- Can repeatedly fetch a tuple (moving the cursor)
- ORDER BY allows control over the order in which tuples are returned.
- Fields in ORDER BY clause must also appear in SELECT clause.
- Can also modify/delete tuple pointed to by a cursor
  - A "non-relational" way to get a handle to a particular tuple
- There’s an Embedded SQL syntax for cursors
  - DECLARE <cursorname> CURSOR FOR <select stmt>
  - FETCH FROM <cursorname> INTO <variable names>
  - But we’ll use JDBC instead

Database APIs: Alternative to embedding

- Rather than modify compiler, add a library with database calls (API)
  - special procedures/objects
  - passes SQL strings from language, presents result sets in a language-friendly way
  - ODBC a C/C++ standard started on Windows
  - JDBC a Java equivalent
  - Most scripting languages have similar things
  - E.g. For Perl there is DBI, "oraPerl", other packages
- Mostly DBMS-neutral
  - at least try to hide distinctions across different DBMSs

Architecture

- A lookup service maps "data source names" ("DSNs") to drivers
- Typically handled by OS
- Based on the DSN used, a "driver" is linked into the app at runtime
- The driver traps calls, translates them into DBMS-specific code
- Database can be across a network
- ODBC is standard, so the same program can be used (in principle) to access multiple database systems
- Data source may not even be an SQL database!

ODBC/JDBC

- Various vendors provide drivers
  - MS bundles a bunch into Windows
  - Vendors like DataDirect and OpenLink sell drivers for multiple OSes
- Drivers for various data sources
  - Relational DBMSs (Oracle, DB2, SQL Server, etc.)
  - "Desktop" DBMSs (Access, Dbase, Paradox, FoxPro, etc.)
  - Spreadsheets (MS Excel, Lotus 1-2-3, etc.)
  - Delimited text files (.CSV, .TXT, etc.)
- You can use JDBC/ODBC clients over many data sources
  - E.g. MS Query comes with many versions of MS Office (msqry32.exe)
- Can write your own Java or C++ programs against xDBC

JDBC

- Part of Java, very easy to use
- Java comes with a JDBC-to-ODBC bridge
  - So JDBC code can talk to any ODBC data source
  - E.g. look in your Windows Control Panel for JDBC/ODBC drivers!
- JDBC tutorial online

JDBC Basics: Connections

- A Connection is an object representing a login to a database
  // GET CONNECTION
  Connection conn;
  try {
    conn = DriverManager.getConnection("jdbc:odbc:sailorsDB", userName, password);
  } catch(Exception e) { System.out.println(e); }
- Eventually you close the connection
  // CLOSE CONNECTION
  try { conn.close(); }
  catch (Exception e) { System.out.println(e); }
**JDBC Basics: Statements**

- You need a Statement object for each SQL statement

```java
// CREATE STATEMENT
Statement stmt;
try {
    stmt = con.createStatement();
} catch (Exception e) {
    System.out.println(e);
}
```

Soon we'll say `stmt.executeQuery("select ...");`

---

**CreateStatement cursor behavior**

- Two optional args to `createStatement`:
  - `createStatement(ResultSet.TYPE, ResultSet.CONCUR)`
  - Corresponds to SQL cursor features
- `<TYPE>` is one of:
  - `TYPE_FORWARD_ONLY`: can't move cursor backward
  - `TYPE_SCROLL_INSENSITIVE`: can move backward, but doesn't show results of any updates
  - `TYPE_SCROLL_SENSITIVE`: can move backward, will show updates made while result set is open
- `<CONCUR>` is one of:
  - `CONCUR_READ_ONLY`: this statement doesn't allow updates
  - `CONCUR_UPDATABLE`: this statement allows updates
- Defaults:
  - `TYPE_FORWARD_ONLY` and `CONCUR_READ_ONLY`

---

**JDBC Basics: ResultSet**

- A `ResultSet` object serves as a cursor for the statement's results

```java
// EXECUTE QUERY
ResultSet results;
try {
    results = stmt.executeQuery("select * from Sailors");
} catch (Exception e) {
    System.out.println(e);
}
```

- Obvious handy methods:
  - `results.next()` advances cursor to next tuple
  - Returns `false` when the cursor slides off the table (beginning or end)
  - `scrollable` cursors:
    - `results.previous()`, `results.relative(int)`, `results.absolute(int)`, `results.first()`, `results.last()`, `results.beforeFirst()`, `results.afterLast()`

---

**ResultSet Metadata**

- Can find out stuff about the ResultSet schema via `ResultSetMetaData`

```java
ResultSetMetaData rsmd = results.getMetaData();
int numCols = rsmd.getColumnCount();
int i, rowcount = 0;
// get column header info
for (i=1; i <= numCols; i++) {
    if (i > 1)
        buf.append(",");
    buf.append(rsmd.getColumnLabel(i));
} buf.append("n");
```

- Other `ResultSetMetaData` methods:
  - `getColumnType(i)`, `isNullable(i)`, etc.

---

**Getting Values in Current of Cursor**

- `getString`

```java
// Break it off at 100 rows max
while (results.next()) as count < 100) {
    // Loop through each column, getting the
    // column data and displaying
    for (i=1; i <= numCols; i++) {
        if (i > 1)
            buf.append(",");
        buf.append(results.getString(i));
    } buf.append("n");
    rowcount++;
}
```

- Similarly, `getFloat`, `getInt`, etc.

---

**Updating Current of Cursor**

- Update fields in current of cursor:

```java
result.next();
result.updateInt("Rating", 10);
```

- Also `updateString`, `updateFloat`, etc.
- Or can always submit a full SQL UPDATE statement
  - Via `executeQuery()`

- The original statement must have been `CONCUR_UPDATABLE` in either case!
Cleaning up Neatly

```java
try {
    // CLOSE RESULT SET
    results.close();
    // CLOSE STATEMENT
    stmt.close();
    // CLOSE CONNECTION
    con.close();
} catch (Exception e) {
    System.out.println(e);
}
```

Putting it Together (w/o try/catch)

```java
Connection con =
    DriverManager.getConnection("jdbc:odbc:weblog",userName,password);
Statement stmt = con.createStatement();
ResultSet results = stmt.executeQuery("select * from Sailors");
ResultSetMetaData rsmd = results.getMetaData();
int numCols = rsmd.getColumnCount();
StringBuffer buf = new StringBuffer();
while (results.next() && rowcount < 100) {
    for (i=1; i <= numCols; i++) {
        if (i > 1) buf.append(",");
        buf.append(results.getString(i));
    }
    buf.append("
");
}
results.close(); stmt.close(); con.close();
```

Similar deal for web scripting languages

- **Common scenario today is to have a web client**
  - A web form issues a query to the DB
  - Results formatted as HTML
- **Many web scripting languages used**
  - jsp, asp, PHP, Ruby, etc.
  - most of these are similar, look a lot like JDBC with HTML mixed in

E.g. PHP/Postgres

```php
<?php
$conn = pg_pconnect("dbname=cowbook user=jmh\password=secret");
if (!$conn) {
    echo "An error occurred.\n";
    exit;
}
$result = pg_query ($conn, "SELECT * FROM Sailors");
if (!$result) {
    echo "An error occurred.\n";
    exit;
}
$num = pg_num_rows($result);
for ($i=0; $i < $num; $i++) {
    for ($j=0; $j < count($r); $j++) {
        echo "$r[$j]nbsp;";
    }
    echo "<BR>");
    if (!$i) {
        echo "<HR>";
    }
}
?>
```

API Summary

**APIs are needed to interface DBMSs to programming languages**

- Embedded SQL uses "native drivers" and is usually faster but less standard
- ODBC (used to be Microsoft-specific) for C/C++
- JDBC the standard for Java
- Scripting languages (PHP, Perl, JSP) are becoming the preferred technique for web-based systems