CAS CS 460/660
Introduction to Database Systems

SQL III
CREATE VIEW  view_name
AS select_statement

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

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
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

<table>
<thead>
<tr>
<th>b.bid</th>
<th>scount</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>1</td>
</tr>
</tbody>
</table>

Reds
Views

create view vs INTO

(1) SELECT bname, bcity
    FROM   branch
    INTO   branch2

(2) CREATE VIEW branch2 AS
    SELECT  bname, bcity
    FROM     branch

(1) creates new table that gets stored on disk

(2) creates “virtual table” (materialized when needed)

Therefore: changes in branch are seen in the view version of branch2 (2) but not for the (1) case.
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.

```sql
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:

```sql
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 (may need to rename with “AS”)

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

<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>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</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>

<table>
<thead>
<tr>
<th>s.sid</th>
<th>s.name</th>
<th>r.bid</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>101</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>103</td>
</tr>
</tbody>
</table>
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
SELECT s.sid, s.name, r.bid 
FROM Sailors s LEFT OUTER JOIN Reserves r 
ON s.sid = r.sid 

<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>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</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>

<table>
<thead>
<tr>
<th>s.sid</th>
<th>s.name</th>
<th>r.bid</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>101</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>103</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>null</td>
</tr>
</tbody>
</table>
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.
SELECT r.sid, b.bid, b.name
FROM Reserves r RIGHT OUTER JOIN Boats b
ON r.bid = b.bid

<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>

<table>
<thead>
<tr>
<th>bid</th>
<th>bname</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Interlake</td>
<td>blue</td>
</tr>
<tr>
<td>102</td>
<td>Interlake</td>
<td>red</td>
</tr>
<tr>
<td>103</td>
<td>Clipper</td>
<td>green</td>
</tr>
<tr>
<td>104</td>
<td>Marine</td>
<td>red</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>r.sid</th>
<th>b.bid</th>
<th>b.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>Interlake</td>
</tr>
<tr>
<td>null</td>
<td>102</td>
<td>Interlake</td>
</tr>
<tr>
<td>95</td>
<td>103</td>
<td>Clipper</td>
</tr>
<tr>
<td>null</td>
<td>104</td>
<td>Marine</td>
</tr>
</tbody>
</table>
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 Sailors s FULL OUTER JOIN Boats b
ON s.sname = b.bname
```
### SQL Query

```sql
SELECT s.sid, s.sname, b.bid, b.name
FROM Sailors s FULL OUTER JOIN Boats b
ON s.sname = b.bname
```

### Table 1: Sailors

<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>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</td>
</tr>
</tbody>
</table>

### Table 2: Boats

<table>
<thead>
<tr>
<th>bid</th>
<th>bname</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Interlake</td>
<td>blue</td>
</tr>
<tr>
<td>105</td>
<td>Lubber</td>
<td>purple</td>
</tr>
</tbody>
</table>

### Table 3: Joined Data

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>bid</th>
<th>bname</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>105</td>
<td>Lubber</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>null</td>
<td>null</td>
<td>101</td>
<td>Interlake</td>
</tr>
</tbody>
</table>
CREATE TABLE table_name
( { column_name data_type
    [ DEFAULT default_expr ] [ column_constraint [, ... ] ] | table_constraint } [, ... ] )

Data Types (PostgreSQL) include:
character(n) – fixed-length character string
character varying(n) – variable-length character string
smallint, integer, bigint, numeric, real, double precision
date, time, timestamp, …
serial - unique ID for indexing and cross reference

PostgreSQL also allows OIDs and other “system types”, arrays, inheritance, rules…
conformance to the SQL-1999 standard is variable.
Constraints

- Recall that the schema defines the legal instances of the relations.

- Data types are a way to limit the kind of data that can be stored in a table, but they are often insufficient.
  
  - e.g., prices must be positive values
  - uniqueness, referential integrity, etc.

- Can specify constraints on individual columns or on tables.
Column constraints

[ CONSTRAINT constraint_name ]
  { NOT NULL | NULL | UNIQUE | PRIMARY KEY | CHECK (expression) |
  REFERENCES reftable [ ( refcolumn ) ] [ ON DELETE action ]
  [ ON UPDATE action ] }

primary key = unique + not null; also used as default target for references. (can have at most 1)

expression must produce a boolean result and reference that column’s value only.

references is for foreign keys; action is one of:
  NO ACTION, CASCADE, SET NULL, SET DEFAULT
Table constraints

- CREATE TABLE `table_name`
  ( { `column_name` `data_type` [ DEFAULT `default_expr` ]
    [ `column_constraint` [, ... ] ] | `table_constraint` } [, ... ] )

Table Constraints:

- [ CONSTRAINT `constraint_name` ]
  { UNIQUE ( `column_name` [, ... ] ) |
    PRIMARY KEY ( `column_name` [, ... ] ) |
    CHECK ( `expression` ) |
    FOREIGN KEY ( `column_name` [, ... ] ) REFERENCES `reftable`
    [ ( `refcolumn` [, ... ] ) ] [ ON DELETE `action` ] [ ON UPDATE `action` ] }

Here, expressions, etc can include multiple columns
CREATE TABLE films (  
    code     CHAR(5) PRIMARY KEY,  
    title    VARCHAR(60),  
    did      DECIMAL(3),  
    date_prod DATE,  
    kind     VARCHAR(10),  
  CONSTRAINT production UNIQUE(date_prod)  
  FOREIGN KEY did REFERENCES distributors  
    ON DELETE NO ACTION  
);  
CREATE TABLE distributors (  
    did      DECIMAL(3) PRIMARY KEY,  
    name     VARCHAR(40)  
  CONSTRAINT con1 CHECK (did > 100 AND name <> ' ')  
);
Other DDL Statements

- **Alter Table**
  - use to add/remove columns, constraints, rename things …

- **Drop Table**
  - Compare to “Delete * From Table”

- **Create/Drop View**

- **Create/Drop Index**

- **Grant/Revoke privileges**
  - SQL has an authorization model for saying who can read/modify/delete etc. data and who can grant and revoke privileges!
SQL: Modification Commands

Deletion:  

```
DELETE FROM <relation>
[WHERE <predicate>]
```

Example:  

```
1. DELETE FROM account
   -- deletes all tuples in account

2. DELETE FROM account
   WHERE bname IN (SELECT bname
                     FROM branch
                     WHERE bcity = 'Bkln')
   -- deletes all accounts from Brooklyn branch
```
Delete the record of all accounts with balances below the average at the bank.

```
DELETE FROM account
WHERE balance < (SELECT AVG(balance)
FROM account)
```

Problem: as we delete tuples from deposit, the average balance changes.

Solution used in SQL:

1. First, compute `avg` balance and find all tuples to delete
2. Next, delete all tuples found above (without recomputing `avg` or retesting the tuples)
SQL: Modification Commands

Insertion: \[ \text{INSERT INTO } <\text{relation}> \text{ values } (\ldots, \ldots, \ldots) \]

or \[ \text{INSERT INTO } <\text{relation}> (\text{att1}, \ldots, \text{attn}) \]
\[ \text{values (} \ldots, \ldots, \ldots \text{)} \]

or \[ \text{INSERT INTO } <\text{relation}> \text{ <query expression>} \]

Examples:
\[ \text{INSERT INTO account VALUES (‘Perry’, A-768, 1200)} \]

or \[ \text{INSERT INTO account( bname, acct_no, balance)} \]
\[ \text{VALUES (‘Perry’, A-768, 1200)} \]

\begin{verbatim}
INSERT INTO account
  SELECT bname, lno, 200
  FROM loan
  WHERE bname = ‘Kenmore’
\end{verbatim}

gives free $200 savings account for each loan holder at Kenmore
SQL: Modification Commands

Update: 
UPDATE <relation>
SET <attribute> = <expression>
WHERE <predicate>

Ex. 
UPDATE account
SET balance = balance * 1.06
WHERE balance > 10000

UPDATE account
SET balance = balance * 1.05
WHERE balance <= 10000

Alternative:  
UPDATE account
SET balance =
(CASE
    WHEN balance <= 10000 THEN balance*1.05
    ELSE balance*1.06
END)
Embedded 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 lang simplicity
Allow SQL to be embedded in regular programming languages.

- The SQL standard defines embeddings of SQL in a variety of programming languages such as Pascal, PL/I, Fortran, C, and Cobol…. Java and C++.
A language to which SQL queries are embedded is referred to as a *host* language, and the SQL structures permitted in the host language comprise *embedded* SQL.

EXEC SQL statement is used to identify embedded SQL request to the preprocessor

```sql
EXEC SQL <embedded SQL statement > END-EXEC
```

Note: this varies by language. E.g. the Java embedding uses

```java
# SQL { .... } ;
```

Problem:

- SQL relations are (multi-)sets, no *a priori* bound on the number of records.
  
  No such data structure in C or Java.

- SQL supports a mechanism called a *cursor* to handle this.
Example Query

From within a host language, find the names and cities of customers with more than the variable $amount$ dollars in some account.

- Specify the query in SQL and declare a cursor for it

EXEC SQL

```
declare c cursor for
select customer-name, customer-city
from depositor, customer, account
where depositor.customer-name = customer.customer-name
    and depositor.account-number = account.account-number
    and account.balance > :amount
```

END-EXEC
The `open` statement causes the query to be evaluated

```
EXEC SQL open c END-EXEC
```

The `fetch` statement causes the values of one tuple in the query result to be placed on host language variables.

```
EXEC SQL fetch c into :cn, :cc END-EXEC
```

Repeated calls to `fetch` get successive tuples in the query result.

A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to ‘02000’ to indicate no more data is available.

The `close` statement causes the database system to delete the temporary relation that holds the result of the query.

```
EXEC SQL close c END-EXEC
```

Note: above details vary with language. E.g. the Java embedding defines Java iterators to step through result tuples.
Every fetch call, will get the values of the current tuple and will advance the pointer.

A while loop to get all the tuples.

Also, you can move up/down, go to the start, go to end, etc..

Finally, you can update/modify a tuple through a cursor.
Updates Through Cursors

Can update tuples fetched by cursor by declaring that the cursor is for update

```sql
declare c cursor for
select *
from account
where branch-name = 'Kenmore'
for update
```

To update tuple at the current location of cursor

```sql
update account
set balance = balance + 100
where current of c
```
Open Database Connectivity (ODBC) standard

- Standard for application program to communicate with a database server.
- Application program interface (API) to:
  - Open a connection with a database,
  - Send queries and updates,
  - Get back results.

Applications such as GUI, spreadsheets, etc. can use ODBC.
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 theory) 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, Informix, 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 JDB/C/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 ODBC drivers!
- JDBC tutorial online
A **Connection** is an object representing a login to a database

```java
// GET CONNECTION
Connection con;
try {
    con = DriverManager.getConnection("jdbc:odbc:bankDB",
            userName, password);
} catch(Exception e) { System.out.println(e); }
```

Eventually you close the connection

```java
// CLOSE CONNECTION
try { con.close(); }
catch (Exception e) { System.out.println(e); }
```
You need a Statement object for each SQL statement

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

Soon we’ ll say stmt.executeQuery(“select …”);
JDBC Basics: ResultSet

- A ResultSet object serves as a cursor for the statement’s results (stmt.executeQuery())
  ```java
  // EXECUTE QUERY
  ResultSet results;
  try {
    results = stmt.executeQuery("select * from branch");
  } 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()
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 from this statement

- `<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
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() && rowcount < 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");
    System.out.println(buf);
    rowcount++;
}
```

- Similarly, getFloat, getInt, etc.
Update fields in current of cursor:

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

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!
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(), i;
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("\n");
}
results.close(); stmt.close(); con.close();
```
Similar deal for web scripting langs

- 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, etc.
  - most of these are similar, look a lot like jdbc with HTML mixed in
<?php
    $conn = pg_pconnect("dbname=cowbook user=jmh\n    password=secret");

    if (!$conn) {
        echo "An error occured.\n";
        exit;
    }

    $result = pg_query ($conn, "SELECT * FROM Sailors");
    if (!$result) {
        echo "An error occured.\n";  exit;
    }

    $num = pg_num_rows($result);
    for ($i=0; $i < $num; $i++) {
        $r = pg_fetch_row($result, $i);
        for ($j=0; $j < count($r); $j++) {
            echo "$r[$j] \n";
        }
        echo "<BR>";
    }
?>
Stored Procedures

Sometimes better to execute parts of application inside the database system
• more efficient, minimize the amount of data transferred
• can be reused by other users

Stored Procedure: a program that uses a single SQL statement and executed at the database server

CREATE PROCEDURE ShowNumberOfOrders
    SELECT C.cid, C.cname, COUNT(*)
    FROM Customers C, Orders O
    WHERE C.cid = O.cid
    GROUP BY C.cid, C.cname

Can have parameters:
Types: IN, OUT, INOUT
Stored procedures

You can call it from your application. E.g. in JDBC:

```java
CallableStatement cstmr = con.prepareCall("{call ShowNumberOfOrders}");
ResultSet rs = cstmr.executeQuery();
while (rs.next())
    ...
```