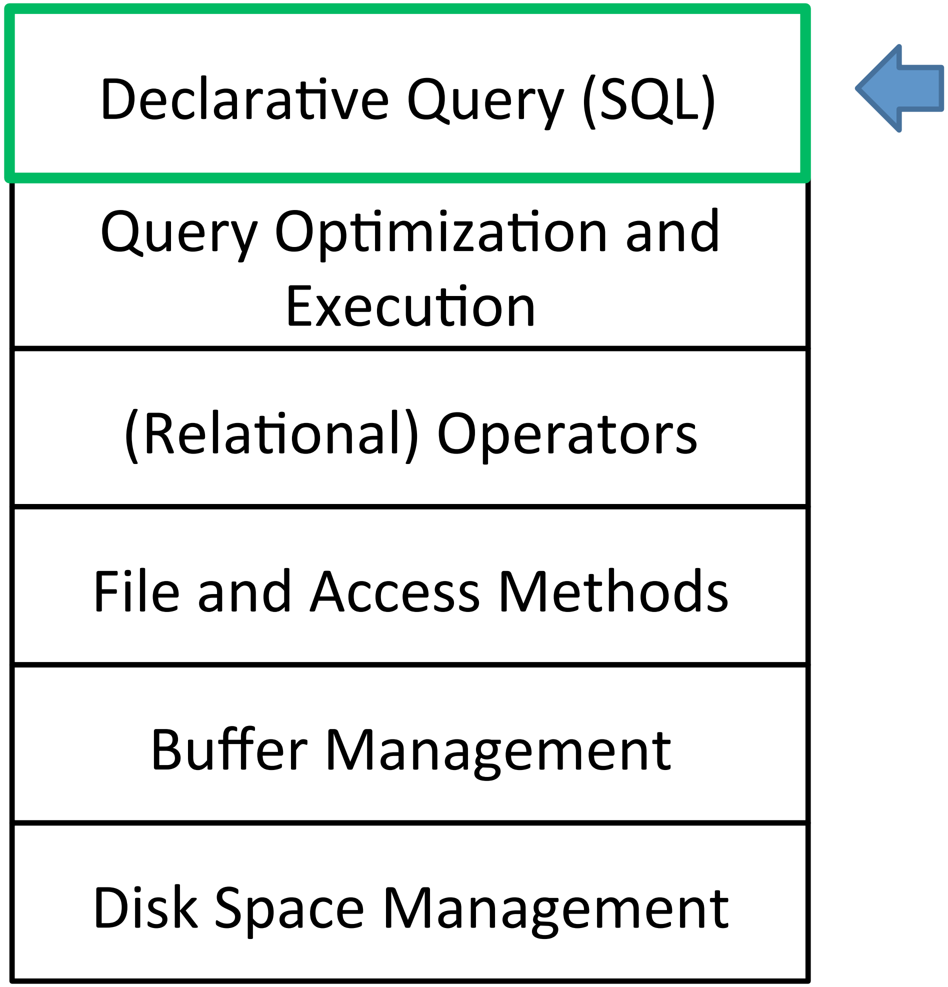
SQL III

The Query Language

R & G - Chapter 5

Based on Slides from UC Berkeley and book.

Query Execution



Declarative Query (SQL)

Query Optimization and Execution

(Relational) Operators

File and Access Methods

Buffer Management

Disk Space Management

We start from here

NULL Values: Truth table

p	***q***	***p* OR *q***	***p* AND *q***	***p* = *q***
TRUE	**TRUE**	TRUE	TRUE	TRUE
TRUE	**FALSE**	TRUE	FALSE	FALSE
TRUE	**Unknown**	TRUE	Unknown	Unknown
FALSE	**TRUE**	TRUE	FALSE	FALSE
FALSE	**FALSE**	FALSE	FALSE	TRUE
FALSE	**Unknown**	Unknown	FALSE	Unknown
Unknown	**TRUE**	TRUE	Unknown	Unknown
Unknown	**FALSE**	Unknown	FALSE	Unknown
Unknown	**Unknown**	Unknown	Unknown	Unknown

NULLs

Given:

branch2=

bname	bcity	assets
Downtown	Boston	9M
Perry	Horse	1.7M
Mianus	Horse	.4M
Kenmore	Boston	NULL

Aggregate operations:

```
SELECT SUM(assets)
FROM     branch2
```

returns

```
SUM
--------
11.1M
```

NULL is ignored
Same for AVG, MIN, MAX

But.... COUNT(assets) retunrs 4!

Let branch3 an empty relation

Then: SELECT SUM(assets)
FROM branch3 returns NULL

but COUNT(<empty rel>) = 0

Joins

```
SELECT (column_list)
FROM   table_name
 [INNER | NATURAL | {LEFT | RIGHT | FULL} | {OUTER}]
JOIN table_name
    ON qualification_list
WHERE …
```

- INNER is default

SELECT sname FROM sailors S JOIN reserves R ON S.sid=R.sid;

SELECT sname FROM sailors S NATURAL JOIN reserves R

WHERE R.bid = 102;

Inner Joins

```
SELECT s.sid, s.sname, r.bid
  FROM Sailors s, Reserves r
 WHERE s.sid = r.sid

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

Both are equivalent!

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)

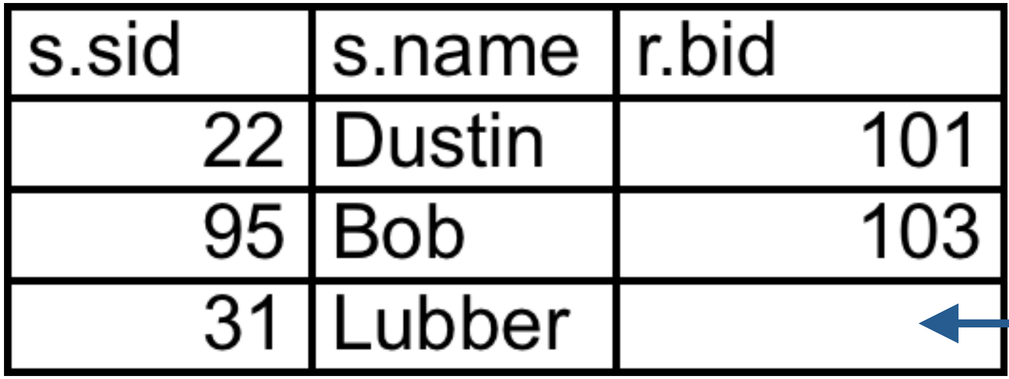
```
SELECT s.sid, s.sname, r.bid
  FROM Sailors s LEFT OUTER JOIN
       Reserves r
        ON s.sid = r.sid;
```

- Returns all sailors & bid for boat in any of their reservations
  - Note: no match for s.sid? r.sid `IS NULL`!

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

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
95	Bob	3	63.5

sid	bid	day
22	101	10/10/96
95	103	11/12/96



s.sid	s.name	r.bid
22	Dustin	101
95	Bob	103
31	Lubber	

NULL

Right Outer Join

- Returns all matched rows, plus all unmatched rows from the table on the **right** of the join clause
  - (use nulls in fields of non-matching tuples)

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

- Returns all boats & information on which ones are reserved
  - Note: no match for b.bid? r.bid `IS NULL`!

Full Outer Join

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

```
SELECT r.sid, b.bid, b.bname
   FROM Reserves2 r FULL OUTER JOIN
        Boats2 b
         ON r.bid = b.bid;
```

- Returns all boats & all information on reservations
- No match for r.bid?
  - b.bid IS NULL AND b.bname is NULL
- No match for b.bid?
  - r.sid is NULL

Constraints (revisited)

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 ))
```

Number of boats plus number of sailors is < 100

Constraints Over Multiple Relations

- Awkward and wrong!
  - Only checks sailors!
- ASSERTION is the right solution; not associated with either table.
  - Unfortunately, not supported in many DBMS.
  - Triggers are another solution.

```
CREATE TABLE Sailors
        ( sid     INTEGER,
          sname   CHAR(10),
          rating  INTEGER,
          age     REAL,
          PRIMARY KEY  (sid),
        )
```

Number of boats plus number of sailors is < 100

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

Views

Views: Named Queries

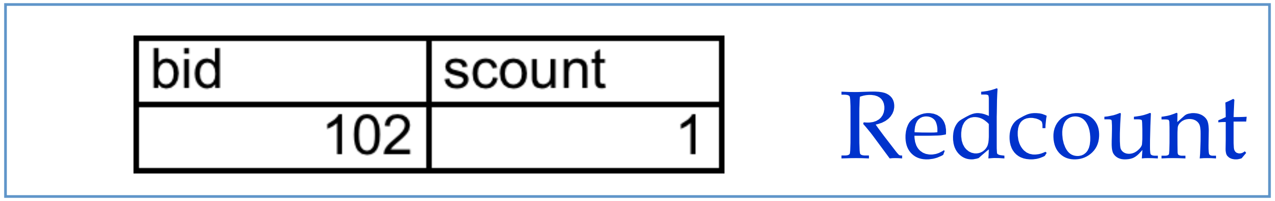
-
  ```
  CREATE VIEW view_name
      AS select_statement
  ```

- Makes development simpler
- Often used for security
- Not “materialized”

```
CREATE VIEW Redcount
AS SELECT b.bid, COUNT(*) AS scount
     FROM Boats b, Reserves2 r
    WHERE r.bid = b.bid AND b.color = 'red'
 GROUP BY b.bid
```

Views Instead of Relations in Queries

```
CREATE VIEW Redcount
AS SELECT b.bid, COUNT(*) AS scount
     FROM Boats b, Reserves2 r
    WHERE r.bid = b.bid AND b.color = 'red'
 GROUP BY b.bid
```



bid	scount
102	1

Redcount

```
SELECT bname, scount
  FROM Redcount r, Boats2 b
 WHERE r.bid = b.bid AND scount < 10
```

Views

create view vs INTO

```
(1)   SELECT bname, bcity
      FROM    branch
      INTO    branch2
```

vs

```
(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.

Subqueries in FROM

- Like a “view create on the fly”

```
SELECT  bname, scount
  FROM  Boats2 b,
        (SELECT b.bid, COUNT(*)
           FROM Boats b, Reserves2 r
          WHERE r.bid=b.bid AND b.color='red'
       GROUP BY b.bid) AS Reds(bid, scount)
 WHERE Reds.bid=b.bid AND scount < 10
```

Common Table Expressions: WITH

- Another “view creation on the fly” syntax

```
WITH Reds(bid, scount) AS
        (SELECT b.bid, COUNT(*)
           FROM Boats b, Reserves2 r
          WHERE r.bid=b.bid AND b.color='red'
       GROUP BY b.bid)
SELECT  bname, scount
  FROM  Boads2 b, Reds
WHERE Reds.bid=b.bid AND scount < 10
```

Find the rating for which the average age of sailors is the minimum over all ratings :

```
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)
```

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
```

SQL: Modification Commands

View Updates:

Suppose we have a view:

```
   CREATE VIEW branch-loan AS
            SELECT bname, lno
            FROM    loan
```

And we insert: INSERT INTO branch-loan VALUES(“Perry”, L-308)

Then, the system will insert a new tuple (“Perry”, L-308, NULL) into loan

SQL: Modification Commands

What about...

```
CREATE VIEW depos-account AS
       SELECT cname, bname, balance
        FROM    depositor as d, account as a
        WHERE  d.acct_no = a.acct_no
```

```
INSERT INTO depos-account VALUES( “Smith”, “Perry”, 500)
```

How many relations we need to update?

Many systems disallow

Discretionary Access Control

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

- Object can be a Database, 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 R&G Chapter 17 for more details.

Embedded SQL

Writing Applications with SQL

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

- Options:
  - Make the query language "Turing complete"
    - Avoids the "impedance mismatch"
    - makes "simple" relational language complex
  - Allow SQL to be embedded in regular programming languages.

Cursors

- Can declare a cursor on a relation or query
- Can open a cursor
- Can repeatedly fetch a tuple (moving the cursor)
- Special return value when all tuples have been retrieved.
- ORDER BY allows control over the order tuples are returned.
  - Fields in ORDER BY clause must also appear in SELECT clause.
- LIMIT controls the number of rows returned (good fit w/ ORDER BY)
- Can also modify/delete tuple pointed to by a cursor
  - A “non-relational” way to get a handle to a particular tuple

Database APIs

- A library with database calls (API)
  - special objects/methods
  - 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. in Python there's the "psycopg2" driver
- ODBC/JDBC try to be DBMS-neutral
  - at least try to hide distinctions across different DBMSs

Summary

- Relational model has well-defined query semantics

- SQL provides functionality close to basic relational model
  - (some differences in duplicate handling, null values, set operators, …)

- Typically, many ways to write a query
  - DBMS figures out a fast way to execute a query, regardless of how it is written.

Triggers (Active database)

- Trigger: A procedure that starts automatically if specified changes occur to the DBMS
- Analog to a "daemon" that monitors a database for certain events to occur
- Three parts:
  - Event (activates the trigger)
  - Condition (tests whether the triggers should run) [Optional]
  - Action (what happens if the trigger runs)
- Semantics:
  - When event occurs, and condition is satisfied, the action is performed.

Triggers – Event,Condition,Action

- Events could be :

```
BEFORE|AFTER INSERT|UPDATE|DELETE ON <tableName>
```

e.g.: `BEFORE INSERT ON Professor`

- Condition is SQL expression or even an SQL query (query with non-empty result means TRUE)

- Action can be many different choices :
  - SQL statements , body of PSM, and even DDL and transaction-oriented statements like "commit".

Example Trigger

Assume our DB has a relation schema :

Professor (pNum, pName, salary)

We want to write a trigger that :

Ensures that any new professor inserted has salary >= 60000

Example Trigger

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor

     for what context  ?

BEGIN

     check for violation here ?


END;
```

Example Trigger

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor

     FOR EACH ROW

BEGIN

    Violation of Minimum Professor Salary?

END;
```

Example Trigger

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor

     FOR EACH ROW

BEGIN

   IF (:new.salary < 60000)
    THEN RAISE_APPLICATION_ERROR (-20004,         ‘Violation
   of Minimum Professor Salary’);
   END IF;

END;
```

Example trigger

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor
   FOR EACH ROW

DECLARE temp int;     -- dummy variable not needed

BEGIN
   IF (:new.salary < 60000)
    THEN RAISE_APPLICATION_ERROR (-20004,         ‘Violation
   of Minimum Professor Salary’);
   END IF;

temp := 10;        -- to illustrate declared variables

END;
.
run;
```

Details of Trigger Example

- BEFORE INSERT ON Professor
  - This trigger is checked before the tuple is inserted
- FOR EACH ROW
  - specifies that trigger is performed for each row inserted
- :new
  - refers to the new tuple inserted
- If (:new.salary < 60000)
  - then an application error is raised and hence the row is not inserted; otherwise the row is inserted.
- Use error code: -20004;
  - this is in the valid range

Example Trigger Using Condition

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor
FOR EACH ROW
WHEN (new.salary < 60000)
BEGIN
   RAISE_APPLICATION_ERROR (-20004,       ‘Violation of
   Minimum Professor Salary’);
END;
.
run;
```

- Conditions can refer to old/new values of tuples modified by the statement activating the trigger.

Triggers: REFERENCING

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor
REFERENCING NEW as newTuple
FOR EACH ROW
WHEN (newTuple.salary < 60000)
BEGIN
  RAISE_APPLICATION_ERROR (-20004,         ‘Violation
  of Minimum Professor Salary’);
END;
.
run;
```

Example Trigger

```
CREATE TRIGGER minSalary
     BEFORE UPDATE ON Professor
REFERENCING OLD AS oldTuple NEW as newTuple
FOR EACH ROW
WHEN (newTuple.salary < oldTuple.salary)
BEGIN
  RAISE_APPLICATION_ERROR (-20004, ‘Salary
  Decreasing !!’);
END;
.
run;
```

- Ensure that salary does not decrease

Triggers (Active database)

- Trigger: A procedure that starts automatically if specified changes occur to the DBMS

- Analog to a "daemon" that monitors a database for certain events to occur

- Three parts:
  - Event (activates the trigger)
  - Condition (tests whether the triggers should run) [Optional]
  - Action (what happens if the trigger runs)

- Semantics:
  - When event occurs, and condition is satisfied, the action is performed.

Another Trigger Example (SQL:99)

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

Row vs Statement Level Trigger

- Row level: activated once per modified tuple
- Statement level: activate once per SQL statement

- Row level triggers can access new data, statement level triggers cannot always do that (depends on DBMS).

- Statement level triggers will be more efficient if we do not need to make row-specific decisions

When to use BEFORE/AFTER

- Based on efficiency considerations or semantics.

- Suppose we perform statement-level after insert,
then all the rows are inserted first,
then if the condition fails,
and all the inserted rows must be "rolled back"

- Not very efficient !!

Combining multiple events into one trigger

```
CREATE TRIGGER salaryRestrictions
AFTER INSERT OR UPDATE ON Professor
FOR EACH ROW
BEGIN
IF (INSERTING AND :new.salary < 60000) THEN
  RAISE_APPLICATION_ERROR (-20004, 'below min
  salary'); END IF;
IF (UPDATING AND :new.salary < :old.salary) THEN
  RAISE_APPLICATION_ERROR (-20004, ‘Salary
  Decreasing !!'); END IF;
END;
```

Summary : Trigger Syntax

```
CREATE TRIGGER <triggerName>
BEFORE|AFTER   INSERT|DELETE|UPDATE
   [OF <columnList>] ON <tableName>|<viewName>
   [REFERENCING [OLD AS <oldName>] [NEW AS <newName>]]
[FOR EACH ROW] (default is “FOR EACH STATEMENT”)
[WHEN (<condition>)]
<PSM body>;
```

MySQL Triggers

```
mysql> delimiter //
mysql> CREATE TRIGGER upd_check BEFORE UPDATE ON
account
   -> FOR EACH ROW
    -> BEGIN
    ->     IF NEW.amount < 0 THEN
    ->         SET NEW.amount = 0;
    ->     ELSEIF NEW.amount > 100 THEN
    ->         SET NEW.amount = 100;
    ->     END IF;
    -> END;//
mysql> delimiter ;
```

```
CREATE TABLE employees_audit (
    id INT AUTO_INCREMENT PRIMARY KEY,
    employeeNumber INT NOT NULL,
    lastname VARCHAR(50) NOT NULL,
    changedat DATETIME DEFAULT NULL,
    action VARCHAR(50) DEFAULT NULL
);
```

```
DELIMITER $$
CREATE TRIGGER before_employee_update
    BEFORE UPDATE ON employees
    FOR EACH ROW
BEGIN
    INSERT INTO employees_audit
    SET action = 'update',
     employeeNumber = OLD.employeeNumber,
        lastname = OLD.lastname,
        changedat = NOW();
END$$
DELIMITER ;
```

Constraints versus Triggers

- Constraints **are useful for database consistency**
  - Use IC when sufficient
  - More opportunity for optimization
  - Not restricted into insert/delete/update

- Triggers are flexible and powerful
  - Alerters
  - Event logging for auditing
  - Security enforcement
  - Analysis of table accesses (statistics)
  - Workflow and business intelligence …

- But can be hard to understand ……
  - Several triggers   (Arbitrary order → unpredictable !?)
  - Chain triggers   (When to stop ?)
  - Recursive triggers (Termination?)

Database Application Development

Example Query

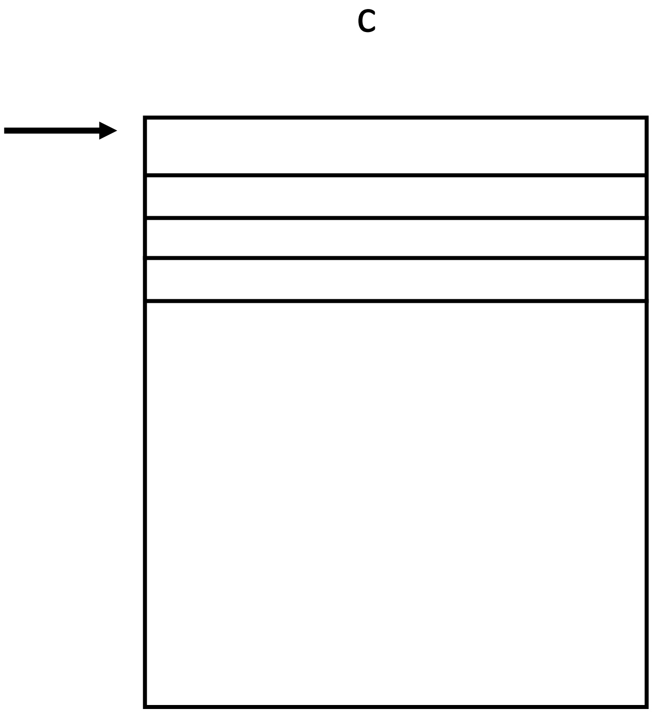
From within a host language, find the names and cities of customers with more than the variable *amount* dollars in some account.

- 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
```

Cursor

EXEC SQL **open** *c* END-EXEC



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

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
  - http://developer.java.sun.com/developer/Books/JDBCTutorial/

JDBC Basics: Connections

- A Connection is an object representing a login to a database

```
// 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

```
// CLOSE CONNECTION
try { con.close(); }
catch (Exception e) { System.out.println(e); }
```

JDBC Basics: Statements

- 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())

```
// 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

```
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

- getStrin

```
// break it off at 100 rows ma

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.

Updating Current of Cursor

- Update fields in current of cursor:

```
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!

Cleaning up Neatly

```
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)

```
Connection con =
   DriverManager.getConnection("jdbc:odbc:weblog",userName,passwor
   d);
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

E.g. PHP/Postgres

```
<?php   $conn = pg_pconnect("dbname=cowbook user=jmh\
                             password=secret");
  if (!$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]&nbsp;";
    }
    echo "<BR>";
  }
?>
```