Views

Views

- A *view* is a relation defined in terms of stored tables (called *base tables*) and other views
- Two kinds:

  1. *Virtual* = not stored in the database; just a query for constructing the relation
  2. *Materialized* = actually constructed and stored

Declaring Views

- Declare by:

  CREATE [MATERIALIZED] VIEW
  <name> AS <query>;
- Default is virtual
- PostgreSQL has no direct support for materialized views

Materialized Views

- Problem: each time a base table changes, the materialized view may change
  - Cannot afford to recompute the view with each change
- Solution: Periodic reconstruction of the materialized view, which is otherwise "out of date"

Example: A Data Warehouse

- Bilka stores every sale at every store in a database
- Overnight, the sales for the day are used to update a *data warehouse* = materialized views of the sales
- The warehouse is used by analysts to predict trends and move goods to where they are selling best

Virtual Views

- only a query is stored
- no need to change the view when the base table changes
- expensive when accessing the view often

Example: View Definition

- CanDrink(drinker, beer) is a view "containing" the drinker-beer pairs such that the drinker frequents at least one bar that serves the beer:

```
CREATE VIEW CanDrink AS
    SELECT drinker, beer
    FROM Frequents, Sells
    WHERE Frequents.bar = Sells.bar;
```

Example: View Definition

- CanDrink(drinker, beer) is a view "containing" the drinker-beer pairs such that the drinker frequents at least one bar that serves the beer:

```
CREATE VIEW CanDrink AS
    SELECT drinker, beer
    FROM Frequents NATURAL JOIN Sells;
```

Example: View Definition

- CanDrink(drinker, beer) is a view "containing" the drinker-beer pairs such that the drinker frequents at least one bar that serves the beer:

```
CREATE TABLE CanDrink
    (drinker TEXT, beer TEXT);
CREATE RULE "_RETURN" AS ON SELECT
    TO CanDrink DO INSTEAD
    SELECT drinker, beer
    FROM Frequents NATURAL JOIN Sells;
```

Example: Accessing a View

- Query a view as if it were a base table
- Example query:

```
SELECT beer FROM CanDrink
WHERE drinker = 'Peter';
```

- The *rule* "_RETURN" will rewrite this to:

```
SELECT beer FROM (SELECT
drinker, beer FROM Frequents
NATURAL JOIN Sells) AS CanDrink
where drinker = 'Peter';
```

Modifying Virtual Views

- Generally, it is impossible to modify a virtual view, because it does not exist
- But a *rule* lets us interpret view modifications in a way that makes sense
- Example: the view Synergy has (drinker, beer, bar) triples such that the bar serves the beer, the drinker frequents the bar and likes the beer

Example: The View

Pick one copy of each attribute

```
CREATE VIEW Synergy AS
  SELECT Likes.drinker, Likes.beer, Sells.bar
  FROM Likes, Sells, Frequents
  WHERE Likes.drinker = Frequents.drinker
     AND Likes.beer = Sells.beer
     AND Sells.bar = Frequents.bar;
```

Natural join of Likes, Sells, and Frequents

Example: The View

```
CREATE VIEW Synergy AS
   SELECT drinker, beer, bar
   FROM Likes NATURAL JOIN Sells NATURAL
   JOIN Frequents;
```

Interpreting a View Insertion

- We cannot insert into Synergy – it is a virtual view
- But we can use a rule to turn a (drinker, beer, bar) triple into three insertions of projected pairs, one for each of Likes, Sells, and Frequents
  - Sells.price will have to be NULL

The Rule

```
CREATE RULE ViewRule AS
     ON INSERT TO Synergy
          DO INSTEAD (
     INSERT INTO Likes VALUES
     (NEW.drinker, NEW.beer);
     INSERT INTO Sells(bar, beer) VALUES
     (NEW.bar, NEW.beer);
     INSERT INTO Frequents VALUES
     (NEW.drinker, NEW.bar);
   );
```

Example: Assertion

```
CREATE FUNCTION CheckNumbers()
  RETURNS TRIGGER AS $$BEGIN IF
   (SELECT COUNT(*) FROM Bars) >
   (SELECT COUNT(*) FROM Drinkers)
  THEN RAISE EXCEPTION '2manybars';
  END IF; RETURN NEW; END$$
  LANGUAGE plpgsql;

CREATE TRIGGER NumberBars AFTER
  INSERT ON Bars EXECUTE PROCEDURE
  CheckNumbers();

CREATE TRIGGER NumberDrinkers AFTER
  DELETE ON Drinkers EXECUTE PROCEDURE
  CheckNumbers();
```

Example: Assertion

```
CREATE FUNCTION CheckNumbers()
  RETURNS TRIGGER AS $$BEGIN IF
   (SELECT COUNT(*) FROM Bars) >
   (SELECT COUNT(*) FROM Drinkers)
  THEN RETURN NULL;
  END IF; RETURN NEW; END$$
  LANGUAGE plpgsql;
CREATE TRIGGER NumberBars AFTER
  INSERT ON Bars EXECUTE PROCEDURE
  CheckNumbers();
CREATE TRIGGER NumberDrinkers AFTER
  DELETE ON Drinkers EXECUTE PROCEDURE
  CheckNumbers();
```

Example: Assertion

```
CREATE RULE CheckBars AS
     ON INSERT TO Bars
   WHEN (SELECT COUNT(*) FROM Bars) >=
   (SELECT COUNT(*) FROM Drinkers)
     DO INSTEAD NOTHING;


CREATE RULE CheckDrinkers AS
     ON DELETE TO Drinkers
   WHEN (SELECT COUNT(*) FROM Bars) >=
   (SELECT COUNT(*) FROM Drinkers)
     DO INSTEAD NOTHING;
```

Transactions

Why Transactions?

- Database systems are normally being accessed by many users or processes at the same time
  - Both queries and modifications
- Unlike operating systems, which *support* interaction of processes, a DMBS needs to keep processes from troublesome interactions

Example: Bad Interaction

- You and your domestic partner each take $100 from different ATM's at about the same time
  - The DBMS better make sure one account deduction does not get lost
- Compare: An OS allows two people to edit a document at the same time; If both write, one's changes get lost

Transactions

- *Transaction* = process involving database queries and/or modification
- Normally with some strong properties regarding concurrency
- Formed in SQL from single statements or explicit programmer control

ACID Transactions

- *ACID transactions* are:
  - *Atomic:* Whole transaction or none is done
  - *Consistent:* Database constraints preserved
  - *Isolated:* It appears to the user as if only one process executes at a time
  - *Durable:* Effects of a process survive a crash
- Optional: weaker forms of transactions are often supported as well

COMMIT

- The SQL statement COMMIT causes a transaction to complete
  - database modifications are now permanent in the database

ROLLBACK

- The SQL statement ROLLBACK also causes the transaction to end, but by *aborting*
  - No effects on the database
- Failures like division by 0 or a constraint violation can also cause rollback, even if the programmer does not request it

Example: Interacting Processes

- Assume the usual Sells(bar,beer,price) relation, and suppose that C.Ch. sells only Od.Cl. for 20 and Er.We. for 30
- Peter is querying Sells for the highest and lowest price C.Ch. Charges
- C.Ch. decides to stop selling Od.Cl. And Er.We., but to sell only Tuborg at 35

Peter's Program

- Peter executes the following two SQL statements called (min) and (max) to help us remember what they do

(max) SELECT MAX(price) FROM Sells
WHERE bar = 'C.Ch.';

(min) SELECT MIN(price) FROM Sells
WHERE bar = 'C.Ch.';

Cafe Chino's Program

- At about the same time, C.Ch. executes the following steps: (del) and (ins)

(del) DELETE FROM Sells
WHERE bar = 'C.Ch.';

(ins) INSERT INTO Sells
VALUES('C.Ch.', 'Tuborg', 35);

Interleaving of Statements

- Although (max) must come before (min), and (del) must come before (ins), there are no other constraints on the order of these statements, unless we group Peter's and/or Cafe Chino's statements into transactions

Example: Strange Interleaving

- Suppose the steps execute in the order (max)(del)(ins)(min)

C.Ch. Prices:	{20, 30}	{20,30}		{35}
Statement:	(max)	(del)	(ins)	(min)
Result:	30			35

- Peter sees MAX < MIN!

Fixing the Problem

- If we group Peter's statements (max) (min) into one transaction, then he cannot see this inconsistency
- He sees C.Ch.'s prices at some fixed time
  - Either before or after they changes prices, or in the middle, but the MAX and MIN are computed from the same prices

Another Problem: Rollback

- Suppose C.Ch. executes (del)(ins), not as a transaction, but after executing these statements, thinks better of it and issues a ROLLBACK statement
- If Peter executes his statements after (ins) but before the rollback, he sees a value, 35, that never existed in the database

Solution

- If Cafe Chino executes (del)(ins) as a transaction, its effect cannot be seen by others until the transaction executes COMMIT
  - If the transaction executes ROLLBACK instead, then its effects can *never* be seen

Isolation Levels

- SQL defines four *isolation levels* = choices about what interactions are allowed by transactions that execute at about the same time
- Only one level ("serializable") = ACID transactions
- Each DBMS implements transactions in its own way

Choosing the Isolation Level

- Within a transaction, we can say:

SET TRANSACTION ISOLATION LEVEL *X*

where *X* =

1. SERIALIZABLE
2. REPEATABLE READ
3. READ COMMITTED
4. READ UNCOMMITTED

Serializable Transactions

- If Peter = (max)(min) and C.Ch. = (del)(ins) are each transactions, and Peter runs with isolation level SERIALIZABLE, then he will see the database either before or after C.Ch. runs, but not in the middle

Isolation Level Is Personal Choice

- Your choice, e.g., run serializable, affects only how *you* see the database, not how others see it
- Example: If Cafe Chino Runs serializable, but Peter does not, then Peter might see no prices for Cafe Chino
  - i.e., it looks to Peter as if he ran in the middle of Cafe Chino's transaction

Read-Commited Transactions

- If Peter runs with isolation level READ COMMITTED, then he can see only committed data, but not necessarily the same data each time.
- Example: Under READ COMMITTED, the interleaving (max)(del)(ins)(min) is allowed, as long as Cafe Chino commits
  - Peter sees MAX < MIN

Repeatable-Read Transactions

- Requirement is like read-committed, plus: if data is read again, then everything seen the first time will be seen the second time
  - But the second and subsequent reads may see *more* tuples as well

Example: Repeatable Read

- Suppose Peter runs under REPEATABLE READ, and the order of execution is (max)(del)(ins)(min)
  - (max) sees prices 20 and 30
  - (min) can see 35, but must also see 20 and 30, because they were seen on the earlier read by (max)

Read Uncommitted

- A transaction running under READ UNCOMMITTED can see data in the database, even if it was written by a transaction that has not committed (and may never)
- Example: If Peter runs under READ UNCOMMITTED, he could see a price 35 even if Cafe Chino later aborts

Indexes

Indexes

- *Index* = data structure used to speed access to tuples of a relation, given values of one or more attributes
- Could be a hash table, but in a DBMS it is always a balanced search tree with giant nodes (a full disk page) called a *B-tree*

Declaring Indexes

- No standard!
- Typical syntax (also PostgreSQL):

```
CREATE INDEX BeerInd ON
  Beers(manf);
CREATE INDEX SellInd ON
  Sells(bar, beer);
```

Using Indexes

- Given a value *v*, the index takes us to only those tuples that have *v* in the attribute(s) of the index
- Example: use BeerInd and SellInd to find the prices of beers manufactured by Albani and sold by Cafe Chino (next slide)

Using Indexes

```
SELECT price FROM Beers, Sells
WHERE manf = 'Albani' AND
    Beers.name = Sells.beer AND
    bar = 'C.Ch.';
```

1. Use BeerInd to get all the beers made by Albani
2. Then use SellInd to get prices of those beers, with bar = 'C.Ch.'

Database Tuning

- A major problem in making a database run fast is deciding which indexes to create
- Pro: An index speeds up queries that can use it
- Con: An index slows down all modifications on its relation because the index must be modified too

Example: Tuning

- Suppose the only things we did with our beers database was:
  1. Insert new facts into a relation (10%)
  2. Find the price of a given beer at a given bar (90%)
- Then SellInd on Sells(bar, beer) would be wonderful, but BeerInd on Beers(manf) would be harmful

Tuning Advisors

- A major research area
  - Because hand tuning is so hard
- An advisor gets a *query load*, e.g.:
  1. Choose random queries from the history of queries run on the database, or
  2. Designer provides a sample workload

Tuning Advisors

- The advisor generates candidate indexes and evaluates each on the workload
  - Feed each sample query to the query optimizer, which assumes only this one index is available
  - Measure the improvement/degradation in the average running time of the queries

Summary 7

More things you should know:

- Constraints, Cascading, Assertions
- Triggers, Event-Condition-Action
- Triggers in PostgreSQL, Functions
- Views, Rules
- Transactions

Real SQL Programming

SQL in Real Programs

- We have seen only how SQL is used at the generic query interface – an environment where we sit at a terminal and ask queries of a database
- Reality is almost always different: conventional programs interacting with SQL

Options

1. Code in a specialized language is stored in the database itself (e.g., PSM, PL/pgsql)
2. SQL statements are embedded in a *host language* (e.g., C)
3. Connection tools are used to allow a conventional language to access a database (e.g., CLI, JDBC, psycopg2)

Stored Procedures

- PSM, or "*persistent stored modules*," allows us to store procedures as database schema elements
- PSM = a mixture of conventional statements (if, while, etc.) and SQL
- Lets us do things we cannot do in SQL alone

Procedures in PostgreSQL

```
CREATE PROCEDURE <name>
  ([<arguments>]) AS $$
  <program>$$ LANGUAGE <lang>;
```

- PostgreSQL only supports functions:

```
CREATE FUNCTION <name>
  ([<arguments>]) RETURNS VOID AS $$
  <program>$$ LANGUAGE <lang>;
```

Parameters for Procedures

- Unlike the usual name-type pairs in languages like Java, procedures use mode-name-type triples, where the *mode* can be:
  - IN = function uses value, does not change
  - OUT = function changes, does not use
  - INOUT = both

Example: Stored Procedure

- Let's write a procedure that takes two arguments *b* and *p*, and adds a tuple to Sells(bar, beer, price) that has bar = 'C.Ch.', beer = *b*, and price = *p*
  - Used by Cafe Chino to add to their menu more easily

The Procedure

```
CREATE FUNCTION ChinoMenu (
    IN b      CHAR(20),
    IN p      REAL
) RETURNS VOID AS $$
INSERT INTO Sells
VALUES('C.Ch.', b, p);
$$ LANGUAGE plpgsql;
```

Parameters are both read-only, not changed

The body --- a single insertion

Invoking Procedures

- Use SQL/PSM statement CALL, with the name of the desired procedure and arguments
- Example:

```
CALL ChinoMenu(’Eventyr’, 50);
```

- Functions used in SQL expressions wherever a value of their return type is appropriate
- No CALL in PostgreSQL:

```
SELECT ChinoMenu(’Eventyr’, 50);
```

Kinds of PL/pgsql statements

- Return statement: RETURN <expression> returns value of a function
  - Like in Java, RETURN terminates the function execution
- Declare block: DECLARE <name> <type> used to declare local variables
- Groups of Statements: BEGIN . . . END
  - Separate statements by semicolons

Kinds of PL/pgsql statements

- Assignment statements:

  <variable> := <expression>;

  - Example: `b := 'Od.Cl.';`
- Statement labels: give a statement a label by prefixing a name and a colon

IF Statements

- Simplest form:
  IF <condition> THEN
  <statements(s)>
  END IF;
- Add ELSE <statement(s)> if desired, as
  IF . . . THEN . . . ELSE . . . END IF;
- Add additional cases by ELSEIF <statements(s)>: IF … THEN … ELSEIF … THEN … ELSEIF … THEN … ELSE … END IF;

Example: IF

- Let' s rate bars by how many customers they have, based on Frequents(drinker,bar)
  - <100 customers: 'unpopular'
  - 100-199 customers: 'average'
  - >= 200 customers: 'popular'
- Function Rate(b) rates bar b

Example: IF

```
CREATE FUNCTION Rate (IN b CHAR(20))
      RETURNS CHAR(10) AS $$
      DECLARE cust INTEGER;
   BEGIN
      cust := (SELECT COUNT(*) FROM Frequents
                WHERE bar = b);
      IF cust < 100 THEN RETURN ’unpopular’;
      ELSEIF cust < 200 THEN RETURN ’average’;
      ELSE RETURN ’popular’;
      END IF;
   END;
```

Number of customers of bar b

Nested IF statement

Loops

- Basic form:

```
<<<label>>>        LOOP
        <statements>
END LOOP;
```

- Exit from a loop by:

```
    EXIT <label> WHEN <condition>
```

Example: Exiting a Loop

```
<<loop1>> LOOP
    . . .
    EXIT loop1 WHEN ...;
    . . .
END LOOP;
```

If this statement is executed and the condition holds ...

... control winds up here

Other Loop Forms

- WHILE <condition> LOOP
    <statements>
  END LOOP;
- Equivalent to the following LOOP:
  LOOP
    EXIT WHEN NOT <condition>;
    <statements> END
  LOOP;

Other Loop Forms

- FOR <name> IN <start> TO <end>
  LOOP
      <statements>
  END LOOP;
- Equivalent to the following block:

  <name> := <start>;

  LOOP EXIT WHEN <name> > <end>;
      <statements>
      <name> := <name>+1;   END
  LOOP;

Other Loop Forms

- FOR <name> IN REVERSE <start> TO <end> LOOP

      <statements>

  END LOOP;
- Equivalent to the following block:

  <name> := <start>;

  LOOP EXIT WHEN <name> < <end>;

      <statements>

      <name> := <name> - 1;

  END LOOP;

Other Loop Forms

- FOR <name> IN <start> TO <end> BY <step> LOOP
  <statements>
  END LOOP;
- Equivalent to the following block:

  <name> := <start>;
  LOOP EXIT WHEN <name> > <end>;
  <statements>
  <name> := <name>+<step>;
  END LOOP;

Queries

- General SELECT-FROM-WHERE queries are *not* permitted in PL/pgsql
- There are three ways to get the effect of a query:
  1. Queries producing one value can be the expression in an assignment
  2. Single-row SELECT ... INTO
  3. Cursors

Example: Assignment/Query

- Using local variable *p* and Sells(bar, beer, price), we can get the price Cafe Chino charges for Odense Classic by:

```
p := (SELECT price FROM Sells
   WHERE bar = 'C.Ch' AND
      beer = 'Od.Cl.');
```

SELECT … INTO

- Another way to get the value of a query that returns one tuple is by placing INTO <variable> after the SELECT clause
- Example:

```
SELECT price INTO p FROM Sells
WHERE bar = 'C.Ch.' AND
    beer = 'Od.Cl.';
```

Cursors

- A *cursor* is essentially a tuple-variable that ranges over all tuples in the result of some query
- Declare a cursor *c* by:

DECLARE c CURSOR FOR <query>;

Opening and Closing Cursors

- To use cursor *c*, we must issue the command:

  OPEN c;

  - The query of *c* is evaluated, and *c* is set to point to the first tuple of the result
- When finished with *c*, issue command:

  CLOSE c;

Fetching Tuples From a Cursor

- To get the next tuple from cursor c, issue command:

  FETCH FROM c INTO x_1, x_2,…,x_n ;

- The *x*'s are a list of variables, one for each component of the tuples referred to by *c*
- c is moved automatically to the next tuple

Breaking Cursor Loops – (1)

- The usual way to use a cursor is to create a loop with a FETCH statement, and do something with each tuple fetched
- A tricky point is how we get out of the loop when the cursor has no more tuples to deliver

Breaking Cursor Loops – (2)

- Many operations return if a row has been found, changed, inserted, or deleted (SELECT INTO, UPDATE, INSERT, DELETE, FETCH)
- In plpgsql, we can get the value of the status in a variable called FOUND

Breaking Cursor Loops – (3)

- The structure of a cursor loop is thus:

```
<<cursorLoop>> LOOP
  …
  FETCH c INTO … ;
  IF NOT FOUND THEN EXIT cursorLoop;
  END IF;
  …
END LOOP;
```

Example: Cursor

- Let us write a procedure that examines Sells(bar, beer, price), and raises by 10 the price of all beers at Cafe Chino that are under 30
- Yes, we could write this as a simple UPDATE, but the details are instructive anyway

The Needed Declarations

```
CREATE FUNCTION RaisePrices()
  RETURNS VOID AS $$
  DECLARE theBeer CHAR(20);
          thePrice REAL;
          c CURSOR FOR
    (SELECT beer, price FROM Sells
     WHERE bar = 'C.Ch.');
```

Used to hold beer-price pairs when fetching through cursor c

Returns Cafe Chino's price list

The Procedure Body

```
BEGIN
   OPEN c;
   <<menuLoop>> LOOP
      FETCH c INTO theBeer, thePrice;
      EXIT menuLoop WHEN NOT FOUND;
      IF thePrice < 30 THEN
         UPDATE Sells SET price = thePrice + 10
         WHERE bar = ’C.Ch.’ AND beer = theBeer;
      END IF;
   END LOOP;
   CLOSE c;
END;$$ LANGUAGE plpgsql;
```

Check if the recent FETCH failed to get a tuple

If Cafe Chino charges less than 30 for the beer, raise its price at at Cafe Chino by 10

Tuple-Valued Variables

- PL/pgsql allows a variable *x* to have a tuple type
- x R%ROWTYPE gives *x* the type of R's tuples
- *R* could be either a relation or a cursor
- x.a gives the value of the component for attribute *a* in the tuple *x*

Example: Tuple Type

- Repeat of RaisePrices() declarations with variable *bp* of type beer-price pairs

```
CREATE FUNCTION RaisePrices()
  RETURNS VOID AS $$
  DECLARE CURSOR c IS
  SELECT beer, price FROM Sells
  WHERE bar = 'C.Ch.';
          bp c%ROWTYPE;
```

RaisePrices() Body Using *bp*

```
BEGIN
   OPEN c;
   LOOP
        FETCH c INTO bp;
        EXIT WHEN NOT FOUND;
        IF bp.price < 30 THEN
           UPDATE Sells SET price = bp.price + 10
           WHERE bar = 'C.Ch.' AND beer = bp.beer;
        END IF;
   END LOOP;
   CLOSE c;
END;
```

Components of bp are obtained with a dot and the attribute name

Database-Connection Libraries

Host/SQL Interfaces Via Libraries

- The third approach to connecting databases to conventional languages is to use library calls

1. C + CLI
2. Java + JDBC
3. Python + psycopg2

Three-Tier Architecture

- A common environment for using a database has three tiers of processors:
  1. *Web servers* – talk to the user.
  2. *Application servers* – execute the business logic
  3. *Database servers* – get what the app servers need from the database

Example: Amazon

- Database holds the information about products, customers, etc.
- Business logic includes things like "what do I do after someone clicks 'checkout'?"
  - Answer: Show the "how will you pay for this?" screen

Environments, Connections, Queries

- The database is, in many DB-access languages, an *environment*
- Database servers maintain some number of *connections*, so app servers can ask queries or perform modifications
- The app server issues *statements:* queries and modifications, usually

JDBC

- *Java Database Connectivity* (JDBC) is a library similar for accessing a DBMS with Java as the host language
- >200 drivers available: PostgreSQL, MySQL, Oracle, ODBC, ...
- `http://jdbc.postgresql.org/`

Making a Connection

The JDBC classes

```
import java.sql.*;
...
Class.forName("org.postgresql.Driver");
Connection myCon =
  DriverManager.getConnection(...);
...
```

Loaded by forName

URL of the database your name, and password go here

The driver for postgresql; others exist

URL for PostgreSQL database

- jdbc:postgresql://<host>[:<port>]/<database>?user=<user>&password=<password>
- Alternatively use getConnection variant:
- getConnection(“jdbc:postgresql://<host>[:<port>]/<database>“, <user>, <password>);
- DriverManager.getConnection(“jdbc:postgresql://10.110.4.32:5434/postgres“, “petersk“, “geheim“);

Statements

- JDBC provides two classes:
  1. *Statement* = an object that can accept a string that is a SQL statement and can execute such a string
  2. *PreparedStatement* = an object that has an associated SQL statement ready to execute

Creating Statements

- The Connection class has methods to create Statements and PreparedStatements

```
Statement stat1 = myCon.createStatement();
PreparedStatement stat2 =
  myCon.createStatement(
      ”SELECT beer, price FROM Sells ” +
      ”WHERE bar = ’C.Ch.’ ”
  );
```

createStatement with no argument returns a Statement; with one argument it returns a PreparedStatement

Executing SQL Statements

- JDBC distinguishes queries from modifications, which it calls "updates"
- Statement and PreparedStatement each have methods executeQuery and executeUpdate
  - For Statements: one argument – the query or modification to be executed
  - For PreparedStatements: no argument

Example: Update

- stat1 is a Statement
- We can use it to insert a tuple as:

```
stat1.executeUpdate(
  ”INSERT INTO Sells ” +
  ”VALUES(’C.Ch.’,’Eventyr’,30)”
);
```

Example: Query

- stat2 is a PreparedStatement holding the query ”SELECT beer, price FROM Sells WHERE bar = ’C.Ch.’ ”
- executeQuery returns an object of class ResultSet – we’ll examine it later
- The query:

```
ResultSet menu = stat2.executeQuery();
```

Accessing the ResultSet

- An object of type ResultSet is something like a cursor
- Method next() advances the "cursor" to the next tuple
  - The first time next() is applied, it gets the first tuple
  - If there are no more tuples, next() returns the value false

Accessing Components of Tuples

- When a ResultSet is referring to a tuple, we can get the components of that tuple by applying certain methods to the ResultSet
- Method get*X* (*i*), where *X* is some type, and *i* is the component number, returns the value of that component
  - The value must have type *X*

Example: Accessing Components

- Menu = ResultSet for query “SELECT beer, price FROM Sells WHERE bar = ’C.Ch.’ ”
- Access beer and price from each tuple by:

```
while (menu.next()) {
  theBeer = menu.getString(1);
  thePrice = menu.getFloat(2);
     /*something with theBeer and
       thePrice*/
}
```

Important Details

- Reusing a Statement object results in the ResultSet being closed
  - Always create new Statement objects using createStatement() or explicitly close ResultSets using the close method
- For transactions, for the Connection con use con.setAutoCommit(false) and explicitly con.commit() or con.rollback()
  - If AutoCommit is false and there is no commit, closing the connection = rollback