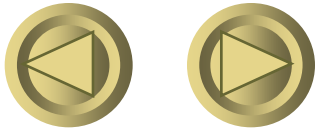
# Chapter 6 The database Language SQL –as a tutorial

- **About SQL**

  **SQL is a standard database language, adopted by many commercial systems.**

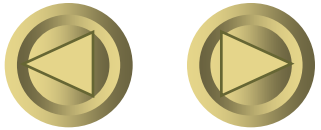
  **ANSI SQL, SQL-92 or SQL2, SQL99 or SQL3 extends SQL2 with object-relational features. SQL2003 is the collection of extensions to SQL3.**

- **How to query the database**
- **How to make modifications on database**
- **Transactions in SQL**



# Union, Intersection and Difference

- **Union, intersection, and difference of relations are expressed by the following forms, each involving subqueries:**
  - **( subquery ) UNION ( subquery )**
  - **( subquery ) INTERSECT ( subquery )**
  - **( subquery ) EXCEPT ( subquery )**

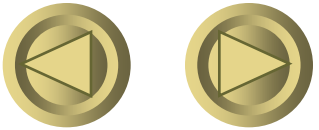


# Example

- **From relations**

  **Likes(drinker, beer), Sells(bar, beer, price) and Frequents(drinker, bar), find the drinkers and beers such that:**

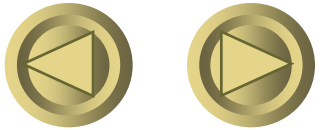
  1. **The drinker likes the beer, and**
  2. **The drinker frequents at least one bar that sells the beer.**



# Solution

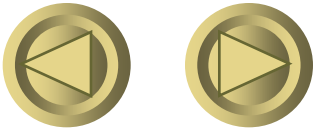
```
(SELECT * FROM Likes)
   INTERSECT
(SELECT drinker, beer
 FROM Sells, Frequents
 WHERE Frequents.bar = Sells.bar
);
```

The drinker frequents a bar that sells the beer.



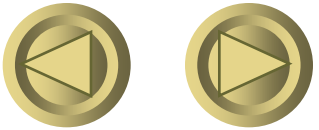
# Bag Semantics

- **Although the SELECT-FROM-WHERE statement uses bag semantics, the default for union, intersection, and difference is set semantics.**
  - **That is, duplicates are eliminated as the operation is applied.**



# Motivation: Efficiency

- **When doing projection in relational algebra, it is easier to avoid eliminating duplicates.**
  - **Just work tuple-at-a-time.**
- **When doing intersection or difference, it is most efficient to sort the relations first.**
  - **At that point you may as well eliminate the duplicates anyway.**



# Controlling Duplicate Elimination

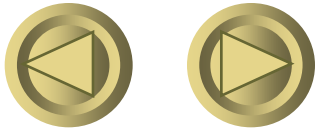
- **Force the result to be a set by SELECT DISTINCT . . .**
- **Force the result to be a bag (i.e., don’t eliminate duplicates) by ALL, as in . . . UNION ALL . . .**


# Example: DISTINCT

- **From Sells(bar, beer, price), find all the different prices charged for beers:**

```
SELECT DISTINCT price
FROM Sells;
```

- **Notice that without DISTINCT, each price would be listed as many times as there were bar/beer pairs at that price.**



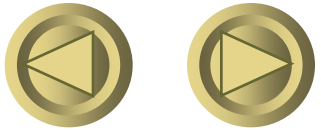
# Example: ALL

## Force bag semantics with ALL

- Using relations Frequents(drinker, bar) and Likes(drinker, beer):

```
(SELECT drinker FROM Frequents)
    EXCEPT ALL
(SELECT drinker FROM Likes);
```

- Lists drinkers who frequent more bars than they like beers, and does so as many times as the difference of those counts.



# Classroom Exercise

**Assume R and S have the same schema**

**Q1: (select * from R) INTERSECT ALL (select * from S)**

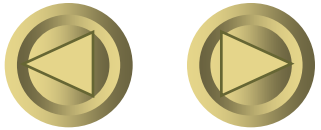
**Q2: (select * from R) NATURAL JOIN (select * from S)**

**a) Q1 and Q2 produce the same answer.**

**b)The answer to Q1 is contained in the answer to Q2**

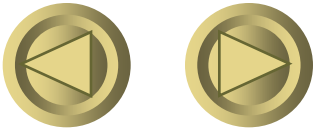
**c) The answer to Q2 is contained in the answer to Q1**

**d) Q1 and Q2 produce different answers.**



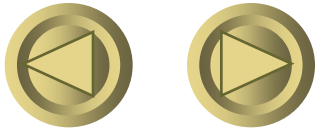
# Answer (b)

**As sets, both produce the intersection of *R* and *S*. However, SQL is a bag language. Suppose a tuple *t* appears *m* times in *R* and *n* times in *S*. Then *T* appears *min(m,n)* times in the answer to Q1. However, for the join, we pair all tuples of *R* with all tuples of *S* that agree in the common attributes (i.e., all attributes in this case), and we produce of copy for each successful pairing. Thus, Q2 produces *mn* copies of *t*. It is easy to verify that as long as *m* and *n* are nonnegative integers, *min(m,n)* <= *mn*.**



# Join Expressions

- **SQL provides a number of expression forms that act like varieties of join in relational algebra.**
  - **But using bag semantics, not set semantics.**
- **These expressions can be stand-alone queries or used in place of relations in a FROM clause.**



# Products and Natural Joins

- **Natural join is obtained by:**

  **R NATURAL JOIN S;**

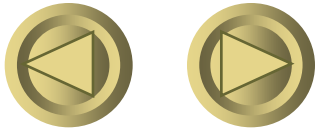
- **Product is obtained by:**

  **R CROSS JOIN S;**

- **Example:**

  ```
  Likes NATURAL JOIN Serves;
  ```

- **Relations can be parenthesized subexpressions, as well.**

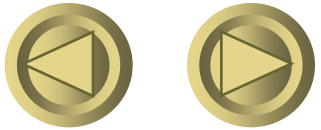


# Theta Join

- **R JOIN S ON <condition> is a theta-join, using <condition> for selection.**
- **Example: using Drinkers(name, addr) and Frequents(drinker, bar):**

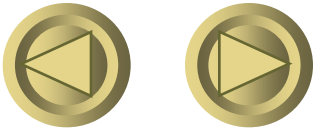
```
Drinkers JOIN Frequents ON
    name = drinker;
```

**gives us all (*d, a, d, b*) quadruples such that drinker *d* lives at address *a* and frequents bar *b*.**



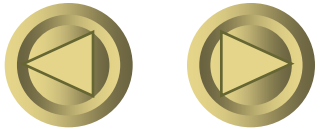
# Outerjoins

- **R OUTER JOIN S is the core of an outerjoin expression. It is modified by:**
  1. **Optional NATURAL in front of OUTER.**
  2. **Optional ON <condition> after JOIN.**
  3. **Optional LEFT, RIGHT, or FULL before OUTER.**
     - **LEFT = pad dangling tuples of R only.**
     - **RIGHT = pad dangling tuples of S only.**
     - **FULL = pad both; this choice is the default.**



# Database Modifications

- **A modification command does not return a result as a query does, but it changes the database in some way.**
- **There are three kinds of modifications:**
  1. ***Insert*** **a tuple or tuples.**
  2. ***Delete*** **a tuple or tuples.**
  3. ***Update*** **the value(s) of an existing tuple or tuples.**



# Insertion

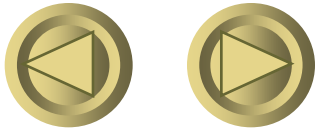
- **To insert a single tuple:**

  **INSERT INTO <relation>**

  **VALUES ( <list of values> );**

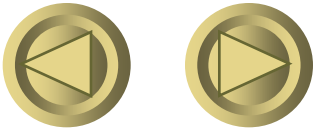
- **Example: add to Likes(drinker, beer) the fact that Sally likes Bud.**

```
INSERT INTO Likes
VALUES('Sally', 'Bud');
```



# Specifying Attributes in INSERT

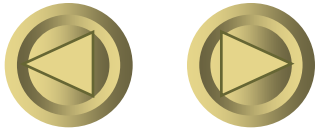
- **We may add to the relation name a list of attributes.**
- **There are two reasons to do so:**
  1. **We forget the standard order of attributes for the relation.**
  2. **We don’t have values for all attributes, and we want the system to fill in missing components with NULL or a default value.**



# Example: Specifying Attributes

- **Another way to add the fact that Sally likes Bud to Likes(drinker, beer):**

```
INSERT INTO Likes(beer,
  drinker)
VALUES('Bud', 'Sally');
```

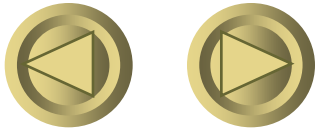


# Inserting Many Tuples

- **We may insert the entire result of a query into a relation, using the form:**

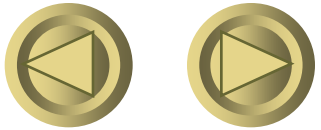
  **INSERT INTO <relation>**

  **( <subquery> );**



# Example: Insert a Subquery

- **Using Frequents(drinker, bar), enter into the new relation PotBuddies(name) all of Sally's "potential buddies," i.e., those drinkers who frequent at least one bar that Sally also frequents.**

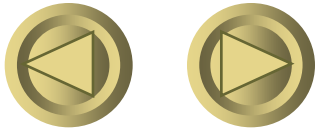


# Solution

The other drinker

Pairs of Drinker tuples where the first is for Sally, the second is for someone else, and the bars are the same.

```
INSERT INTO PotBuddies
(SELECT d2.drinker
 FROM Frequents d1, Frequents d2
 WHERE d1.drinker = 'Sally' AND
   d2.drinker <> 'Sally' AND
   d1.bar = d2.bar
);
```

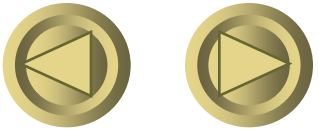


# Deletion

- **To delete tuples satisfying a condition from some relation:**

  **DELETE FROM <relation>**

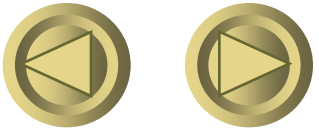
  **WHERE <condition>;**



# Example: Deletion

- **Delete from Likes(drinker, beer) the fact that Sally likes Bud:**

```
DELETE FROM Likes
WHERE drinker = 'Sally' AND
    beer = 'Bud';
```

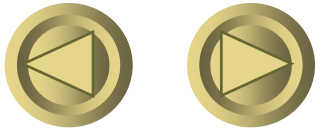


# Example: Delete all Tuples

- **Make the relation Likes empty:**

```
DELETE FROM Likes;
```

- **Note no WHERE clause needed.**

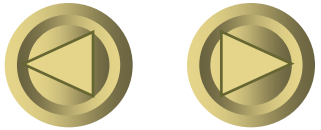


# Example: Delete Many Tuples

- **Delete from Beers(name, manf) all beers for which there is another beer by the same manufacturer.**

```
DELETE FROM Beers b
WHERE EXISTS (
    SELECT name FROM Beers
    WHERE manf = b.manf AND
        name <> b.name);
```

Beers with the same manufacturer and a different name from the name of the beer represented by tuple b.

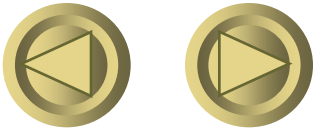


# Semantics of Deletion -- 1

- **Suppose Anheuser-Busch makes only Bud and Bud Lite.**
- **Suppose we come to the tuple *b* for Bud first.**
- **The subquery is nonempty, because of the Bud Lite tuple, so we delete Bud.**
- **Now, When *b* is the tuple for Bud Lite, do we delete that tuple too?**

# Semantics of Deletion -- 2

- **The answer is that we *do* delete Bud Lite as well.**
- **The reason is that deletion proceeds in two stages:**
  1. **Mark all tuples for which the WHERE condition is satisfied in the original relation.**
  2. **Delete the marked tuples.**



# Updates

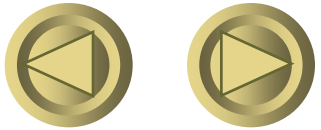
- **To change certain attributes in certain tuples of a relation:**

  **UPDATE <relation>**

  **SET <list of attribute assignments>**

  **WHERE <condition on tuples>;**

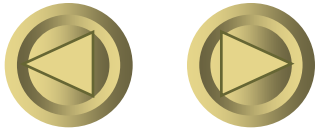
**Updates many tuples at once.**



# Example: Update

- **Change drinker Fred's phone number to 555-1212:**

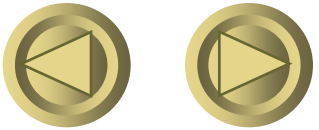
```
UPDATE Drinkers
SET phone = '555-1212'
WHERE name = 'Fred';
```



# Example: Update Several Tuples

- **Make $4 the maximum price for beer:**

```
UPDATE Sells
SET price = 4.00
WHERE price > 4.00;
```



# Adding Attributes

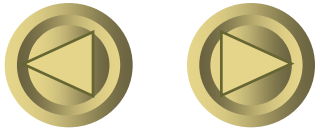
- **We may change a relation schema by adding a new attribute ("column") by:**

  **ALTER TABLE <name> ADD**

  **<attribute declaration>;**

- **Example:**

```
ALTER TABLE Bars ADD
phone CHAR(16)DEFAULT
  'unlisted';
```



# Deleting Attributes

- **Remove an attribute from a relation schema by:**

  **ALTER TABLE <name>**

  **DROP <attribute>;**

- **Example: we don't really need the license attribute for bars:**

```
ALTER TABLE Bars DROP license;
```