Chapter 8: SQL - A Relational Database Language

- relational algebra is high-level language - operations performed on entire relations
  - queries specify how to execute operations

- SQL is declarative - specifies what the expected result is
  - implementation of determines how the result is achieved

- Structured Query Language (originally called SEQUEL)
- variations implemented by most commercial dbms vendors
- revisions: SQL2 (1992) (most of chapter uses this version)
  SQL3 (future version with object-oriented features)

- used for both data definition and data manipulation (via queries)
  - see table 7.1 page 226 for full syntax of SQL commands

Data Definition:

table, row, column used for relational concepts of relation, tuple, and attribute

- creating schemas and tables; dropping and changing tables
  ( refer to handout that creates all of the tables for the TOY database)

CREATE SCHEMA - gives the schema a name and authorization (name of user who created schema)

- can include the descriptors for all elements in the schema (tables, views, domains, etc)
- or define name and authorization and all other elements defined later

CREATE TABLE - used to define a new relation - creates base tables
- specify attributes first with name and data type
- constraints are also specified

  data types:
    numeric - INT, FLOAT, DECIMAL(I,J) , DOUBLE PRECISION
    string - CHAR(N), VARCHAR(N)
    bitstring - BIT(n), BIT VARYING(n)
    date/time - DATE (in yyyy-mm-dd format)
      TIME (hh:mm:ss)
      TIMESTAMP (both date and time)
    interval - INTERVAL (relative time - YEAR/MONTH or DAY/TIME intervals)

  constraints:
    key constraint - NOT NULL (must be explicitly specified)
    definition of primary and foreign keys
    referential triggered action - SET NULL, CASCADE, SET DEFAULT
    constraints can be named by preceding them with the keyword CONSTRAINT
    - names within a schema must be unique
DROP TABLE - used to eliminate a relation from a schema

**DROP TABLE** TOY CASCADE;

- CASCADE option removes all constraints and views that reference the table
- RESTRICT option only drops table if no other elements reference the table

ALTER TABLE - change the named table

**ALTER TABLE** TOY **ADD** SOLDSINCEDATE;

- ADD - values of new attribute will have null values
- DROP - can choose CASCADE (remove all elements that refer to attrib) or RESTRICT (only drops if no elements refer to attrib)
- also add/drop defaults and constraints

Data Manipulation - SQL queries

**SELECT** statement - one statement for retrieving data from db
- no related to relational algebra select operation

**SELECT** <attrib list> -- list of attrib names - projection attribs
**FROM** <table list> -- list of relation names
**WHERE** <cond> -- Boolean expression (select and join conds)

Some example queries to illustrate the various uses of the **SELECT** FROM WHERE clause

1) Retrieve the address and last order date of the customer named Karen Smith.

```sql
SELECT ADDRESS, LAST_ORDER_DATE
FROM CUSTOMER
WHERE NAME = 'KAREN SMITH'
```

- single relation in from clause - similar to select-project pair of relational algebra

2) Retrieve the name and price of all toys made by Fischer-Price.

```sql
SELECT NAME, MSRP
FROM TOY, MANUFACTURER
WHERE MAN_NAME = 'FISCHER PRICE' AND TOY.MAN_ID=MANUFACTURER.MAN_ID
```

- select-project-join query
- MAN_NAME = ‘FISCHER PRICE’ is the select-project part;
  TOY.MAN_ID=MANUFACTURER.MAN_ID is the join part

3) For every undelivered order, list the toy name, the manufacturer name and the customer name

```sql
```
SELECT TOY.NAME, MAN_NAME, CUSTOMER.NAME
FROM ORDER, TOY, MANUFACTURER, CUSTOMER
WHERE DELIV='00/00/00' AND ORDER.CUST_NUM = CUSTOMER.CUST_NUM AND ORDER.TOX_NUM = TOY.TOX_NUM AND TOY.MAN_ID = MANUFACTURER.MAN_ID

- notice that the first part of the where (deliv = ...) sets the condition
- the next two link the order relation with customer and toy
- the last one is needed to link toy to manufacturer

4) Retrieve the name of every toy in the toy relation.

SELECT NAME
FROM TOY

- missing WHERE clause selects all tuples

5) Retrieve the name of every toy and the name of every manufacturer.

SELECT NAME, MAN_NAME
FROM TOY, MANUFACTURER

- equivalent to doing a cross product followed by a project

6) Retrieve all attributes of the TOY relation for which the manufacturer is FP

SELECT *
FROM TOY
WHERE MAN_ID = 'FP'

7) List the prices of all toys in the TOY relation.

SELECT MSRP
FROM TOY

- notice that the result can have duplicate tuples (45.00)
- sometimes we want this - as in the case where we do aggregate functions on results
- if we don’t want duplicates - we specify DISTINCT in the SELECT clause

SELECT DISTINCT MSRP
FROM TOY

- can also apply UNION, INTERSECT and EXCEPT (difference) operations to query results

8) (Using the Company db from the textbook - because our example does not have recursion)
Make a list of all project names for projects that involve an employee whose last name is SMITH as a worker or as a manager of the dept that controls the project.

```
(SELECT PNAME
FROM EMPLOYEE, WORKS_ON, PROJECT
WHERE LNAME=SMITH AND SSN=ESSN AND
  PNO=PNUMBER)
UNION
(SELECT PNAME
FROM EMPLOYEE, DEPARTMENT, PROJECT
WHERE LNAME=SMITH AND SSN=MGRSSN AND
  DNUMBER=DNUM)
```

- eliminates duplicates because it uses UNION (set operator)

9) Reformulate the above query as a nested query

```
SELECT DISTINCT PNUMBER
FROM PROJECT
WHERE PNUMBER IN (SELECT PNUMBER
  FROM PROJECT, DEPARTMENT,
   EMPLOYEE
  WHERE DNUM=DNUMBER AND
    MGRSSN=SSN AND
    LNAME=SMITH)
  OR
  PNUMBER IN (SELECT PNO
            FROM WORKS_ON, EMPLOYEE
            WHERE ESSN=SSN AND
                LNAME=SMITH)
```

- the first select chooses the project numbers that have SMITH as a manager
- the second select chooses the project numbers that have SMITH as an employee
- the outer query chooses the distinct project numbers from the results of both nested queries

10) Select the toy numbers of all toys that have the same price and age group as the Farm House.

```
SELECT DISTINCT TOY_NUM
FROM TOY
WHERE (MSRP, AGE_GROUP) IN (SELECT MSRP, AGE_GROUP
  FROM TOY
  WHERE NAME = FARM HOUSE)
```

- the IN operator compares a tuple of values in parentheses with a set of union compatible tuples
- can use other comparison operators similarly

11) Select the toy names of all toys that cost more than the Farm House.

```
SELECT DISTINCT TOY_NAME
FROM TOY
WHERE MSRP > (SELECT MSRP
               FROM TOY
               WHERE NAME = FARM HOUSE)
```
SELECT NAME
FROM TOY
WHERE MSRP > ALL (SELECT MSRP
FROM TOY
WHERE NAME=FARM HOUSE)

- ALL is used because NAME is not a key field

- possible problem with ambiguous names in nested queries - in the FROM clause of the outer query and the FROM clause of the nested query

- rule: reference to unqualified attribute refers to the relation declared in the innermost nested query

- to refer to attributes in an outer query, aliases are used

12) (from Company db) Retrieve the name of each employee who has a dependent with the same first name and sex as the employee.

SELECT E.FNAME, E.LNAME
FROM EMPLOYEE E
WHERE E.SSN IN (SELECT ESSN
FROM DEPENDENT
WHERE ESSN=E.SSN AND E.FNAME=DEPENDENT_NAME AND SEX=E.SEX)

- we need to use E to refer to the SEX attribute of the EMPLOYEE relation of the outer query

- this query is a correlated query - the WHERE clause of a nested query references an attribute in the relation listed in the outer query

- to evaluate this type of query, every tuple of the inner query is tested with every tuple of the inner query

- i.e. for every employee, evaluate the inner query to see if there is a dependent with the same sex and fname

- for non-correlated queries, inner queries are evaluated first and then the outer query is performed on the results

13) Query 12 can be rewritten using the EXISTS clause

SELECT E.FNAME, E.LNAME
FROM EMPLOYEE E
WHERE EXISTS (SELECT *
FROM DEPENDENT
WHERE E.SSN=ESSN AND SEX=E.SEX AND E.FNAME=DEPENDENT_NAME)

- EXISTS usually used in conjunction with a correlated nested query
- for each employee tuple, evaluate the nested query, which retrieves all dependent tuples with the same ssn, sex and name as the employee tuple; if at least one tuple EXIST in the result, select that employee tuple

- EXISTS returns true if at least one tuple is in the result

14) List the names of customers who have no outstanding orders.

```
SELECT NAME
FROM CUSTOMER C
WHERE NOT EXISTS (SELECT *
FROM ORDER
WHERE C.CUST_NUM=ORDER.CUST_NUM
AND DELIV=00/00/00)
```

- the nested query retrieves all toys related to the given manufacturer
- if none exists, select that manufacturer tuple

15) Retrieve the names of all toys manufactured by FP or FY.

```
SELECT TOY_NAME
FROM TOY
WHERE MAN_ID IN (FP, FY)
```

- can explicitly specify a set of values using an IN clause.

16) Retrieve the names of customers who have never ordered a toy from the catalog.

```
SELECT NAME
FROM CUSTOMER
WHERE LAST_ORDER_DATE IS NULL
```

- can look for NULL values

17) Retrieve the toy names and the customer names for every outstanding order for toys whose names fall in the first half of the alphabet.

```
SELECT T.NAME AS TOY_NAME, C.NAME AS CUSTOMER_NAME
FROM CUSTOMER AS C, TOY AS T, ORDER AS O
WHERE (C.CUST_NUM=O.CUST_NUM) AND (DELIV=00/00/00)
AND (T.TOY_NUM=ORDER.TOY_NUM) AND (TOY_NAME < 'N')
```

- kind of a contrived example - but shows how to use AS both for attribute names and relation names to make the query shorter and easier to understand
- the new name can be used throughout the query and for attribute names, appears in the column heading of the result

18) Retrieve the toy number of every toy ordered by KAREN SMITH.
SELECT TOY_NUM
FROM (ORDER JOIN CUSTOMER ON
ORDER.CUST_NUM=CUSTOMER.CUST_NUM)
WHERE NAME='KAREN SMITH'

- another way to specify a join condition - create a join table in the FROM clause
- some users find this easier than mixing the select and join clauses in the
  WHERE part of the query
- can also specify NATURAL JOINs and OUTER JOINs

19) Find the average price of all toys in the TOY relation.

SELECT AVG(MSRP)
FROM TOY

- built-in aggregate functions SUM, MAX, MIN, AVG, COUNT

20) Find the total number of toys orderd by and the total amount of money spent by
    customer GEORGE GRANT.

SELECT SUM(MSRP), COUNT(*)
FROM TOY AS T, CUSTOMER AS C, ORDER AS O
WHERE O.TOY_NUM=T.TOY_NUM AND
O.CUST_NUM=C.CUST_NUM AND
C.NAME='GEORGE GRANT'

- COUNT(*) returns the number of tuples that satisfy the query

21) Find the total number of toys order by and the total amount of money spent by each
    customer.

SELECT CUST_NUM, SUM(MSRP), COUNT(*)
FROM TOY AS T, ORDER AS O
WHERE O.TOY_NUM=T.TOY_NUM
GROUP BY CUST_NUM

- can specify a grouping attribute to apply a function to each group
  independently
- because we are doing this for every customer, we do not need to join with the
  customer relation
- the grouping attribute must be selected in the SELECT clause

22) Find the total number of toys order by and the total amount of money spent by each
    customer who has ordered at least 3 toys.

SELECT CUST_NUM, SUM(MSRP), COUNT(*)
FROM TOY AS T, ORDER AS O
WHERE O.TOY_NUM=T.TOY_NUM
GROUP BY CUST_NUM
HAVING COUNT(*) > 3
- HAVING clause allows you to put a condition on the groups that end up in the result

23) Retrieve all customers who live in New York state.

```
SELECT NAME
FROM CUSTOMER
WHERE ADDRESS LIKE '%NY%
```

- the LIKE clause compares partial strings
  - '%' replaces an arbitrary number of characters
  - '_' replaces a single arbitrary character

24) Show the new prices if Fischer Price raised their MSRP's by 10%.

```
SELECT NAME, 1.1*MSRP
FROM TOY
WHERE MAN_ID=FP
```

- can use standard arithmetic operators: +, -, *, /

25) Retrieve all toys with fewer than 50 in inventory sorted by manufacturer and by price within each manufacturer.

```
SELECT MAN_ID, NAME, MSRP
FROM TOY
WHERE NUM_IN_STOCK<50
ORDER BY MAN_ID, MSRP
```

- the result of a query can be sorted on any field
- default is ascending - can specify descending with the keyword DESC

```
ORDER BY MAN_ID DESC, MSRP
```

Updates in SQL: INSERT, DELETE, UPDATE

**INSERT:**
- add a single tuple to a relation
- list name of relation and attribute values in order the attributes appeared in CREATE TABLE command

```
ex: INSERT INTO CUSTOMER
VALUES (004, 'KEN LIGHT', '15 STONE ROAD LOS ANGELES, CA 11111', 2, '5,6', '(555)333-4334', 12/12/95)
```

- can specify attributes to fill - default (or NULL) used for all other attributes
ex:  INSERT INTO CUSTOMER(CUST_NUM, NAME) VALUES (004, ‘KEN LIGHT’)

- with the INSERT command, the user is responsible for checking any integrity constraints that the DBMS does not support

DELETE:
- removes tuples from a relation
- one tuple at a time
- may propagate to other tuples due to referential integrity constraints
- tuples satisfying a condition clause are removed
- if condition is blank, removes all tuples

ex:  DELETE FROM MANUFACTURER WHERE MAN_NAME=FISCHER PRICE

DELETE FROM TOY AS T WHERE T.MAN_ID IN (SELECT M.MAN_ID FROM MANUFACTURER AS M WHERE M.MAN_NAME= FISCHER PRICE)

UPDATE:
- modifies attributes of existing tuples
- one or more tuples (dependin on the WHERE clause)
- one relation at a time
- updating a primary key may propagate to foreign keys in other relations

ex:  UPDATE TOY AS T SET MSRP=MSRP*1.1 WHERE T.MAN_ID IN (SELECT M.MAN_ID FROM MANU AS M WHERE M.MAN_NAME= FISCHER PRICE)

Views in SQL:
- virtual relations
- not actually stored - based on other relations (actual or virtual)
- useful for users who specify the same query frequently
- since the view is not stored, changes to the relations involved are reflected in the view - view is “realized” at the time the query is specified

ex:  CREATE VIEW UNSENT_ORDERS AS SELECT NAME, ADDRESS, TOY_NUM FROM ORDER, CUSTOMER WHERE ORDER.CUST_NUM=CUSTOMER.CUST_NUM AND DELIV=00/00/00

- can write queries on this view just as we did on the stored relations
- can specify new attribute names for the view
- to remove a view, use DROP VIEW command

```
DROP VIEW UNSENT_ORDERS
```

Updating views - considered a complex research problem
- modification of all relations involved is not straightforward and can be ambiguous

ex:
```
CREATE VIEW PRICES
AS SELECT NAME, MSRP, MAN_NAME
FROM TOY, MANUFACTURER
WHERE TOY.MAN_ID=MANU.MAN_ID
```
```
UPDATE PRICES
SET MAN_NAME='HAPPY YEARS'
WHERE NAME='WIGGLE WORM' AND MAN_NAME='FIRST YEARS'
```

- there are two different ways this update can be performed and still have the result in the view be correct:
  1) update all man names in MANUFACTURER from 'FIRST YEARS' to HAPPY YEARS
  2) make the update only for the toy called WIGGLE WORM

- the first interpretation does what the update wants, but also updates more than

- there is no way to guarantee that any view can be updated
- some researchers try to find a way to perform the most likely interpretation of an update
- others ask the user for the correct interpretation

- rules of thumb:
  1) single relation view is updatable if a key is involved in the view
  2) views based on joins are generally not updatable
  3) views based on aggregate functions are generally not updatable

Indexes in SQL (Part of the DDL)
- not present in all implementations - use keys as indexes
- not part of SQL2 - because it specifies a physical access path
- read section 7.6 if interested

Embedded SQL:
- permit embedding a query into the control constructs of a host language like C
- ex: loop to permit processing on several tuples at a time