Database Languages

In ordinary programming languages the declarations and executable statements are all part of one language. In the database world, however, it is common to separate the two functions of declaration and computation into two different languages. While in an ordinary program data exists only while the program is running, in a database system, the data persists and may be declared once and for all.

Data Definition Languages (DDL)

The data definition language is not a procedural language. It is used when the database is designed and when that design is modified. It is not used for obtaining or modifying the data itself. The data definition language has statements that describe, in somewhat abstract terms what the physical layout of the database should be. Detailed design of the physical database is done by compiled statements in the data definition language.

The description of subschemes and their correspondence to the conceptual scheme requires a subscheme data definition language, which is often quite similar to the data definition language itself.

Data Manipulation Language (DML)

Operations on the database require a specialized language, called a data manipulation language or query language.

Host Languages

Often, manipulation of the database is done by an **application program**, written in advance to perform a certain tasks. For example, a program used to book reservations needs to make a decision: are there enough seats available?
Thus, programs to manipulate the database are commonly written in a host language, which is a conventional programming language such as C, C++, Java, C#, or even COBOL. The host language is used for decisions, for displaying questions, and for reading answers; in fact, it is used for everything but the actual querying and modification of the database.

Relational Query Languages

Relational query languages are used in systems built upon the relational model of data. A language that can (at least) simulate safe tuple calculus, or equivalently, relational algebra or safe domain calculus, is said to be ***complete***. DML have capabilities of relational algebra or calculus; they include insertion, deletion, and modification commands; they have some additional features:

1. Arithmetic capability. Atoms in calculus expressions or selections in algebraic expressions can involve arithmetic computations as well as comparisons, e.g., A<B+3. Note that + and other arithmetic operators appear in neither relational algebra nor calculus.
2. Assignment and print commands. Languages generally allow the printing of the relation constructed by an algebraic or calculus expressions and the assignment of a computed relation to be the value of a relation name.
3. Aggregate functions. Operations such as average, sum, min, or max can often be applied to columns of a relation to obtain a single quantity.

The Query Language SQL (Structured Query Language)

SQL, formerly knows as SEQUEL is the most commonly implemented relational language. It is developed by IBM in San Hose, originally for use in the experimental database system known as System R.

The SELECT Statement

SELECT $R_{i_1}.A_1,\ldots,R_{i_r}.A_r$
FROM $R_1,\ldots,R_k$
WHERE Ψ;

Here, $R_1,\ldots,R_k$ is a list of relation names, and $R_{i_1}.A_1,\ldots,R_{i_r}.A_r$ is a list of component references to be printed; R.A refers to the attribute A of relation R. If only one relation in the list following the keyword FROM has an attribute A, then we may use A in place of R.A in the selected list. Ψ is a formula involving logical connectives AND, OR, and NOT, and comparison operators =, <=, and so on.
The meaning of query is most easily expressed in relational algebra as:

$$\Pi_{R_{i_1}.A_1,\ldots,R_{i_r}.A_r}\left(\sigma_{\Psi}\left(R_1\times\ldots\times R_k\right)\right)$$

We take the product of all the relations in the FROM-clause, select according to the WHERE-clause (Ψ is replaced by an equivalent expression Ψ', using the operators of relational algebra, and finally project onto the attributes of the SELECT-clause.

Example: Print the names of customers with negative balances.

```
SELECT CNAME
FROM CUSTOMERS
WHERE BALANCE<0;
```

Here, since there is only one relation in the FROM-clause, thus, we did not have to prefix attributes by their relation names. However, we could have written:

```
SELECT CUSTOMERS.CNAME
FROM CUSTOMERS
WHERE CUSTOMERS.BALANCE<0;
```

Had we wanted another header for the column, we could have provided an alias for CNAME by writing that alias immediately after the AS clause in the SELECT-clause. Thus,

```
SELECT CNAME AS CUSTOMER
FROM CUSTOMERS
WHERE BALANCE<0;
```

Had we wished to print the entire tuple for customers with a negative balance, we could have written

```
SELECT CNAME, CADDR, BALANCE
FROM CUSTOMERS
WHERE BALANCE<0;
```

or just

```
SELECT *
FROM CUSTOMERS
WHERE BALANCE<0;
```

Example: Print the suppliers who supply at least one item ordered by Zack Zebra.

Microsoft Access uses brackets surrounded a name if the name contains a space or a special character.

```
SELECT SNAME
FROM ORDERS, INCLUDES, SUPPLIES
WHERE CUST = 'Zack Zebra' AND
      ORDERS.[O#] = INCLUDES.[O#] AND
      INCLUDES.INAME=SUPPLIES.INAME;
```

Here, we take the natural join of ORDERS, INCLUDES and SUPPLIES, using equalities in the WHERE-clause to define the join. The WHERE-clause also contains the condition that the customer be Zack Zebra, and the SELECT-clause causes only the supplier name to be printed.

$$\Pi_{SNAME}\left(\sigma_{CUST = \text{'Zack Zebra'}}\left(ORDERS \underset{ORDERS.O\# = INCLUDES.O\#}{\theta} INCLUDES \underset{INCLUDES.INAME = SUPPLIES.INAME}{\theta} SUPPLIES\right)\right)$$

The attribute CUST and SNAME refer to ORDERS and SUPPLIES, respectively, so they do not have to be prefixed by a relation name. However, O# is an attribute of both ORDERS and INCLUDES, so it has to be prefixed by the relations intended. The same is for INAME.

SQL does not remove duplicates automatically. To remove duplicates, we use the keyword DISTINCT following SELECT:

```
SELECT DISTINCT SNAME
```

Tuple Variables

Sometimes we need to refer to two or more tuples in the same relation. We define several tuple variables for that relation in the FROM-clause and we use the tuple variables aliases of the relation.

Example: Print the name and address of each customer whose balance is lower than Judy Giraffe's.

```
SELECT c1.CNAME, c1.CADDR
FROM CUSTOMERS c1, CUSTOMERS c2
WHERE c1.BALANCE<c2.BALANCE AND
      c2.CNAME = 'Judy Giraffe';
```

c1 and c2 are aliases of CUSTOMERS, they are tuple variables for CUSTOMERS.

tupple variables

Pattern Matching

An operator LIKE in the WHERE-clause expresses the condition that a certain value matches a pattern.

?	Any single character.
*	Zero or more characters.
#	Any single digit (0 - 9)
[charlist]	Any single character in charlist.
[!charlist]	Any single character not in charlist.

Example: Print those items that begin with "B".

```
SELECT INAME
FROM SUPPLIES
WHERE INAME LIKE 'B*';
```

Example: Print those orders whose number is in the range 1000-1999, i.e., those whose order members are a "1" followed by any three characters. We assume that order numbers are stored as character strings, rather than integers.

```
SELECT *
FROM ORDERS
WHERE [O#] LIKE '1###';
```

Set Operations in the WHERE-Clause

We can use selection conditions in a WHERE-clause. SQL allows sets as operands; these sets are defined by complete SELECT-FROM-WHERE statements nested within a where-clause, and are called subqueries. The operators IN, NOT IN, ANY, and ALL are used, respectively, to denote membership in a set, nonmembership, existential quantification over a set, and universal quantification over a set.

Example: Print the suppliers who supply at least one item ordered by Zack Zebra, using subqueries to replace a sequence of joins by semijoins.
1. Find the set S_1 of orders placed by Zebra, using ORDERS.
2. Find the set S_2 of items in set of orders S_1, using INCLUDES.
3. Find the set S_3 of suppliers of the items in set S_2, using SUPPLIES.

```
SELECT SNAME                                   S3
FROM SUPPLIES
WHERE INAME IN
      (SELECT INAME                            S2
       FROM INCLUDES
       WHERE [O#] IN
             (SELECT [O#]                      S1
              FROM ORDERS
              WHERE CUST = 'Zack Zebra'));
```

Example: Print each item whose price is as large as any appearing in the SUPPLIES relation by using a subquery to form the set S of all prices, and then saying that the price of a given item is as large as any in the set S. (Finding the most costly item)

```
SELECT INAME
FROM SUPPLIES
WHERE PRICE>=ALL
      (SELECT PRICE
       FROM SUPPLIES);
```

Aggregate Operators

SQL provides aggregate operators:

- AVG
- COUNT
- SUM
- MIN
- MAX
- STDDEV (standard deviation)
- VARIANCE (variance of a list of numbers)

```
agg_op (A)
```

where A is the attribute of the aggregate operator agg_op.

Example: Compute the average balance in the database.

```
SELECT AVG (BALANCE)
FROM CUSTOMERS;
```

The print result is with a column header AVG(BALANCE). To change the column header, say AV_BAL, we could specify an alias, as in:

```
SELECT AVG(BALANCE) AS AV_BAL
FROM CUSTOMERS;
```

Example: Count the number of suppliers.
We must eliminate duplicates before we count.

```
SELECT COUNT(SNAME) AS [#SUPPS]
FROM SUPPLIES
WHERE SNAME IN
   (SELECT DISTINCT SNAME
    FROM SUPPLIES);
```

Thus, we will print the number of different suppliers in a column headed by #SUPPS.

Example: Count the number of suppliers who sell Brie.

```
SELECT COUNT(SNAME) AS [#BRIE_SUPPS]
FROM SUPPLIES
WHERE INAME = 'Brie';
```

Note it is unnecessary to remove duplicates here, because the fact that a supplier sells Brie appears only once, assuming {SNAME, INAME} is a key for SUPPLIES in the database.

Example: Print each item whose price is as large as any appearing in the SUPPLIES relation by using a subquery to form the set S of maximum price, and then saying that the price of a given item is as large as the maximum once in the set S. (Finding the most costly item)

```
SELECT INAME
FROM SUPPLIES
WHERE PRICE IN
      (SELECT MAX(PRICE)
       FROM SUPPLIES);
```

Aggregation by Groups

The clause

GROUP BY $A_1,...A_k$

partitions the relation into groups, such that two tuples are in the same group if and only if they agree on all the attributes $A_1,...,A_k$. The attributes $A_1,...,A_k$ must also appear in the SELECT-clause, although they could be given aliases for printing.

Example: Print a table of all the items and their average prices.

```
SELECT INAME, AVG(PRICE) AS [Average Price]
FROM SUPPLIES
GROUP BY INAME;
```

If we write

GROUP BY $A_1,...A_k$
HAVING Ψ

then the condition Ψ is applied to each relation $Ra_1,...,a_k$ that consists of the group of tuples with values $a_1,...,a_k$ for attributes $A_1,...,A_k$, respectively. Those groups for which satisfies Ψ are part of the output, and the others do not appear.

Example: Print a table of those items that were sold by more than one supplier and their average prices.

```
SELECT INAME, AVG(PRICE) AS [Average Price]
FROM SUPPLIES
GROUP BY INAME
HAVING COUNT(*)>1;
```

* stands for all attributes of the relation referred to.

Insertion

To insert new tuples into a relation we use the statement form:

```
INSERT INTO R
VALUES (a1,a2,...,ak);
```

Here, R is a relation name and $a_1,a_2,...,a_k$ is a list of values for the attributes of R. These attributes are given a particular order when the relation R is declared.

Example: If Ajax starts selling Country Wine at $4.50 each, we can say:

```
INSERT INTO SUPPLIES
VALUES ('Ajax', 'Country Wine', 4.50);
```

We do not have to specify values for all attributes. If a value is not provided, NULL will be the assumed value.

MS Access converts the VALUE clause to a SELECT clause.

```
INSERT INTO SUPPLIES
SELECT 'Ajax', 'Country Wine', 4.50;
```

Example: Suppose that the attribute PRICE of SUPPLIES may have nulls. Ajax sells Country Wine but we don't know the price.

```
INSERT INTO SUPPLIES (SNAME, INAME)
VALUES ('Ajax', 'Country Wine');
```

Example: Suppose we wanted a new relation

ACME_SELLS (INAME, PRICE)

that listed just the item and price components of the SUPPLIES tuples with SNAME equal to "Acme". The insert command is

```
INSERT INTO ACME_SELL(INAME, PRICE)
SELECT INAME, PRICE
FROM SUPPLIES
WHERE SNAME = 'Acme';
```

The VALUES-clause is replaced by a SELECT-FROM-WHERE statement that produces a relation of values.

Deletion

The form of the deletion command is

```
DELETE R.* FROM R
WHERE Ψ;
```

R is a relation name and Ψ is a condition. Every tuple of R for which Ψ is true is deleted form R.

Example: Acme no longer sells Perrier.

```
DELETE SNAME, INAME FROM SUPPLIES
WHERE SNAME = 'Acme' AND INAME = 'Perrier';
```

Example: Delete from ORDERS relation all orders that included Brie.

```
DELETE [O#] FROM ORDERS
WHERE [O#] IN
      (SELECT [O#]
       FROM INCLUDES
       WHERE INAME = 'Brie');
```

Update

The general form of an update command is

```
UPDATE R
SET A1=ε1,...,Ak=εk
WHERE Ψ;
```

Example: Update the price Acme charges for Perrier to $1.00.

```
UPDATE SUPPLIES
SET PRICE = 1.00
WHERE SNAME = 'Acme' AND INAME = 'Perrier';
```

Example: Lower all of Acme's prices by 10%.

```
UPDATE SUPPLIES
SET PRICE = 0.9*PRICE
WHERE SNAME = 'Acme';
```

Completeness of SQL

SQL can simulate expressions of relational algebra.

1. To rename the attributes of the relation $S(B_1,B_2,...,B_m)$ with those of $R(A_1,A_2,...A_n)$, we can create a new relation Snew with the same attributes, $A_1,A_2,...A_n$ as R. We then copy S into Snew by:

   ```
   INSERT INTO Snew
   SELECT *
   FROM S;
   ```

2. To compute the union RUS we write

   ```
   INSERT INTO T
   SELECT *
   FROM R;
   INSERT INTO T
   SELECT *
   FROM S;
   ```

3. To compute the set difference T=R-S:

```
INSERT INTO T
SELECT *
FROM R;
DELETE FROM T
WHERE (A1,...,An) IN          this list refers to T
      (SELECT *               subquery for S
       FROM S);
```

4. For the Cartesian product T=RxS we say:

```
INSERT INTO T
SELECT R.A1,...,R.An,S.B1,...,S.Bm
FROM R, S;
```

5. The selection $T=\sigma_F(R)$ is written

```
INSERT INTO T
SELECT *
FROM R
WHERE F';
```

F' is the selection condition F translated into SQL notation.

6. The projection $\Pi i_1,i_2,...,i_m(R)$ is written

```
INSERT INTO T
SELECT Ai1,...,Aim
FROM R;
```

Data Definition in SQL

The DML and DDL in SQL are really two sets of commands that are part of a single language.

Create

The most fundamental DDL command is the one that creates a new relation.

```
CREATE TABLE R (data typed list of attributes)
```

If the attribute is a primary key, we use:

```
CONSTRAINT name PRIMARY KEY
```

Example: We can define the AJAX_SELLS relation scheme by

```
CREATE TABLE AJAX_SELLS
(INAME TEXT(20) CONSTRAINT MyKey PRIMARY
KEY,  PRICE CURRENCY);
```

The two attributes are INAME (a string of up to 20 characters), that is a primary key and PRICE (a currency).

Delete

```
DROP TABLE R;
```

Example: To delete the relation AJAX_SELLS from the database entirely, we would write

```
DROP TABLE AJAX_SELLS;
```

Creation of Indices

Indices are used to speed up access to a relation. If relation R has an index on attribute A, then we can retrieve all the tuples with a given value a for attribute A, in time roughly proportional to the number of such tuples, rather than in time proportional to the size of R. That is, in the absence of an index on A, the only way to find the tuples μ in R such that $\mu[A]=a$ is to look at all tuples in R.

```
CREATE INDEX I
ON R(A);
```

An index named I is created on attribute A of relation R.

Example:

```
CREATE INDEX [O#_INDEX]
ON ORDERS ([O#]);
```

This creates an index on attribute O# of the relation ORDERS.

```
CREATE UNIQUE INDEX [O#_INDEX]
ON ORDERS ([O#]);
```

The index O#_INDEX would not only speed up process given an order number, but it would make sure, that we never had two tuples with the same order number.

Deletion of Indices

```
DROP INDEX I;
```

Example:

```
DROP INDEX [O#_INDEX];
```