Chapter 4

SQL

Commercial database systems require more user-friendly query languages. We will look at

- **SQL** in detail.
- **QBE** briefly in the next chapter.
- **Quel** briefly in the next chapter.

Although referred to as query languages, they each contain facilities for designing and modifying the database.

1. The relation schemes for the banking example used throughout the new edition of the textbook are:
   - *Branch-scheme = (bname, bcity, assets)*
   - *Customer-scheme = (cname, street, ccity)*
   - *Depositor-scheme = (cname, account#)*
   - *Account-scheme = (bname, account#, balance)*
   - *Loan-scheme = (bname, loan#, amount)*
   - *Borrower-scheme = (cname, loan#)*

4.1 Background

1. SQL has become *the* standard relational database language. It has several parts:
   - Data definition language (DDL) - provides commands to
     - Define relation schemes.
     - Delete relations.
     - Create indices.
     - Modify schemes.
   - Interactive data manipulation language (DML) - a query language based on both relational algebra and tuple relational calculus, plus commands to insert, delete and modify tuples.
   - Embedded data manipulation language - for use within programming languages like C, PL/1, Cobol, Pascal, etc.
   - View Definition - commands for defining views
   - Authorization - specifying access rights to relations and views.

- Integrity - a limited form of integrity checking.
- Transaction control - specifying beginning and end of transactions.

We will only look at basic DDL, the DML and views. Integrity features will be covered in Chapter 5.

4.2 Basic Structure

1. Basic structure of an SQL expression consists of **select, from** and **where** clauses.
   - **select** clause lists attributes to be copied - corresponds to relational algebra **project**.
   - **from** clause corresponds to Cartesian product - lists relations to be used.
   - **where** clause corresponds to selection predicate in relational algebra.
2. Typical query has the form

   select $A_1, A_2, \ldots, A_n$
   from $r_1, r_2, \ldots, r_m$
   where P

   where each A_i represents an attribute, each r_i a relation, and P is a predicate.
3. This is equivalent to the relational algebra expression

$$\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

   - If the where clause is omitted, the predicate P is true.
   - The list of attributes can be replaced with a * to select all.
   - SQL forms the Cartesian product of the relations named, performs a selection using the predicate, then projects the result onto the attributes named.
   - The result of an SQL query is a relation.
   - SQL may internally convert into more efficient expressions.

4.2.1 The select Clause

1. An example: Find the names of all branches in the *account* relation.

   select *bname*
   from *account*

2. **distinct** vs. **all**: elimination or not elimination of duplicates.

   Find the names of all branches in the *account* relation.

   select distinct *bname*
   from *account*

   By default, duplicates are not removed. We can state it explicitly using **all**.

   select all *bname*
   from *account*

3. select * means select all the attributes. Arithmetic operations can also be in the selection list.

4.2.2 The where Clause

1. The predicates can be more complicated, and can involve
   - Logical connectives **and, or** and **not**.
   - Arithmetic expressions on constant or tuple values.
   - The **between** operator for ranges of values.
2. Example: Find account number of accounts with balances between \$90,000 and \$100,000.

   select *account#*
   from *account*
   where *balance* **between** *90000* **and** *100000*

4.2.3 The from Clause

1. The **from** class by itself defines a Cartesian product of the relations in the clause.
2. SQL does not have a natural join equivalent. However, natural join can be expressed in terms of a Cartesian product, selection, and projection.
3. For the relational algebra expression

$$\Pi_{cname,loan\#}(borrower \bowtie loan)$$

   we can write in SQL,

   select distinct *cname, borrower.loan#*
   from *borrower, loan*
   where *borrower.loan# = loan.loan#*

4. More selections with join: "Find the names and loan numbers of all customers who have a loan at the SFU branch," we can write in SQL,

   select distinct *cname, borrower.loan#*
   from *borrower, loan*
   where *borrower.loan# = loan.loan#*
   and *bname="SFU"*

4.2.4 The Rename Operation

1. Rename: a mechanism to rename both relations and attributes.
2. **as**-clause can appear in both the select and from clauses:

   old-name **as** *new-name*.

3. Example.

   select distinct *cname, borrower.loan#* **as** *loan_id*
   from *borrower, loan*
   where *borrower.loan# = loan.loan#* **and** *bname= "SFU"*

4.2.5 Tuple Variables

1. Tuple variables can be used in SQL, and are defined in the **from** clause:

   select distinct *cname, T.loan#*
   from *borrower* **as** *S, loan* **as** *T*
   where *S.loan# = T.loan#*

   Note: The keyword **as** is optional here.

2. These variables can then be used throughout the expression. Think of it as being something like the rename operator.

   Finds the names of all branches that have assets greater than at least one branch located in Burnaby.

   select distinct *T.bname*
   from *branch S, branch T*
   where *S.bcity="Burnaby"* **and** *T.assets* > *S.assets*

4.2.6 String Operations

1. The most commonly used operation on strings is pattern matching using the operator **like**.
2. String matching operators % (any substring) and _ (underscore, matching any character).

   E.g., "__%" matches any string with at least 3 characters.
3. Patterns are case sensitive, e.g., "Jim" does not match "jim".
4. Use the keyword **escape** to define the *escape* character.

   E.g., like "ab%tely\%\\" escape "\" matches all the strings beginning with "ab" followed by a sequence of characters and then "tely" and then "%\".

   Backslash overrides the special meaning of these symbols.
5. We can use **not like** for string mismatching.
6. Example. Find all customers whose street includes the substring "Main".

   select *cname*
   from *customer*
   where *street* **like** *"%Main%"*
7. SQL also permits a variety of functions on character strings, such as concatenating (using "||"), extracting substrings, finding the length of strings, converting between upper case and lower case, and so on.

4.2.7 Ordering the Display of Tuples

1. SQL allows the user to control the order in which tuples are displayed.

   - **order by** makes tuples appear in sorted order (ascending order by default).
   - **desc** specifies descending order.
   - **asc** specifies ascending order.

   select *
   from *loan*
   order by *amount* **desc**, *loan#* **asc**

   Sorting can be costly, and should only be done when needed.

4.2.8 Duplicate Tuples

- Formal query languages are based on mathematical relations. Thus no duplicates appear in relations.
- As duplicate removal is expensive, SQL allows duplicates.
- To remove duplicates, we use the **distinct** keyword.
- To ensure that duplicates are not removed, we use the **all** keyword.
- *Multiset* (bag) versions of relational algebra operators.

  - if there are c_1 copies of tuples t_1 in r_1, and t_1 satisfies selection σ_θ, then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$.
  - for each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$.
  - if there are c_1 copies of tuple t_1 in r_1, and c_2 copies of tuple t_2 in r_2, there is $c_1 \times c_2$ copies of tuple $t_1.t_2$ in $r_1 \times r_2$.

- An SQL query of the form

  select $A_1, A_2, \ldots, A_n$
  from $r_1, r_2, \ldots, r_m$
  where P

  is equivalent to the algebra expression

$$\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

  using the multiset versions of the relational operators σ, Π, and $\times$.

4.3 Set Operations

1. SQL has the set operations **union, intersect** and **except**.

2. Find all customers having an account.

   select distinct *cname*
   from *depositor*

3. **union**: Find all customers having a loan, an account, or both. branch.

   (**select** *cname*
   from *depositor*)
   union
   (**select** *cname*
   from *borrower*)

4. **intersect**: Find customers having a loan **and** an account.

   (**select distinct** *cname*
   from *depositor*)
   intersect
   (**select distinct** *cname*
   from *borrower*)

5. **except**: Find customers having an account, but **not** a loan.

   (**select distinct** *cname*
   from *depositor*)
   except
   (**select** *cname*
   from *borrower*)

6. Some additional details:

   - **union** eliminates duplicates, being a set operation. If we want to retain duplicates, we may use **union all**, similarly for **intersect** and **except**.
   - Not all implementations of SQL have these set operations.
   - **except** in SQL-92 is called **minus** in SQL-86.
   - It is possible to express these queries using other operations.

4.4 Aggregate Functions

1. In SQL we can compute functions on groups of tuples using the **group by** clause. Attributes given are used to form groups with the same values. SQL can then compute

   - average value — **avg**
   - minimum value — **min**
   - maximum value — **max**
   - total sum of values — **sum**
   - number in group — **count**

   These are called **aggregate functions**. They return a single value.

2. Some examples:

   (a) Find the average account balance at each branch.

   select *bname*, **avg** (*balance*)
   from *account*
   group by *bname*

   (b) Find the number of depositors at each branch.

   select *bname*, **count (distinct** *cname*)
   from *account, depositor*
   where *account.account# = depositor.account#*
   group by *bname*

   We use **distinct** so that a person having more than one account will not be counted more than once.

   (c) Find branches and their average balances where the average balance is more than $1200.

   select *bname*, **avg** (*balance*)
   from *account*
   group by *bname*
   having avg (*balance*) $>$ 1200

   Predicates in the **having** clause are applied after the formation of groups.

   (d) Find the average balance of each customer who lives in Vancouver and has at least three accounts:

   select *depositor.cname*, **avg** (*balance*)
   from *depositor, account, customer*
   where *depositor.cname = customer.cname* **and** *account.account# = depositor.account#*
   and *ccity="Vancouver"*
   group by *depositor.cname*
   having count (distinct *account#*) $\geq$ 3

3. If a **where** clause and a **having** clause appear in the same query, the **where** clause predicate is applied first.

   - Tuples satisfying **where** clause are placed into groups by the **group by** clause.
   - The **having** clause is applied to each group.
   - Groups satisfying the having clause are used by the **select** clause to generate the result tuples.
   - If no **having** clause is present, the tuples satisfying the **where** clause are treated as a single group.

4.5 Null Values

1. With insertions, we saw how **null** values might be needed if values were unknown. Queries involving nulls pose problems.

2. If a value is not known, it cannot be compared or be used as part of an aggregate function.

3. All comparisons involving null are **false** by definition. However, we can use the keyword **null** to test for null values:

   select distinct *loan#*
   from *loan*
   where *amount* **is null**

4. All aggregate functions except **count** ignore tuples with null values on the argument attributes.

4.6 Nested Subqueries

4.6.1 Set Membership

1. We use the **in** and **not in** operations for set membership.

   select distinct *cname*
   from *borrower*
   where *cname* **in**
   (**select** *cname* **from** *account* **where** *bname*=*"SFU"*)

2. Note that we can write the same query several ways in SQL.

3. We can also test for more than one attribute:

   select distinct *cname*
   from *borrower, loan*
   where *borrower.loan# = loan.loan#* **and** *bname*=*"SFU"*
   and (*bname, cname*) **in**
   (**select** *bname, cname* **from** *account, depositor* **where** *depositor.account# = account.account#*)

   This finds all customers who have a loan and an account at the SFU branch in yet another way.

4. Finding all customers who have a loan but not an account, we can use the **not in** operation.

4.6.2 Set Comparison

1. To compare set elements in terms of inequalities, we can write

   select distinct *T.bname*
   from *branch T,branch S*
   where *T.assets* > *S.assets* **and** *S.bcity*=*"Burnaby"*

   or we can write

   select *bname*
   from *branch*
   where *assets* > **some**
   (**select** *assets* **from** *branch* **where** *bcity*=*"Burnaby"*)

   to find branches whose assets are greater than some branch in Burnaby.

2. We can use any of the equality or inequality operators with **some**. If we change > **some** to > **all**, we find branches whose assets are greater than all branches in Burnaby.

3. Example. Find branches with the highest average balance. We cannot compose aggregate functions in SQL, e.g. we cannot do **max (avg** ...**)**. Instead, we find the branches for which average balance is greater than

or equal to all average balances:

select *bname*
from *account*
group by *bname*
having avg (*balance*) $\geq$ **all**
(**select avg** (*balance*)
from *account*
group by *bname*)

4.6.3 Test for Empty Relations

1. The **exists** construct returns **true** if the argument subquery is nonempty.

2. Find all customers who have a loan **and** an account at the bank.

select *cname*
from *borrower*
where exists (**select** *
from *depositor*
where *depositor.cname = borrower.cname*)

4.6.4 Test for the Absence of Duplicate Tuples

1. The **unique** construct returns **true** if the argument subquery contains no duplicate tuples.

2. Find all customers who have only one account at the SFU branch.

select *T.cname*
from *depositor* **as** *T*
where unique (**select** *R.cname*
from *account, depositor* **as** *R*
where *T.cname = R.cname* **and**
R.account# = account.account# **and** *account.bname = "SFU"*)

4.7 Derived Relations

1. SQL-92 allows a subquery expression to be used in the **from** clause.

2. If such an expression is used, the result relation must be given a name, and the attributes can be renamed.

3. Find the average account balance of those branches where the average account balance is greater than $1,000.

select *bname, avg-balance*
from (**select** *bname*, **avg**(*balance*)
from *account*
group by *bname*)
as *result*(*bname, avg-balance*)
where *avg-balance > 1000*

4.7.1 Views

1. A view in SQL is defined using the **create view** command:

create view *v* **as** ⟨query expression⟩

where ⟨*query expression*⟩ is any legal query expression. The view created is given the name *v*.

2. To create a view *all-customer* of all branches and their customers:

   create view *all-customer* **as**
   (**select** *bname, cname*
   from *depositor, account*
   where *depositor.account# = account.account#*)
   union
   (**select** *bname, cname*
   from *borrower, loan*
   where *borrower.loan# = loan.loan#*)

3. Having defined a view, we can now use it to refer to the virtual relation it creates. View names can appear anywhere a relation name can.

4. We can now find all customers of the SFU branch by writing

   select *cname*
   from *all-customer*
   where *bname="SFU"*

4.8 Modification of the Database

Up until now, we have looked at extracting information from the database.

4.8.1 Deletion

1. **Deletion** is expressed in much the same way as a query. Instead of displaying, the selected tuples are removed from the database. We can only delete whole tuples.

2. A deletion in SQL is of the form

   delete from r
   where P

   Tuples in r for which P is true are deleted. If the where clause is omitted, all tuples are deleted.

3. The request **delete from** *loan* deletes **all** tuples from the relation *loan*.

4. Some more examples:

   (a) Delete all of Smith's account records.

   delete from *depositor*
   where *cname="Smith"*

   (b) Delete all loans with loan numbers between 1300 and 1500.

   delete from *loan*
   where *loan#* **between** 1300 **and** 1500

   (c) Delete all accounts at branches located in Surrey.

   delete from *account*
   where *bname* **in**
   (**select** *bname*
   from *branch*
   where *bcity="Surrey"*)

5. We may only delete tuples from one relation at a time, but we may reference any number of relations in a **select-from-where** clause embedded in the **where** clause of a **delete**.

6. However, if the delete request contains an embedded select that references the relation from which tuples are to be deleted, ambiguities may result.

   For example, to delete the records of all accounts with balances below the average, we might write

   delete from *account*
   where *balance* < (**select avg**(*balance*) **from** *account*)

   You can see that as we delete tuples from *account*, the average balance changes!

   Solution: The delete statement first test each tuple in the relation *account* to check whether the account has a balance less than the average of the bank. Then all tuples that fail the test are deleted. Perform all the tests (and **mark** the tuples to be deleted) before any deletion then delete them en masse after the evaluations!

4.8.2 Insertion

1. To insert data into a relation, we either specify a tuple, or write a query whose result is the set of tuples to be inserted. Attribute values for inserted tuples must be members of the attribute's domain.

2. Some examples:

   (a) To insert a tuple for Smith who has $1200 in account A-9372 at the SFU branch.

   insert into *account*
   values (*"SFU"*, "A-9372", 1200)

   (b) To provide each loan that the customer has in the SFU branch with a $200 savings account.

   insert into *account*
   select *bname, loan#, 200*
   from *loan*
   where *bname="SFU"*

   Here, we use a **select** to specify a set of tuples.

   It is important that we evaluate the **select** statement fully before carrying out any insertion. If some insertions were carried out even as the **select** statement were being evaluated, the insertion

   insert into *account*
   select *
   from *account*

   might insert an infinite number of tuples. Evaluating the **select** statement completely before performing insertions avoids such problems.

   (c) It is possible for inserted tuples to be given values on only some attributes of the schema. The remaining attributes are assigned a null value denoted by *null*.

   We can prohibit the insertion of null values using the SQL DDL.

4.8.3 Updates

1. Updating allows us to change some values in a tuple without necessarily changing all.

2. Some examples:

   (a) To increase all balances by 5 percent.

   update *account*
   set *balance=balance* * 1.05

   This statement is applied to every tuple in *account*.

(b) To make two different rates of interest payment, depending on balance amount:

update *account*
set *balance*=*balance* * 1.06
where *balance* > 10,000

update *account*
set *balance*=*balance* * 1.05
where *balance* $\leq$ 10,000

Note: in this example the order of the two operations is important. (Why?)

3. In general, **where** clause of **update** statement may contain any construct legal in a **where** clause of a **select** statement (including nesting).

4. A nested **select** within an update may reference the relation that is being updated. As before, all tuples in the relation are first tested to see whether they should be updated, and the updates are carried out afterwards.

   For example, to pay 5% interest on account whose balance is greater than average, we have

   update *account*
   set *balance*=*balance* * 1.05
   where *balance* > **select avg** (*balance*) **from** *account*

4.8.4 Update of a view

1. The view update anomaly previously mentioned in Chapter 3 exists also in SQL.

2. An example will illustrate: consider a clerk who needs to see all information in the *loan* relation except *amount*.

   Let the view *branch-loan* be given to the clerk:

   create view *branch-loan* **as**
   select *bname, loan#*
   from *loan*

   Since SQL allows a view name to appear anywhere a relation name may appear, the clerk can write:

   insert into *branch-loan*
   values (*"SFU"*, "L-307")

   This insertion is represented by an insertion into the actual relation *loan*, from which the view is constructed. However, we have no value for *amount*.

   This insertion results in (*"SFU"*, *"L-307"*, **null**) being inserted into the *loan* relation.

   As we saw, when a view is defined in terms of several relations, serious problems can result. As a result, many SQL-based systems impose the constraint that a modification is permitted through a view **only** if the view in question is defined in terms of **one** relation in the database.

4.9 Joined Relations

4.9.1 Examples

1. Two given relations: *loan* and *borrower*.

2. inner join:

   loan **inner join** *borrower* **on** *loan.loan# = borrower.loan#*

   Notice that the loan# will appear twice in the inner joined relation.

bname	loan#	amount
Downtown	L-170	3000
Redwood	L-230	4000
Perryridge	L-260	1700

cname	loan#
Jones	L-170
Smith	L-230
Hayes	L-155

Figure 4.1: The *loan* and *borrower* relations.

bname	loan#	amount	cname	loan#
Downtown	L-170	3000	Jones	L-170
Redwood	L-230	4000	Smith	L-230

Figure 4.2: Result of *loan* inner join *borrower*.

3. left outer join:

   loan **left outer join** *borrower* **on** *loan.loan# = borrower.loan#*

4. natural inner join:

   loan **natural inner join** *borrower*

4.9.2 Join types and conditions

1. Each variant of the join operations in SQL-92 consists of a *join type* and a *join condition*.

2. Join types: **inner join, left outer join, right outer join, full outer join**.

   The keyword **inner** and **outer** are optional since the rest of the join type enables us to deduce whether the join is an inner join or an outer join.

   SQL-92 also provides two other join types:

   (a) **cross join**: an inner join without a join condition.

   (b) **union join**: a full outer join on the "false" condition, i.e., where the inner join is empty.

3. Join conditions: **natural**, **on** predicate, **using** $(A_1, A_2, \ldots, A_n)$.

   The use of join condition is mandatory for outer joins, but is optional for inner joins (if it is omitted, a Cartesian product results).

4. Ex. A natural full outer join:

   loan **natural full outer join** *borrower*
   using (loan#)

bname	loan#	amount	cname	loan#
Downtown	L-170	3000	Jones	L-170
Redwood	L-230	4000	Smith	L-230
Perryridge	L-260	1700	*null*	*null*

Figure 4.3: Result of *loan* left outer join *borrower*.

bname	loan#	amount	cname
Downtown	L-170	3000	Jones
Redwood	L-230	4000	Smith

Figure 4.4: Result of *loan* natural inner join *borrower*.

bname	loan#	amount	cname
Downtown	L-170	3000	Jones
Redwood	L-230	4000	Smith
Perryridge	L-260	1700	*null*
null	L-155	*null*	Hayes

Figure 4.5: Result of *loan* natural full outer join *borrower* **using** (loan#).

5. Ex. Find all customers who have either an account or a loan (but not both) at the bank.

   select cname
   from (**natural full outer join** *borrower*)
   where *account#* **is** *null* **or** *loan#* **is** *null*

4.10 Data-Definition Language

The SQL DDL (Data Definition Language) allows specification of not only a set of relations, but also the following information for each relation:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints.
- The set of indices for each relation.
- Security and authorization information.
- Physical storage structure on disk.

4.10.1 Domain Types in SQL

1. The SQL-92 standard supports a variety of built-in domain types:
   - **char**(n) (or **character**(n)): fixed-length character string, with user-specified length.
   - **varchar**(n) (or **character varying**): variable-length character string, with user-specified maximum length.
   - **int** or **integer**: an integer (length is machine-dependent).
   - **smallint**: a small integer (length is machine-dependent).
   - **numeric**(*p, d*): a fixed-point number with user-specified precision, consists of p digits (plus a sign) and d of p digits are to the right of the decimal point. E.g., **numeric**(*3, 1*) allows 44.5 to be stored exactly but not 444.5.
   - **real** or **double precision**: floating-point or double-precision floating-point numbers, with machine-dependent precision.

- **float**(n): floating-point, with user-specified precision of at least n digits.
- **date**: a calendar date, containing four digit year, month, and day of the month.
- **time**: the time of the day in hours, minutes, and seconds.

2. SQL-92 allows arithmetic and comparison operations on various numeric domains, including, **interval** and *cast* (*type coercion*) such as transforming between *smallint* and *int*. It considers strings with different length are compatible types as well.

3. SQL-92 allows **create domain** statement, e.g.,

   create domain *person-name* **char**(20)

4.10.2 Schema definition in SQL

1. An SQL relation is defined by:

   create table r $(A_1, D_1, A_2, D_2, \ldots, A_n, D_n$
   $\langle$*integrity-constraint*$_1$ $\rangle$,
   $\ldots$, $\langle$*integrity-constraint*$_1$ $\rangle)$

   where r is the relation name, A_i is the name of an attribute, and D_i is the domain of that attribute. The allowed integrity-constraints include

   primary key $(A_{j1}, \ldots, A_{jm})$

   and

   check(P)

2. Example.

   create table *branch* (
   bname **char**(15) **not null**
   bcity **char**(30)
   assets **integer**
   primary key (*bname*)
   check (*assets* >= *0*))

3. The values of primary key must be *not null* and *unique*. SQL-92 consider **not null** in primary key specification is redundant but SQL-89 requires to define it explicitly.

4. Check creates type checking functionality which could be quite useful. E.g.,

   create table *student* (
   name **char**(15) **not null**
   student-id **char**(10) **not null**
   degree-level **char**(15) **not null**
   check (*degree-level* **in** ("Bachelors", "Masters", "Doctorate")))

5. Some checking (such as *foreign-key* constraints) could be costly, e.g.,

   check (*bname* **in** (**select** *bname* **from** *branch*))

6. A newly loaded table is empty. The **insert** command can be used to load it, or use special bulk loader untilities.

7. To remove a relation from the database, we can use the **drop table** command:

   drop table r

   This is not the same as

   delete r

   which retains the relation, but deletes all tuples in it.

8. The **alter table** command can be used to add or drop attributes to an existing relation r:

   alter table r **add** A D

   where A is the attribute and D is the domain to be added.

   alter table r **drop** A

   where A is the attribute to be dropped.

4.11 Embedded SQL

1. SQL provides a powerful declarative query language. However, access to a database from a general-purpose programming language is required because,
   - SQL is not as powerful as a general-purpose programming language. There are queries that cannot be expressed in SQL, but can be programmed in C, Fortran, Pascal, Cobol, etc.
   - Nondeclarative actions — such as printing a report, interacting with a user, or sending the result to a GUI — cannot be done from within SQL.
2. The SQL standard defines embedding of SQL as *embedded SQL* and the language in which SQL queries are embedded is referred as *host language*.
3. The result of the query is made available to the program one tuple (record) at a time.
4. To identify embedded SQL requests to the preprocessor, we use EXEC SQL statement:

   EXEC SQL ⟨embedded SQL statement ⟩END-EXEC

   Note: A semi-colon is used instead of END-EXEC when SQL is embedded in C or Pascal.
5. Embedded SQL statements: **declare cursor**, **open**, and **fetch** statements.

   EXEC SQL
   declare *c* **cursor for**
   select *cname, ccity*
   from *deposit, customer*
   where *deposit.cname* = *customer.cname* **and** *deposit.balance* > *:amount*
   END-EXEC

   where *amount* is a host-language variable.

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

   This statement causes the DB system to execute the query and to save the results within a temporary relation.

   A series of **fetch** statement are executed to make tuples of the results available to the program.

   EXEC SQL **fetch** *c* **into** *:cn, :cc* END-EXEC

   The program can then manipulate the variable *cn* and *cc* using the features of the host programming language.

   A single **fetch** request returns only one tuple. We need to use a **while** loop (or equivalent) to process each tuple of the result until no further tuples (when a variable in the SQLCA is set).

   We need to use **close** statement to tell the DB system to delete the temporary relation that held the result of the query.

   EXEC SQL **close** *c* END-EXEC
6. Embedded SQL can execute any valid **update**, **insert**, or **delete** statements.
7. *Dynamic* SQL component allows programs to construct and submit SQL queries ar run time (see p. 147 of the textbook for details).
8. SQL-92 also contains a *module* language, which allows procedures to be defined in SQL (see pp. 147-148 of the textbook for details).

4.12 Other SQL Features

1. 4GL: Most commercial database products include a special language to assist application programmers in creating templates on the screen for a user interface, and in formatting data for report generating.

   No single accepted standard currently exists for 4GL.

2. SQL-92 standard defined SQL sessions and SQL environments.

   - SQL sessions: client/server abstraction (connect, disconnect, commit, rollback).
   - SQL environments: provide user-id and schema for each user.