Chapter 4: SQL

- Embedded SQL
- Data Definition Language
- Jointed Relations
- Modification of the Database
- Views
- Derived Relations
- Nested Subqueries
- Null Values
- Aggregate Functions
- Set Operations
- Basic Structure
The result of an SQL query is a relation.

\[
((\Pi_{w_1} \times \cdots \times \Pi_{w_2} \times I_{d_0})^u A_1, \ldots, A_2)\]  

This query is equivalent to the relational algebra expression:

\[
\text{WHERE } \quad p \quad \text{IS A PREDICATE.} \\
\text{FROM } \quad \Pi_{w_1} \times \cdots \times \Pi_{w_2} \times I_{d_0} \quad \text{REPRESENT RELATIONS.} \\
\text{SELECT } \quad A_1, \ldots, A_2 \quad \text{REPRESENT ATTRIBUTES.} \\
\]

A typical SQL query has the form:

```
SELECT A1, A2, ... FROM W1, W2, ... WHERE P
```

SQL is based on set and relational operations with certain modifications and enhancements.

Basic Structure
**The Select Clause**

```sql
FROM loan
SELECT *
```

An asterisk in the select clause denotes "all attributes"

```sql
IF branch-name (loan)
```

In the "pure" relational algebra syntax, this query would be:

```sql
FROM loan
SELECT branch-name
```

Find the names of all branches in the loan relation

The result of a query in the relational algebra. It is used to list the attributes desired in the select clause corresponds to the projection operation of
from loan
select all branch-name

The keyword all species that duplicates not be removed.

from loan
select distinct branch-name
duplicates

Find the names of all branches in the loan relation, and remove

distinct after select.

To force the elimination of duplicates, insert the keyword

SQL allows duplicates in relations as well as in query results.

The Select Clause (cont.)
Except that the attribute amount is multiplied by 100, would return a relation which is the same as the loan relation.

```
FROM loan
SELECT branch-name, loan-number, amount * 100
```
The where Clause

- The **where** clause corresponds to the selection predicate of the relational algebra. It consists of a predicate involving attributes of the relations that appear in the **from** clause.
- Find all loan numbers for loans made at the Perryridge branch with loan amounts greater than $1200.

```sql
select loan-number
from loan
where branch-name = "Perryridge" and amount > 1200
```

- SQL uses the logical connectives **and**, **or**, and **not**. It allows the use of arithmetic expressions as operands to the comparison operators.
where amount between 900,000 and 1,000,000

From loan

select loan-number

• $100,000 ≥

between $90,000 and $100,000 (that is, $90,000 and

equal to some value and greater than or equal to some other

value. Simplify where clause that specifies that a value be less than or

than 0.1$ includes a between comparison operator in order to

The where clause (cont.)
branch-name = "Perryridge"

where borrower.loan-number = loan.loan-number and
from borrower, loan

select distinct customer-name, borrower.loan-number
at the Perryridge branch.

Find the name and loan number of all customers having a loan
from borrower, loan

* select

Find the Cartesian product borrower x loan

included in the evaluation of the expression.
operation of the relational algebra. It lists the relations to be
The from clause corresponds to the Cartesian product

The from Clause
"rename-name = "Periyrige"

where borrower.loan-number = loan.loan-number and

from borrower, loan

select distinct customer-name, borrower.loan-number as loan-id

loan-number with the name loan-id.

at the Periyrige branch, replace the column name
find the name and loan number of all customers having a loan

old-name as new-name

accomplished through the as clause:

The SQL mechanism for renaming relations and attributes is

The Rename Operation
WHERE T.asssets < S.asssets and S.branch-city = „Brooklyn“

from branch as T branch as S
select distinct T.branch-name

some branch located in Brooklyn.
Find the names of all branches that have greater assets than

WHERE T.loan-number = S.loan-number

from borrower as T loan as S
select distinct customer-name, T.loan-number

customers having a loan at some branch.
Find the customer names and their loan numbers for all

the as clause.
Tuple variables are defined in the from clause via the use of

Tuple Variables
like "Main\%\" escape "Main\%

• Match the name "Main%"

where customer-street like "%Main%"
from customer
select customer-name

substring 'Main'.

• Find the names of all customers whose street includes the
underscore (_) - the underscore matches any character.

• Percent (%) - The % character matches any substring.

characters: character strings. Patterns are described using two special
character string patterns included a string-matching operator for comparisons on

String Operations

SQL.
sort only when necessary.

sorting a large number of tuples may be costly, it is desirable to

SQL must perform a sort to fulfill an order by request. Since

order, for each attribute, ascending order is the default.

We may specify desc for descending order or asc for ascending

order by customer-name

branch-name = "Perryridge"

where borrower.loan-number = loan.loan-number

from borrower, loan

select distinct customer-name

loan at Perryridge branch

List in alphabetical order the names of all customers having a

Ordering the Display of Tuples
1. Given multiset relations \( r_1 \) and \( r_2 \):

2. Multiset versions of some of the relational algebra operators:

3. If there are \( c_1 \times c_2 \) copies of tuple \( t_1 \) in \( r_1 \) and \( c_2 \) copies of tuple the single tuple \( t_1 \).

In relations with duplicates, SQL can define how many copies of tuples appear in the result.
\[(u \times \cdots \times \varpi \times \mathfrak{I})_{\mathcal{D}} \exists \forall \, \cdots, \forall \mathfrak{I}, \forall \varpi, \forall \mathfrak{I}, \forall \varpi\]

is equivalent to the multiset version of the expression:

where

\[\forall \mathfrak{I}, \forall \varpi, \forall \mathfrak{I}, \forall \varpi \]

from \(\mathfrak{I}, \forall \varpi, \forall \mathfrak{I}, \forall \varpi\)

select \(\forall \mathfrak{I}, \forall \varpi, \forall \mathfrak{I}, \forall \varpi\)

\[\exists \mathfrak{I}, \exists \varpi \]

SOL duplicate semantics:

\[
\{(v, (v), (v), (v), (v), (v), (v), (v))\}
\]

where \(\varpi \times (\mathfrak{I}, \mathfrak{I})_{\mathcal{D}} \exists \mathfrak{I}, \exists \varpi\) would be

\[
\{(v, (v), (v), (v)) = \varpi, \{v, (v), (v)\} = \mathfrak{I}\}
\]

\[\exists \mathfrak{I}, \exists \varpi \]

are the following multisets:

Suppose relations \(\mathfrak{I}, \varpi\) with schema \(\mathcal{D}, \forall \mathfrak{I}, \forall \varpi\)

Duplicates (Cont.)
Suppose a tuple occurs in times in \( r \) and \( n \) times in \( s \), then, it

- occurs

mutiple versions union all, intersect all, and except all.

Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding

\[ \land \]

relations and correspond to the relational algebra operations.

The set operations union, intersect, and except operate on
count: number of values
sum: sum of values
max: maximum value
min: minimum value
ave: average value
relation, and return a value
These functions operate on the multiset of values of a column of a
from depositor
select count (distinct customer-name)

Find the number of depositors in the bank.

from customer
select count (*)

Find the number of tuples in the customer relation.

where branch-name = "Perryridge"
from account
select avg (balance)

Find the average account balance at the Perryridge branch.

Aggregate Functions (Cont.)
must appear in Group by List.

Note: Attributes in select clause outside of aggregate functions

Group by branch-name

where depositor.account = account.account-number

from depositor, account

select branch-name, count (distinct customer-name)

• Find the number of depositors for each branch.
Note: predicates in the **having** clause are applied after the **group by** specification.

```sql
select branch-name, avg(balance)
from account
having avg(balance) < 1200

balance is more than $1,200
```

Find the names of all branches where the average account balance is more than $1,200.
null

Values

null

The result of any arithmetic expression involving null is null.

that a value does not exist.

For some of their attributes, null signifies an unknown value or

It is possible for tuples to have a null value, denoted by null.
null values on the aggregated attributes.

All aggregate operations except count(*) ignore tuples with
no non-null amount.

Above statement ignores null amounts; result is null if there is

\[
\text{from loan}
\text{select sum(amount)}
\text{where amount is null}
\text{from loan}
\text{select loan-number}
\text{find all loan numbers which appear in the loan relation with}
\]

**null values (count).**
A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.

• Within another query, a subquery is a select-from-where expression that is nested in SQL to provide a mechanism for the nesting of subqueries.
true = (\begin{pmatrix} 0 \\ \not \in \end{pmatrix}) \qquad (\not \in \in \not \in)

false = (\begin{pmatrix} 0 \\ \not \in \end{pmatrix}) \qquad (\not \in \in \not \in)

true = (\begin{pmatrix} 0 \\ \in \end{pmatrix}) \qquad (\in \in \not \in)

\text{Set Membership}
(from depositor)
where customer-name not in (select customer-name
from borrower
select distinct customer-name
have an account at the bank.

Find all customers who have a loan at the bank but do not

(from depositor)
where customer-name in (select customer-name
from borrower
select distinct customer-name
bank.

Find all customers who have both an account and a loan at

Example Query
account.account-number = where depositor.account-number 
from depositor, account
)
select branch-name, customer-name (in
branch-name = "Perryridge"
and
branch-name = "Perryridge"
and
where borrower.loan-number = loan.loan-number
from borrower, loan
select distinct customer-name
Perryridge branch.

• Find all customers who have both an account and a loan at the

Example Query
S.branch-city = "Brooklyn"

where \( S.\text{assets} < S.\text{assets} \text{ and} \)

from branch as \( S \text{, branch as} \ S \)

select distinct \( S.\text{branch-name} \)

located in Brooklyn.

Find all branches that have greater assets than some branch

Set Comparison
\[ \text{not in } \not\equiv (\text{some } \not\equiv) \]

\[ \text{in } \equiv (\text{some } =) \]

\[ (\not\equiv 0 \text{ since } \text{true} = (\begin{array}{c} \text{some} \\ \not\equiv \end{array}) \]

\[ \text{true} = (\begin{array}{c} \not\equiv \\ \text{some} = \not\equiv \\ \text{false} = (\begin{array}{c} \not\equiv \\ \text{some} > \not\equiv \\
\end{array}) \]

\[ \text{(read: } \not\equiv \text{ some tuple in the relation) } \]

\[ \text{true} = (\begin{array}{c} \not\equiv \\ \text{some} > \not\equiv \\
\end{array}) \]

\[ \neq', ' =', '<', ' \subseteq', '<\subseteq', ' >' \text{ can be: } \]

\[ (\left[ t \right] \text{ comp } F) \forall t \in \text{ some } \not\equiv \]

\[ \text{The Some Clause} \]
(where branch-city = "Brooklyn")

from branch
select assets

where assets < some
from branch
select branch-name

located in Brooklyn.

Find all branches that have greater assets than some branch.
in \neq (\forall \text{ not } in) •

\begin{align*}
(\neq \text{ true}) &= (\begin{array}{c|c}
6 & \neq \\
\hline
4 & \neq  \\
\end{array}) \\
\text{false} &= (\begin{array}{c|c}
5 & \neq \\
\hline
4 & \neq \\
\end{array}) \\
\text{true} &= (\begin{array}{c|c}
10 & \neq  \\
\hline
9 & \neq \\
\end{array}) \\
\text{false} &= (\begin{array}{c|c}
6 & \neq \\
\hline
5 & \neq \\
0 & \neq \\
\end{array})
\end{align*}

([t \text{ comp} \neq R] \lor t \in t) \iff t \text{ comp} \neq R •
(where branch-city = "Brooklyn")

from branch

select assets

where assets < all

from branch

select branch-name

branches located in Brooklyn.

Find the names of all branches that have greater assets than all
\[ \emptyset = \not \exists \in \iff \emptyset \not= \not \exists \in \iff \exists \not\emptyset \exists \not\emptyset \iff \text{subquery is nonempty.} \]

\[ \text{The exists construct returns the value true if the argument} \]

\[ \text{Test for Empty Relations} \]
\[
\forall X \subseteq \emptyset = \forall \neg X
\]

Note that \( X \subseteq \emptyset = \forall \neg X \)

\[
\text{S.customer-name = } \text{L.customer-name}
\]

where \( \text{L.account-number} = \text{R.account-number} \)

from \( \text{depositor as } L \text{ account as } R \)

select \( R \text{ branch-name} = \text{Customer Name} \)

except

where \( \text{branch-city} = \text{Brooklyn} \)

from \( \text{branch} \)

select \( \text{branch-name} = \text{Customer Name} \)

where not exists

from \( \text{depositor as } S \)

select distinct \( S \text{ customer-name} = \text{Customer Name} \)

in \( \text{Brooklyn} \).

Find all customers who have an account at all branches located in \( \text{Brooklyn} \).
account.branch-name = "Perryridge"

R.account-number = account.account-number

where T.customer-name = R.customer-name

from account, deposit as R

select R.customer-name

where unique

from deposit as T

select T.customer-name

Perryridge branch.

Find all customers who have only one account at the
duplicate tuples in its result.

The unique construct tests whether a subquery has any

Test for Absence of Duplicatet Tuples
account.branch-name = "Perryridge"

R.account-number = account.account-number

where T.customer-name = R.customer-name

from account, depositor as R

select R.customer-name

where not unique

from depositor T

select distinct T.customer-name

Perryridge branch.

Find all customers who have at least two accounts at the

Example Query
the attributes of result can be used directly in the WHERE clause in the FROM clause the temporary relation result, and compute in the FROM clause since we do not need to use the HAVING clause. Note that we do not need to use the HAVING clause since we do not need to use the HAVING clause. Since we do not need to use the HAVING clause, since we do not need to use the HAVING clause.

\[
\text{where } \text{avg-balance} < 1200
\]

\[
\text{as result } \text{branch-name, avg-balance}
\]

\[
\text{group by branch-name}
\]

\[
\text{from account}
\]

\[
\text{from select } \text{branch-name, ave (balance)}
\]

\[
\text{select } \text{branch-name, avg-balance}
\]

average account balance is greater than $1200.

Find the average account balance of those branches where the average account balance is greater than $1200.

**Derived Relations**
the view name is represented by

<query_expression> is any legal expression

where

<create_view as query_expression>

To create a view we use the command: •
where branch-name = "Perryridge"
from all-customer
select customer-name

* Find all customers of the Perryridge branch

( where borrower.loan-number = loan.loan-number
  from borrower, loan
  select branch-name, customer-name
) union

( where depositor.account-number = account.account-number
  from depositor, account
  select branch-name, customer-name
) create view all-customer as

* A view consisting of branches and their customers

Example Queries
and branch.branch-name = account.branch-name

where branch.city = "Needham"
from branch, account
where account.number in (select account.number

delete from deposition

where branch.city = "Needham"
from branch

where branch-name in (select branch-name

delete from account

Delete all accounts at every branch located in Needham.

Delete all accounts at every branch located in Needham.  

where branch-name = "Perryridge"
from account

Delete all account records at the Perryridge branch

Modification of the Database – Deletion
1. First, compute average balance and find all tuples to delete
   - Solution used in SQL:
   ```sql
   SELECT balance_changes
   FROM account
   WHERE balance > (SELECT AVG(balance) FROM account)
   DELETE FROM account
   ```

2. Next, delete all tuples found above (without recomputing

Example Query
values ("Permanent", A-777, null)

\textbf{Insert into account}

Add a new tuple to account with balance set to null

values ("Permanent", 1200, A-9732)

\textbf{Insert into account (branch-name, balance, account-number)}
or equivalently

values ("Permanent", 1200)

\textbf{Insert into account}

Add a new tuple to account

\textbf{Modification of the Database – Insertion}
and loan.account-number = borrower.account-number

where branch-name = "Perryridge"

from loan

select customer-name, loan-number

insert into depositor

where branch-name = "Perryridge"

from loan

select branch-name, loan-number

insert into account

the account number for the new savings account branch, a $200 savings account. Let the loan number serve as

provide as a gift for all loan customers of the Perryridge

•
Exercise 4.11

- Can be done better using the `case` statement.

- The order is important.

```sql
WHERE balance >= 10000
  SET balance = balance * 1.05
  UPDATE account;

WHERE balance < 10000
  SET balance = balance * 1.06
  UPDATE account;
```

- Write two update statements:

  
  - Other accounts receive 5%.
  - Increase all accounts with balances over $10,000 by 6%.

Modification of the Database – Updates
Updates on more complex views are difficult or impossible to translate, and hence are disallowed.

- Updates into the loan relation.

This insertion must be represented by the insertion of the tuple:

\[(\text{"Footer"}, \text{"F-307"}, \text{null})\]

Insert into branch-loan

Add a new tuple to branch-loan

From loan

Select branch-name, loan-number, loan-name

Create view branch-loan as

amount attribute

Create a view of all loan data in the loan relation, hiding the

Update of a View
<table>
<thead>
<tr>
<th>Join Types</th>
<th>Join Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>natural</em></td>
<td>using ( A_1, A_2, \ldots, A_n ) \ on &lt;predicate&gt;</td>
</tr>
<tr>
<td><em>inner</em></td>
<td>Full outer join</td>
</tr>
<tr>
<td><em>left</em></td>
<td>Right outer join</td>
</tr>
<tr>
<td><em>right</em></td>
<td>Left outer join</td>
</tr>
</tbody>
</table>

- Join condition (are treated) match any tuple in the other relation (based on the join condition) and what attributes are present in the result of the join.
- Join type - defines how tuples in each relation that do not match, and what attributes are present in the result of the join.
- Join condition - defines which tuples in the two relations expressions in the *from* clause.
- *These additional operations are typically used as subqueries another relation.*
- *Join operations take two relations and return as a result.*
### Relation borrower

<table>
<thead>
<tr>
<th>Customer-name</th>
<th>Loan-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hayes</td>
<td>L-155</td>
</tr>
<tr>
<td>Smith</td>
<td>L-230</td>
</tr>
<tr>
<td>Jones</td>
<td>L-170</td>
</tr>
</tbody>
</table>

### Relation loan

<table>
<thead>
<tr>
<th>Branch-name</th>
<th>Loan-number</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perriville</td>
<td>L-260</td>
<td>1700</td>
</tr>
<tr>
<td>Redwood</td>
<td>L-230</td>
<td>4000</td>
</tr>
<tr>
<td>Downtown</td>
<td>L-170</td>
<td>3000</td>
</tr>
</tbody>
</table>

**Joined Relations – Datasets for Examples**
<table>
<thead>
<tr>
<th>Loan-number</th>
<th>Branch-name</th>
<th>Amount</th>
<th>Loan-number</th>
<th>Branch-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>null</td>
<td>1700</td>
<td>T-260</td>
<td>Portland</td>
</tr>
<tr>
<td>T-230</td>
<td>Smith</td>
<td>4400</td>
<td>T-230</td>
<td>Redwood</td>
</tr>
<tr>
<td>T-170</td>
<td>Jones</td>
<td>3000</td>
<td>T-170</td>
<td>Downtown</td>
</tr>
</tbody>
</table>

```
loan.loan-number=borrower.loan-number
loan left outer join borrower

loan.loan-number=borrower.loan-number
loan inner join borrower
```

**Joined Relations – Examples**
<table>
<thead>
<tr>
<th>customer-name</th>
<th>amount</th>
<th>loan-number</th>
<th>branch-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hayes</td>
<td>null</td>
<td>T-155</td>
<td>null</td>
</tr>
<tr>
<td>Smith</td>
<td>4000</td>
<td>T-230</td>
<td>Redwood</td>
</tr>
<tr>
<td>Jones</td>
<td>3000</td>
<td>T-170</td>
<td>Downtown</td>
</tr>
</tbody>
</table>

#### Loan natural right outer join borrower

<table>
<thead>
<tr>
<th>customer-name</th>
<th>amount</th>
<th>loan-number</th>
<th>branch-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>4000</td>
<td>T-230</td>
<td>Redwood</td>
</tr>
<tr>
<td>Jones</td>
<td>3000</td>
<td>T-170</td>
<td>Downtown</td>
</tr>
</tbody>
</table>

#### Loan natural inner join borrower

**Joined Relations - Examples**
Find all customers who have either an account or a loan (but not both) at the bank.

<table>
<thead>
<tr>
<th></th>
<th>customer-name</th>
<th>amount</th>
<th>loan-number</th>
<th>branch-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hayes</td>
<td>null</td>
<td>null</td>
<td>T-155</td>
<td>null</td>
</tr>
<tr>
<td>null</td>
<td>1700</td>
<td>1-260</td>
<td>T-230</td>
<td>Perryridge</td>
</tr>
<tr>
<td>Smith</td>
<td>4000</td>
<td>1-210</td>
<td>T-170</td>
<td>Redwood</td>
</tr>
<tr>
<td>Jones</td>
<td>3000</td>
<td></td>
<td></td>
<td>Downtown</td>
</tr>
</tbody>
</table>

**Joined Relations – Examples**

- `WHERE account-number is null OR loan-number is null`
The physical storage structure of each relation.

- Security and authorization information for each relation.
- The set of indices to be maintained for each relation.
- Integrity constraints.
- The domain of values associated with each attribute.
- The schema for each relation.

Information about each relation, including:

- Allows the specification of not only a set of relations but also...
precision of \( d \) digits, with \( n \) digits to the right of decimal point.

- \texttt{numeric}(p,d) \quad \text{Fixed point number, with user-specified integer domain type.}
- \texttt{smallint} \quad \text{Small integer (a machine-dependent subset of the machine-dependent).}
- \texttt{int} \quad \text{Integer (a finite subset of the integers that is user-specified maximum length \( n \)).}
- \texttt{varchar}(n) \quad \text{Variable length character string, with length \( n \).}
- \texttt{char}(n) \quad \text{Fixed length character string, with user-specified length \( n \).}
create domain person-name char(20) not null

domain types - create domain construct in SQL-92 creates user-defined attribute to be not null prohibits null values for that attribute.

- Null values are allowed in all the domain types. Declaring an

  - time. Time of day, in hours, minutes and seconds.
  - date. Dates, containing a (4 digit) year, month and date.
    of at least n digits.

  - float(n). Floating point number with user-specified precision.

  - real, double precision. Floating point numbers, with machine-dependent precision.

Domain Types in SQL (Cont).
Example:

- \( \text{create table branch} \)

\[ \text{create table branch} ( A_1, A_2, \ldots, A_n ) \]

An SQL relation is defined using the \textit{create table} command:
ensures not null in SQL-92

primary key declaration on an attribute automatically

\[
\text{check (assets} \leq 0) \text{ }
\]

primary key (branch-name),

\[
\text{assets integer,}
\]

branch-name char(30),

\[
\text{branch-city char(15) not null,}
\]

create table branch

e nsure that the values of assets are non-negative.

Example: Declare branch-name as the primary key for branch and

\[
\text{check (P), where P is a predicate}
\]

\[
\text{primary key (A_1, \ldots, A_n)}
\]

not null

Integrity Constraints In Create Table
Where $A$ is the name of an attribute of relation $R$.

**Alter Table** drop $A$

of a relation

The **alter table** command can also be used to drop attributes

and and is the domain of $A$.

Where $A$ is the name of the attribute be added to Relation $R$

**Alter Table** add $A$

command is

the value for the new attribute. The form of the **alter table**

command is used to add attributes to an existing relation. All tuples in the relation are assigned null as

dropped relation from the database.

The drop table command deletes all information about the

---

**Drop and Alter Table Constructs**
EXEC SQL <embedded SQL statement > END EXEC

Requests to the preprocessor

EXEC SQL statement is used to identify embedded SQL

Embedding of SQL into PL/I

The basic form of these languages follows that of the System R

Languages comprise embedded SQL.

A host language, and the SQL structures permitted in the host

A language in which SQL queries are embedded is referred to as

C, and Cobol.

Programming languages such as such as Pascal, PL/I, Fortran,

The SQL standard defines emebddings of SQL in a variety of

Embedded SQL
and account.balance < amount

where deposition.deposition-number = account.account-number

from deposition, account

select customer-name, account-number

declare c cursor for

declare s glo

EXEC SQL

EXEC SQL

• Specify the query in SQL and declare a cursor for it

account.

FROM within a host language, find the names and account numbers

of customers with more than the variable amount dollars in some

Example Query
EXEC SQL close c END-EXEC

Temporary relation that holds the result of the query. The close statement causes the database system to delete the

•

When end-of-file is reached.

Result: a variable in the SQL communication area indicates repeated calls to fetch get successive tuples in the query.

EXEC SQL fetch c into :cn :an END-EXEC

•

Query result to be placed in host language variables.

The fetch statement causes the values of one tuple in the

EXEC SQL open c END-EXEC

•

The open statement causes the query to be evaluated.

Embedded SQL (Cont.)
The dynamic SQL program contains a placeholder for a value that is provided when the SQL program is executed.

```sql
EXEC SQL execute dynamic using account;

EXEC SQL prepare dynamic from

where account-number = ?

EXEC SQL execute

where

EXEC SQL prepare dynamic from

```

Example of the use of dynamic SQL from within a C program.

```

char * sqlquery = "update account set balance = balance * 1.05"
```

Allows programs to construct and submit SQL queries at run time.
schema a session is using. user identifier, and a schema, which identifies which of several
An SQL environment contains several components, including a
-- can commit or rollback the work carried out in the session
-- disconnects the session
-- executes a series of statements
-- client connects to an SQL server, establishing a session
(possibly remote)
SQL sessions -- provide the abstraction of a client and a server
genization; available in most comerial database products
for a user interface, and in formatting data for report
application programmers in creating templates on the screen
Fourth-generation languages -- Special language to assist

Other SQL Features