4. SQL

Contents

- Basic Queries in SQL (**select** statement)
- Set Operations on Relations
- Nested Queries
- Null Values
- Aggregate Functions and Grouping
- Data Definition Language Constructs
- Insert, Update, and Delete Statements
- Views (Virtual Tables)

Example Database

CUSTOMERS(FName, LName, CAddress, Account)

PRODUCTS(Prodname, Category)

SUPPLIERS(SName, SAddress, Chain)

orders((FName, LName) $\rightarrow$ CUSTOMERS, SName $\rightarrow$ SUPPLIERS,

Prodname $\rightarrow$ PRODUCTS, Quantity)

offers(SName $\rightarrow$ SUPPLIERS, Prodname $\rightarrow$ PRODUCTS, Price)

Basic Structure

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

  In this course we focus on SQL (≈ SQL Standard) but also do some PostgreSQL specifics later

- A typical SQL query has the form

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

  - A_is represent attributes
  - r_is represent relations
  - P is a predicate

- This query is equivalent to the relational algebra expression

  $\pi_{A_1,A_2,\ldots,A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_k))$

- The result of an SQL query is a relation (set of tuples) with a schema defined through the attributes A_is.

- The **select** clause corresponds to the projection operation of the relational algebra; it is used to list the attributes to be output in a query result.

  Find the name of all suppliers.

  select SName **from** SUPPLIERS;

  $\rightarrow \pi_{\text{SName}}(\text{SUPPLIERS})$

Basic Structure (cont.)

- An asterisk "$*$" in the **select** clause denotes all attributes

  select $*$ **from** SUPPLIERS;

- SQL allows duplicate tuples in a relation as well as in query results. Duplicates can be removed from query result using keyword **distinct**

  select distinct Account **from** CUSTOMERS;

- **select** clause can contain arithmetic expressions as well as functions on attributes including attributes and constants.

  select substr(SName,1,10) [**as**] "Name", Prodname, Price $*$ 100
  from offers;

- The **where** clause corresponds to the selection operation of the relational algebra. It consists of a predicate involving attributes of the relations that appear in the **from** clause.

  List the first and last name of customers having a negative account.

  select FName, LName
  from CUSTOMERS
  where Account $<$ 0;

Basic Structure (cont.)

- Logical connectives **and**, **or**, and **not** can be used to formulate complex condition in **where** clause.

  Which suppliers (SName) offer a MegaPC or a TinyMac?

  select SName **from** offers
  where Prodname = 'MegaPC' **or** Prodname = 'TinyMac';

  $\hat{=}$ . . . **where** Prodname **in** ('MegaPC','TinyMac')

  List the name of products that cost more than $10,000 and less than $20,000.

  select Prodname, Price **from** offers
  where Price >= 10000 **and** Price <= 20000;

  $\hat{=}$ . . . **where** Price **between** 10000 **and** 20000

- The **from** clause corresponds to the Cartesian Product of the relational algebra.

  List all customer with the products they can order.

  select $*$ **from** CUSTOMERS, PRODUCTS;

Basic Structure (cont.)

List all customers who are living in Davis and who have ordered at least 10 MegaPCs.

select CUSTOMERS.FName, CUSTOMERS.LName, Quantity
from CUSTOMERS, orders
where CAddress **like** '%Davis%'
and CUSTOMERS.FName = orders.FName
and CUSTOMERS.LName = orders.LName
and Prodname = 'MegaPC' **and** Quantity $\geq$ 10;

$\pi_{\text{CUSTOMERS.FName, CUSTOMERS.LName, Quantity}}$
$(\sigma_{\text{CAddress like '\%Davis\%'} \wedge \text{Quantity} \geq 10 \wedge \text{Prodname='MegaPC'}}$
$(\sigma_{\text{CUSTOMERS.FName=orders.FName} \wedge \text{CUSTOMERS.LName=orders.LName}}$
$(\text{CUSTOMERS} \times \text{orders})))$

Replace the last selection condition $\sigma_{\ldots}$ by a natural join

$(\text{CUSTOMERS} \bowtie \text{orders})$

List the name and address of suppliers that offer products. Remove duplicates from the result and list the result ordered by the supplier's address.

select distinct SUPPLIERS.SName, SAddress
from SUPPLIERS, offers
where SUPPLIERS.SName = offers.SName
order by SAddress;

Basic Structure (cont.)

- Using the rename operator (*aliasing*)

  select distinct S.SName, SAddress
  from SUPPLIERS S, offers O
  where S.SName = O.SName;

  List all information about customers together with information about the suppliers they have ordered products from.

  select C.∗, S.∗, O.∗
  from CUSTOMERS C, orders O, SUPPLIERS S
  where C.LName = O.LName **and** C.FName = O.FName
  and O.SName=S.SName;

  Equivalent expression in relational algebra:

  $((\text{CUSTOMERS} \bowtie \text{orders}) \bowtie \text{SUPPLIERS})$

  List the name of customers who have an account greater or equal than (some) other customers.

  select C1.FName, C1.LName
  from CUSTOMERS C1, CUSTOMERS C2
  where (C1.FName <> C2.FName **or**
  C1.LName <> C2.LName)
  and C1.Account >= C2.Account;

  ☞ query realizes a condition join!

Set Operations

- The SQL set operations **union**, **except** (aka **minus** in Oracle), and **intersect** correspond to the relational algebra operations $\cup$, $-$, and $\cap$.

- Each of the above operations automatically eliminates duplicates. To retain duplicates, one has to use the corresponding multiset versions **union all**, **except all**, **intersect all**.

- Examples:

  Find all suppliers that offer a MegaPC or TinyMac.

  (**select** SName **from** offers **where** Prodname = 'MegaPC')
  union
  (**select** SName **from** offers **where** Prodname = 'TinyMac');

  Find all suppliers that offer both a MegaPC and a TinyMac.

  (**select** SName **from** offers **where** Prodname = 'MegaPC')
  intersect
  (**select** SName **from** offers **where** Prodname = 'TinyMac');

  Find all suppliers that offer a MegaPC but not a TinyMac.

  (**select** SName **from** offers **where** Prodname = 'MegaPC')
  except **minus**
  (**select** SName **from** offers **where** Prodname = 'TinyMac');

Nested Subqueries

- So far, **where** clauses in examples only consist of simple attribute and/or constant comparisons.
- SQL provides language constructs for the nesting of queries using subqueries. A *subquery* is a **select-from-where** expression that is nested within another query.
- Most common use of subqueries is to perform tests for *set membership*, *set comparisons*, and *set cardinality*.
- *Set valued* subqueries in a **where** condition:
  - <expression> [**not**] **in** (<subquery>)
  - <expression> <comparison operator> **any** (<subquery>)
  - <expression> <comparison operator> **all** (<subquery>)
- *Set cardinality* or test for (non-)existence:
  - [**not**] **exists** (<subquery>)
- Subqueries in a **where** clause can be combined arbitrarily using logical connectives.

Examples of Set Valued Subqueries

- *Give the name and chain of all suppliers located in Davis that offer a MegaPC for less than $1,000.*

```
select SName, Chain
from SUPPLIERS
where SName in (select SName from offers
                where Prodname = 'MegaPC'
                and Price < 1000)
and SAddress like '%Davis%';
```

☞ This query can also be formulated using a join!

- *Give the name and address of suppliers that don't offer a MegaPC.*

```
select SName, SAddress
from SUPPLIERS
where SName not in (select SName from offers
                    where Prodname = 'MegaPC');
```

☞ If it is known that a subquery returns at most one value, then one can use "=" instead of **in**.

- *Find the name and address of customers who have ordered a product from Hibbert Lumber.*

```
select * from CUSTOMERS
where (FName, LName) in (select FName, LName
                         from orders
                         where SName = 'Hibbert Lumber');
```

- *Find all customers from Woodland who have an account greater than some customer in Davis.*

```
select * from CUSTOMERS
where Account > any (select Account
                     from CUSTOMERS
                     where CAddress like'%Davis%')
  and CAddress like '%Woodland%';
```

- *Find customers who have ordered more than one MegaPC from a supplier.*

```
select * from CUSTOMERS
where (FName, LName) = any
                     (select FName, LName
                     from orders
                     where Prodname = 'MegaPC'
                      and Quantity > 1);
```

☞ Note that **= any** is equivalent to **in**.

- *List all customers who have an account greater than all customers from Davis.*

```
select * from CUSTOMERS
where Account > all
                (select Account from CUSTOMERS
                 where CAddress like'%Davis%');
```

☞ Note that **<> all** is equivalent to **not in**.

- *List the names of all suppliers who offer the best price on some product.*

```
select SName from offers O1
where Price < all (select Price
                   from offers O2
                   where O1.Prodname = O2.Prodname
                     and O1.SName <> O2.SName);
```

☞ If a subquery refers to attributes of an outer query, the subquery is called a *correlated subquery*. References to outer relations and attributes typically occur through using aliases.

Test for (non-)existence

- *List all customers who have ordered a product from a supplier in Davis.*

```
select * from CUSTOMERS C
where exists (select *
                from orders O, SUPPLIERS S
                where O.SName = S.SName
                  and O.FName = C.FName
                  and O.LName = C.LName
                  and SAddress like '%Davis%');
```

This query can also be formulated using a natural join

```
select distinct C.*
from CUSTOMERS C, orders O, SUPPLIERS S
where O.SName = S.SName
 and O.FName = C.FName and O.LName = C.LName
 and SAddress like '%Davis%';
```

- *Give all products (Prodname, Category) for which no offer exists.*

```
select * from PRODUCTS P
where not exists (select * from offers
                  where P.Prodname = Prodname);
```

☞ attributes without preceding alias refer to relations listed in the **from** clause of the subquery where the attributes occur.

- *Find all suppliers that offer a MegaPC, but no TinyMac.*

```
select * from SUPPLIERS S
where exists (select * from offers
              where SName=S.SName
              and Prodname='MegaPC')
and not exists (select * from offers
                where SName=S.SName
                and Prodname='TinyMac');
```

Examples (cont.)

- *Give all pairs of suppliers that offer exactly the same products.*

```
select distinct O1.SName, O2.SName
from offers O1, offers O2
where O1.SName < O2.SName
  and not exists
        (( (select Prodname
            from offers
            where SName = O1.SName)
           minus
           (select Prodname
            from offers
            where SName = O2.SName)
          )
          union
          ( (select Prodname
             from offers
             where SName = O2.SName)
            minus
            (select Prodname
             from offers
             where SName = O1.SName)
          ))
order by O1.SName, O2.SName;
```

Null Values

- If permitted by the schema definition for a table (i.e., no **not null** constraints), attributes can have *null* values.
- *null* $\hat{=}$ unknown, non-existent, or non-applicable value
- Result of any arithmetic expression involving *null* is *null*
- Result of **where** clause condition is *false* if it evaluates to *null*.

and	true	false	null
true	true	false	null
null	null	false	null
false	false	false	false

or	true	false	null
true	true	true	true
null	true	null	null
false	true	false	null

not	
true	false
null	null
false	true

- *Give all suppliers that are not associated with a chain.*

  select $*$ **from** SUPPLIERS **where** Chain **is null**;

  List all customers who have a known account.

  select $*$ **from** CUSTOMERS **where** Account **is not null**;

- All aggregate functions except **count**$(*)$ ignore tuples with *null* values on the aggregate attribute(s).

Aggregate Functions

- Aggregate functions operate on a multiset of values and return a single value. Typical aggregate functions are **min, max, sum, count**, and **avg**.
- For aggregate functions (and the following grouping), an extension of relational algebra exists.
- Examples:

  What is the total number of suppliers?

  select count(SName) **from** SUPPLIERS;

  How many different products are offered?

  select count(**distinct** Prodname) **from** offers;

  What is the minimum and maximum price for products offered by Davis Lumber?

  select min(Price), **max**(Price) **from** offers
  where SName = 'Davis Lumber';

  What is the average price for a MegaPC?

  select avg(Price) **from** offers
  where Prodname = 'MegaPC';

Aggregate Functions (cont.)

What is the total price for the products ordered by the customer Scott Tiger?

select sum(Price * Quantity)
from CUSTOMERS C, orders O, offers F
where C.FName=O.FName **and** C.LName = O.LName
and O.Prodname = F.Prodname
and O.SName = F.SName
and C.FName = 'Scott' **and** C.LName = 'Tiger';

Grouping

- **Idea**: Group tuples that have the same properties into groups, and apply aggregate function to each group. Optionally, consider only groups for the query result that satisfy a certain group condition.

- Syntax in SQL:

  select <attribute(s) [with aggregate function]>
  from $R_1, R_2, \ldots, R_m$
  [**where** P]
  group by <grouping attribute(s)>
  [**having** <condition on group>];

Grouping

- Examples:

  For each supplier, list the name of the supplier and the total number of products the supplier offers.

  select SName, **count**(Prodname)
  from offers
  group by SName;

  For each customer, list the total quantity of orders.

  select FName, LName, **sum**(Quantity)
  from orders
  group by FName, LName;

  Note: attributes that appear in the **select** clause outside of an aggregate function must appear in the **group by** clause !

  List products that are offered by more than one supplier, together with the minimum and maximum price of these offers.

  select Prodname, **min**(Price), **max**(Price)
  from offers
  group by Prodname
  having count$(*) > 1$;

Grouping (cont.)

- A query containing a **group by** clause is processed in the following way:
  1. Select all rows that satisfy the condition specified in the **where** clause.
  2. From these rows form groups according to the **group by** clause.
  3. Discard all groups that do not satisfy the condition in the **having** clause.
  4. Apply aggregate function(s) to each group.
  5. Retrieve values for the columns and aggregations listed in the **select** clause.

- More examples:

  List all suppliers from Davis that offer more than 10 products.

```
select O.SName, count(Prodname)
from SUPPLIERS S, offers O
where S.SName = O.SName and SAddress like '%Davis%'
group by O.SName
having count(Prodname) > 10;
```

Grouping (cont.)

- *List the names of customers who have ordered products for more than $10,000.*

```
select C.FName, C.LName, sum(Quantity*Price)
from CUSTOMERS C, orders O, offers F
where C.FName=O.FName and C.LName = O.LName
  and O.Prodname = F.Prodname
  and O.SName = F.SName
group by C.FName, C.LName
having sum(Quantity*Price) > 10000;
```

What is the minimum total quantity of all orders for a product?

```
select min(sum(Quantity))
from orders
group by Prodname;
```

Data Definition Language (DDL)

Allows the specification of not only a set of relations but also information about each relation, including

- The schema of a relation
- The domain of attributes
- Integrity constraints
- The set of indexes associated with a relation (later)
- The physical storage structure of a relation (later)

Data Types in SQL

- **char**(n), **varchar**(n), . . .
- **real**, **int**, **smallint**, **long**, . . .
- **date**, **time**, . . .

Creating a Table

- Syntax:

  create table <name> (
  <attribute 1> <data type> [**not null**] [**unique**]
  [<attribute constraint>],
  . . . . . .
  <attribute n> <data type> [**not null**] [**unique**]
  [<attribute constraint>],
  [<table constraint(s)>]
  );

Integrity Constraints

- **not null** (do not allow *null* values)

- **primary key** <attribute> (as attribute constraint)
  primary key (<list of attributes>) (as table constraint)

- **unique** <attribute> (as attribute constraint)
  unique (<list of attributes>) (as table constraint)

- **check** <condition>
  If <condition> only refers to one attribute
  → attribute constraint;
  if <condition> includes more than one attribute of the relation
  → table constraint;
  <condition> must be a simple condition that does not contain queries or references to other relations!

- Foreign key (or referential integrity) constraints:
  references <relation>[.<attribute>]
  → attribute constraint
  foreign key <attributes> **references** <relation>[.<attributes>]
  → table constraint

- Example

```
create table Students (
StID      number(9)     constraint Students_pk primary key,
FName     varchar2(50)  not null,
LName     varchar2(50)  not null,
DOB       date          constraint dob_check
                         check(DOB is not null
                         and to_char(DOB) > '01-JAN-01'),
Major     char(5)       constraint fk_majors references Majors,
ZipCode   integer       constraint check_zip
                        check(ZipCode is not null and
                                ZipCode between 1 and 99999),
City      varchar2(50),
Street    varchar2(50),
Started   date          not null,
          constraint dates_check check(DOB < Started),
          constraint name_add unique(FName, LName, DOB)
);
```

- As usual, different database systems (PostgreSQL, Oracle, etc.) can differ in syntax and capabilities (cf. reference manual).

Modifications of the Database

I. Deletions:

- Syntax: **delete from** <relation> [**where** <condition>];
- Examples:

  Delete all suppliers that don't offer any product.

  delete from SUPPLIERS
  where SName **not in** (**select** SName **from** offers);

  Delete all customers having an account less than the average account of all customers.

  delete from CUSTOMERS
  where Account < (**select avg**(Account)
  from CUSTOMERS);

  Problem: Evaluating the condition after each deletion of a customer tuple leads to a change of the subquery result.

  In SQL: First compute **avg**(Account) and identify tuples from CUSTOMERS to delete; then delete those tuples without recomputing **avg**(Account).

II. Insertions

- *Add the customer Scott Tiger (who is living in Davis).*

  insert into CUSTOMERS
  values('Scott','Tiger','Davis',*null*);

  $\hat{=}$ **insert into** CUSTOMERS(FName, LName, CAddress, Account)
  values('Scott','Tiger','Davis',*null*);

  or **insert into** CUSTOMERS(FName, LName, CAddress)
  values('Scott','Tiger','Davis');

  All suppliers are also customers.

  insert into CUSTOMERS(FName, LName, CAddress, Account)
  select '-', SName, SAddress, 0 **from** SUPPLIERS;

III. Updates

- *Increase the Account of the customer Scott Tiger by $5,000, and change his address to Woodland.*

  update CUSTOMERS
  set Account = Account+5000, CAddress = 'Woodland'
  where LName='Tiger' **and** FName='Scott';

- *Set Clark Kent's account to the account of Scott Tiger.*

  update CUSTOMERS
  set Account = (**select** Account **from** CUSTOMERS
  where LName='Tiger' **and** FName='Scott')
  where FName='Clark' **and** LName='Kent';

Views

- Offer a flexible mechanism to hide certain data from the view of a certain user or application; used to realize external schema definitions in the three level schema architecture
- Syntax of a view definition:

  create view <name>[(<list of attribute names>)]
  as <query>;

- The result set of a view is materialized only when the view is queried ⇒ only the definition of a view requires space

- Examples:

```
create view PC_SUPPLS as
select SName, SAddress, Chain
from SUPPLIERS S
where exists (select * from offers
                  where SName = S.SName
                    and Prodname = 'MegaPC');
```

```
create view GOOD_CUSTS(CName, CFName) as
select LName, FName
from CUSTOMERS C
where 10000 < (select sum(Price * Quantity)
                  from orders O, offers R
                  where O.SName=R.SName
                  and O.FName=C.FName
                  and O.LName=C.LName
                  and O.Prodname=R.Prodname) ;
```

Modifications of a View

- Consider the view

  CUST_ORDERS(FName, LName, Prodname, SName, Quantity)

  defined as

  select C.FName, C.LName, Prodname, SName, Quantity
  from CUSTOMERS C, orders O
  where C.FName=O.FName **and** C.LName=O.LName;

- *View Update Problem:* Insert, delete, and update operations on a view must be translated into respective operations of the underlying relations.

  ☞ No problem if there is only one relation underlying the view definition.

  Delete the customer Scott Tiger from CUST_ORDERS.

  Possibility A: delete Scott Tiger from CUSTOMERS
  Possibility B: delete Scott Tiger from orders

- In PostgreSQL, no updates to views are allowed
- In other systems (e.g., Oracle), updates are supported only for highly restricted classes of views (no joins or aggregates, for starters)