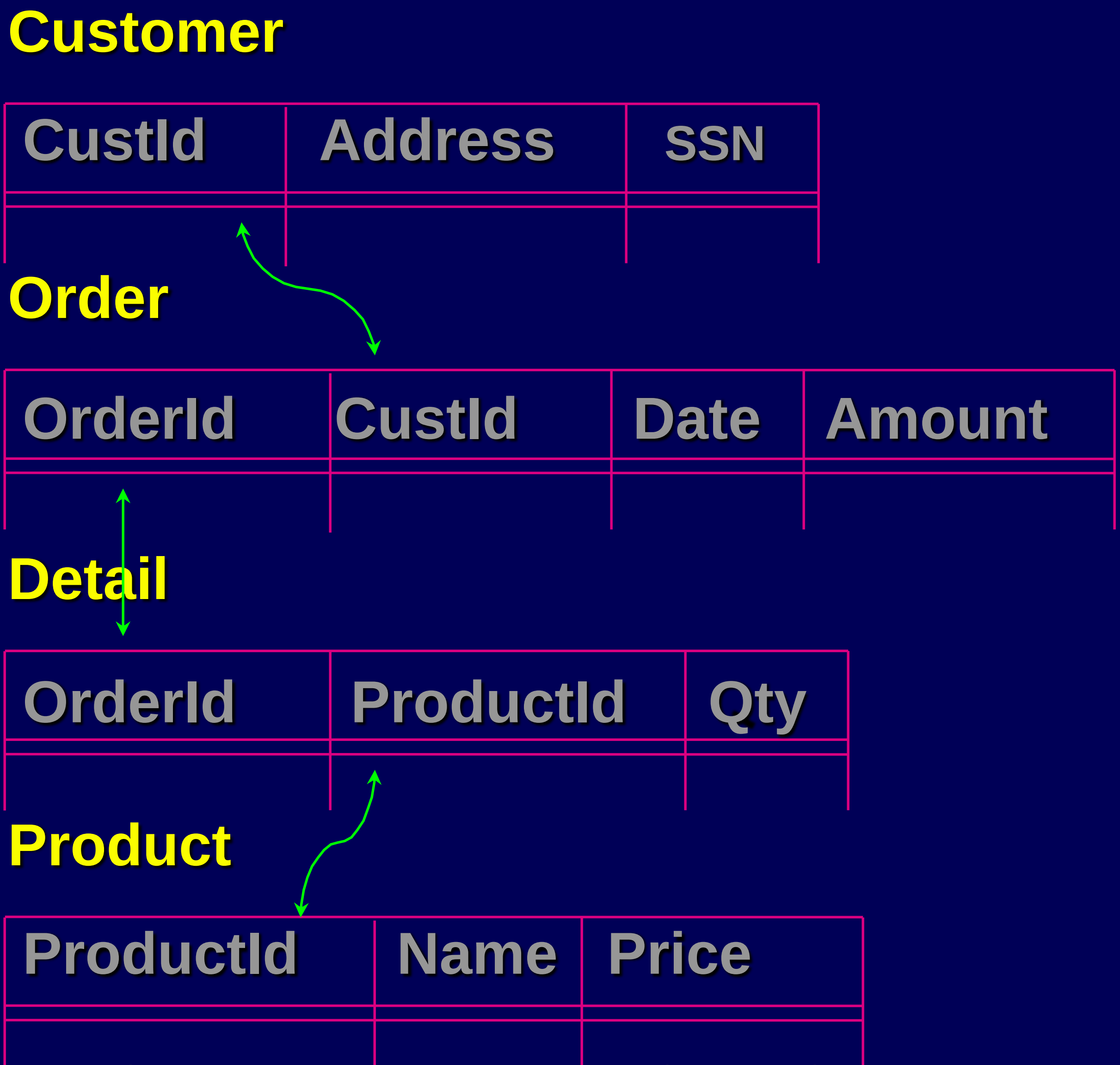
Queries with Ordering and Grouping

Complex Queries in SQL

- Sorting
  - order by attrib [asc | desc]
- Aggregating
  - count, sum, max, min, avg
- Grouping
  - group by ... having ...
- Binary
  - union, intersect, exception
- Nested
  - where attrib = select ...

Order Management



Customer

CustId	Address	SSN

Order

OrderId	CustId	Date	Amount

Detail

OrderId	ProductId	Qty

Product

ProductId	Name	Price

An Instance for Order

Order

OrderId	CustId	Date	Amount
1	3	1-6-97	50.000,00
2	4	3-8-97	8.000,00
3	3	1-9-97	5.500,00
4	1	1-7-97	12.000,00
5	1	1-8-97	1.500,00
6	3	3-9-97	27.000,00

Sorting

- The `order by` clause sorts returned tuples.
- Syntax:

`order by` *Attribute* [`asc` | `desc`]
{, *Attribute2* [`asc` | `desc`] }

- Sorting conditions are evaluated sequentially:
  - If two tuples show the same value for the first sorting attribute, the second sorting attribute is considered.

Query - Sorting

```
select  *
     from Order
     where Amount > 1.000,00
     order by Date
```

OrderId	CustId	Date	Amount
1	3	1-6-97	50.000,00
4	1	1-7-97	12.000,00
5	1	1-8-97	1.500,00
2	4	3-8-97	8.000,00
3	3	1-9-97	1.500,00
6	3	3-9-97	5.500,00

Order by CustId

orderId	CustId	Date	Amount
4	1	1-7-97	12.000,00
5	1	1-8-97	1.500,00
1	3	1-6-97	50.000,00
6	3	3-9-97	5.500,00
3	3	1-9-97	1.500,00
2	4	3-8-97	27.000,00

order by CustId asc, Date desc

OrderId	CustId	Date	Amount
5	1	1-8-97	1.500,00
4	1	1-7-97	12.000,00
6	3	3-9-97	5.500,00
3	3	1-9-97	1.500,00
1	3	1-6-97	50.000,00
2	4	3-8-97	27.000,00

Aggregate Functions

- The results of the query depend on the aggregation of starting tuples into sets.
- The result of a query with aggregate functions depends on the evaluation of the contents of a set of rows.
- SQL-2 provides the following aggregation operators:

– `count`	cardinality
– `sum`	sum
– `max`	maximum
– `min`	minimum
– `avg`	average

count

- count returns the number of rows or of distinct values.
- Syntax:

  count(< * | [distinct | all] *AttributeList* >)

- Find the number of orders:

```
select count(*)
from Order
```

- Find the distinct values of the attribute CustId for all the rows of Order:

```
select count(distinct CustId)
from Order
```

- Find the number of rows of Order for which the CustId attribute does not have a null value:

```
select count(all CustId)
from Order
```

sum, max, min, avg

- Syntax:
  < sum | max | min | avg > ([distinct | all] *AttrExpr*)

- The option distinct considers just once every single value:
  - useful for the sum and avg functions, only.

- The option all considers all the values **NOT** null.

Query with `max`

- **Find the greatest (largest total amount) order:**

```
select max(Amount) as MaxAmt
   from Order
```

MaxAmt
50.000,00

Query with `sum`

- **Find the sum of the amounts of the orders from customer 1.**

```
select sum(Amount) as SumAmt
  from Order
  where CustId = 1
```

SumAmt
13.500,00

Aggregate Functions with join

- Find the greatest (largest amount) order among those orders which include the product ‘ABC’ :

```
select max(Amount) as MaxAmtABC
  from Order, Detail
  where Order.OrderId = Detail.OrderId
        and ProductId = ’ABC’
```

Aggregate Functions and Target List

- **Wrong** query:

```
select Date, max(Amount)
  from Order, Detail
  where Order.OrderId = Detail.OrderId and
        ProductId = ’ABC’
```

- Which Date (of which order) to be considered? The target list must be homogeneous.
- Find the greatest (largest amount) and the smallest amounts of the orders:

```
select max(Amount) as MaxAmt,
       min(Amount) as MinAmt
 from Order
```

Aggregate Functions and Target List

- Find the greatest (largest amount) and the smallest amounts of the orders:

```
select max(Amount) as MaxAmt,
       min(Amount) as MinAmt
 from Order
```

MaxAmt	MinAmt
50.000,00	1.500,00

Query with Grouping

- Queries may use operators which aggregate (group) together tuples into sets, and evaluate the operators for every set.
- Clauses:
  - **group by** (grouping)
  - **having** (selection over returned results for every group)

```
select …
  from …
  where …
  group by …
  having …
```

Query with Grouping

- Find the sum of the Amounts for the orders after 10-6-97 for those customers who issued at least 2 orders.

```
select CustId, sum(Amount)
  from Order
  where Date > 10-6-97
     group by CustId
     having count(Amount)  >= 2
```

Step 1: Evaluate `where`

OrderId	CustId	Date	Amount
2	4	3-8-97	8.000,00
3	3	1-9-97	5.500,00
4	1	1-7-97	12.000,00
5	1	1-8-97	1.500,00
6	3	3-9-97	27.000,00

Step 2 : Grouping

- Evaluate the `group by` clause.

OrderId	CustId	Date	Amount
4	1	1-7-97	12.000,00
5	1	1-8-97	1.500,00
3	3	1-9-97	1.500,00
6	3	3-9-97	5.500,00
2	4	3-8-97	8.000,00

Steo 3 : Compute the Aggregates

- Compute `sum(Amount)` **and** `count(Amount)` for every group

CustId	sum (Amount)	count (Amount)
1	13.500,00	2
3	7.000,00	2
4	5.000,00	1

Step 4 : Extract the groups

- evaluate the predicate `count(Amout) >= 2`

CustId	sum (Amount)	count (Amount)
1	13.500.000	2
3	7.000.000	2
~~4~~	~~5.000.000~~	~~1~~

Step 5 : Output the Result

CustId	sum (Amount)
1	13.500,00
3	7.000,00

Query with `group by` and target list

- **Wrong** query:

```
select Amount
  from Order
  group by CustId
```

- **Wrong** query:

```
select O.CustId, count(*), C.City
  from Order O join Customer C
         on (O.CustId = C.CustId)
  group by O.CustId
```

- Correct query:

```
select O.CustId, count(*), C.City
  from Order O join Customer C
         on (O.CustId = C.CustId)
  group by O.CustId, C.City
```

where or **having**?

- The `having` clause includes the predicates for which the result of the aggregate function (over the group) is considered, only.
- Find the departments for which the average salary of employees working in office number 20 is greater than 25:

```
select Department
  from Employee
  where office = ’20’
  group by Department
  having avg(Salary) > 25
```

Query with grouping and sorting

- The result of the query is to be sorted by the `order by` command.

```
select  …
  from …
  [ where … ]
  group by …
  [ having … ]
  order by …
```

Grouping and Sorting

- Find the sum of the amounts of the orders after 10-6-97 for those customer who issued at least 2 orders, sorting them in the decreasing order for sum of amount

```
select  CustId, sum(Amount)
  from Order
  where Date > 10-6-97
  group by CustId
  having count(Amount) >= 2
  order by sum(Amount) desc
```

After the sorting clause

CustId	sum (Amount)
1	13.500,00
3	7.00,00

Double Grouping

- Find the sum o the quantities of the details of the orders for every customer and for every product, on condition that the sum exceeds 50.

```
select CustId, ProductId, sum(Qty)
  from Order as O, Detail as D
  where  O.OrderId = D.OrderId
     group by CustId, ProductId
     having sum(Qty) > 50
```

After the join and the grouping

Order		Detail			
CustId	Order. OrderId	Detail. OrderId	ProductId	Qty	
1	3	3	1	30	group 1,1
1	4	4	1	20	
1	3	3	2	30	group 1,2
1	5	5	2	10	
2	3	3	1	60	group 2,1
3	1	1	1	40	group 3,1
3	2	2	1	30	
3	6	6	1	25	

Extracting the Result

- evaluate the aggregate function **`sum(Qty)`** and the **`having`** predicate

CustId	ProductId	sum(Qty)
~~1~~	~~1~~	~~50~~
~~1~~	~~2~~	~~40~~
2	1	60
3	1	95

Set Queries

Set queries chain two SQL queries by operators.

Syntax:

SelectSQL **{ <union | intersect | except> [all]** ***SelectSQL*}**

• **union**	plus
• **intersect**	intersection
• **except (minus)**	difference

Duplicate tuples are removed, unless the **all** clause is used.

Union

Find the code of the orders whose grand total exceeds euro 500 or where one product has been ordered in a quantity exceeding 1000.

```
select OrderId
  from Order
  where Amount > 500
union
select OrderId
  from Detail
  where Qty > 1000
```

Attribute Names in the Result Table

```
select Father
   from Paternity
union
select Mother
   from Maternity
```

- Which are the names of the result table?
  - No name
  - Those of the first operand
  - …

Positional Notation

```
select Father, Kid
  from Paternity
  union
select Kid, Mother
  from Maternity
```

```
select Father, Kid
  from Paternity
  union
select Mother, Kid
  from Maternity
```

- They are different queries.

Father	Kid
Luigi	Giorgio
Stefano	Giovanni

Mother	kid
Anna	Giorgio
Paola	Giovanni

Positional Notation

```
select Father, Kid
   from Paternity
   union
Select Kid, Mother
   from Maternity
```

```
select Father, Kid
   from Paternity
   union
select Mother, Kid
   from Maternity
```

Luigi	Giorgio
Stefano	Giovanni
Giorgio	Anna
Giovanni	Paola

Luigi	Giorgio
Stefano	Giovanni
Anna	Giorgio
Paola	Giovanni

The `all` Token

- Find the names of the fathers of the kids named “Giorgio” or “Giovanni”, counting twice the fathers having one kid named Giorgio and another kid named Giovanni.

```
select Father
  from Paternity
  where Kid = ’Giorgio’
union all
select Father
  from Paternity
  where Kid = ’Giovanni’
```

Difference

- Find the OrderId of the orders whose total amount exceeds euro 500, but where no product is included with a quantity exceeding 1000.

```
select OrderId
  from Order
  where Amount > 500
except
select OrderId
  from Detail
  where Qty > 1000
```

- Can be described by a nested query, too.

Difference

- Find the orderId of the orders presenting *n* >=1 lines of order with a quantity greater than 10 and do NOT present *m* >= lines of order with a quantity exceeding 1000.

```
select OrderId
  from Detail
  where Qty > 10
except all
select OrderId
  from Detail
  where Qty > 1000
```

Intersection

- Find the OrderId of the orders whose total amount exceeds euro 500 and where at least one product is reported with a quantity exceeding 1000.

```
select OrderId
   from Order
   where Amount > 500
intersect
Select OrderId
   from Detail
   where Qty > 1000
```

- This query can be expressed as a nested query, too.

Nested Queries

- The `where` clause may include predicates that compare ONE attribute with the result of an SQL query.
- Syntax:

  ScalarValue Operator < **`any`** | **`all`** > *SelectSQL*

  - **`any`**: the predicate is true if **at least** one of the tuples returned by the *SelectSQL* fulfills the comparison.
  - **`all`**: the predicate is true if **all** the tuples returned by the *SelectSQL* fulfill the comparison
  - *Operator*: **=, <>, <, <=, >, >=**
- The nested query is the query inside the **`where`** clause.

A Simple Nested Query

- Find the orders of products with a price greater than 100.

```
select OrderId
   from Detail
   where ProductId = any (select ProductId
                          from Product
                          where Price > 100)
```

- The above query is equivalent to (not a nested query, unless duplicates):

```
select OrderId
   from Detail D, Product P
   where D.ProductId=P.ProductId and Price>100
```

A Simple Nested Query

- Find the products ordered together (within the same order) with the ‘ABC’ product.
- Without a nested query:

```
select D1.ProductId
    from Detail D1, Detail D2
    where D1.OrderId = D2.OrderId and
          D2.ProductId = ’ABC’
```

- By a nested query:

```
select ProductId
    from Detail
    where OrderId = any (select OrderId
          from Detail
          where ProductId = ’ABC’)
```

NOT with Nested Queries

- Find the orders which do NOT include the product 'ABC':

```
select distinct OrderId
    from Order
    where OrderId <> all (select OrderId
                          from Detail
                          where ProductId = 'ABC')
```

- Alternatively:

```
select OrderId
    from Order
except
select OrderId
    from Detail where ProductId = 'ABC'
```

The **`in`** and **`not in`** Operators

- The `in` operator is equivalent to `= any`

```
 select ProductId
   from Detail
   where OrderId in (select OrderId
                     from Detail
                     where ProductId = 'ABC')
```

- The `not in` operator is equivalent to `<> all`

```
select distinct OrderId
     from Order
     where OrderId not in (select OrderId
                           from Detail
                           where ProductId = 'ABC')
```

Nested Queries

- Find the names and the addresses of the customers who issued at least one order for a total amount exceeding 10.000.

```
select Name, Address
  from Customer
  where CustId in
          select CustId
                 from Order
                 where Amount > 10000
```

More Nested Queries

- Find the names of the customers which issued some (at least one) order which includes the “tyre” product.

```
select Name, Address
   from Customer
   where CustId in select CustId
        from Order
        where OrderId in select OrderId
             from Detail
             where Product in select ProductId
                  from Product
                  where Name = ’tyre’
```

An Equivalent Query

An equivalent query (a part from duplicates) is:

```
select C.Name, Address
    from Customer as C, Order as O,
        Detail as D, Product as P
    where C.CustId = O.CustId
        and O.OrderId = D.OrderId
        and D.ProductId = P.roductId
        and P.Name = ’tyre’
```

max and **min** in Nested Queries

- The aggregate operators max and min can be expressed by nested queries.
- Find the order with the greatest amount:
  - by max:

```
select OrderId
 from Order
 where Amount in (select max(Amount)
      from Order)
```

  - by a nested query:

```
select OrderId
 from Order
      where Amount >= all (select Amount
           from Order)
```

Use of any and all

```
select OrderId
from Order
where Amount > any
        select Amount
        from Order
```

```
select OrdeId
from Order
where Amount >= all
        select Amount
        from Order
```

OrderId	Amount
1	50,00
2	300,00
3	90,00

ANY	ALL
F	F
T	T
T	F

The **`exists`** Operator

- The existential quantifier `exists` can be used within a query.
- Syntax:

`exists` *SelectStar*

the predicate returns true if the *SelectStar* query returns a not null result (i.e. a not empty set).

We always use `select *` when the target list is not relevant.

Complex Nested Queries

- The nested query (inner part) can use variables from the main query (outer part) :
  - Interpretation: the inner query is evaluated for every tuple of the main query.
- Find all the customers which issued more than one order in the same day:

```
select CustId
   from Order O
   where exists (select *
        from Order O1
        where O1.CustId = O.CustId
             and O1.Date = O.Date
             and O1.OrderId <> O.OrderId)
```

Complex Nested Query

- Find all the students who [do not] have an homonymous:

```
select *
   from Student S
   where [not] exists
        (select *
             from Student S1
             where S1.Name = S.Name
             and S1.Surname = S.Surname
             and S1.SSN <> S.SSN)
```

Tuple Constructor

- The comparison in the inner query may involve more than one attribute.
- Attributes are delimited di parentheses (tuple constructor).
- The previous query can be expressed as:

```
select *
    from Student S
    where (Name,Surname) [not] in
          (select Name, Surname
               from Student S1
               where S1.SSN <> S.SSN)
```

Comments on Nested Queries

- Use of nested queries may produce queries ‘less declarative’, but, typically, with an increased readability.
- The first version of SQL included the nested (or structured) form with ONE table in the `from` clause, only.
- Nested queries (inner part) may NOT include set operator (“union is performed at the outer level, only”); but some commercial system override this limitation.

Comments on Nested Queries

- Complex queries, which make a heavy use of variables, often read very hardly.
- Use of variables must respect the scope (visibility) of vars:
  - a var can be used within the query where it is defined, or in a nested query if its;
  - if a var name is omitted, the DNMS assumes to use the closest one.

Scope of a Var

- **Wrong** query:

```
select *
   from Customer
   where CustId in
     (select CustId
            from Order O1
            where OrderId = 'AZ1020')
   or CustId in
     (select CustId
            from Order O2
            where O2.Date = O1.Date)
```

- The query is not correct because the `O1` var is not visible in the **second** nested query.