Appendix D Extrema Queries and Ranking Queries in SQL

Simple Extrema Queries

An *extremum* is either a maximum or minimum of some set of values. The values can be of any type that has a meaningful ordering, including numbers, strings, dates, and so on. Standard SQL provides two main ways of computing extrema: the **max**( ) and **min**( ) bag functions, and the **all** clause.

Given a column *a*, we could use the following patterns:

Maximum *a*:	**max**(*a*)	Minimum *a*:	**min**(*a*)
	>= **all** *a*		<= **all** *a*

Table D.1 Relational table with facts about the populations of countries

Country:

countryCode	*countryName*	*population*
AU	Australia	21 000 000
CN	China	1 327 000 000
RU	Russia	140 000 000
US	USA	303 000 000

Table D.1 contains some information about countries and their populations. We can find the countries with the largest and smallest populations in the following way.

```
select countryCode, countryName
from Country
where population =
    (select max(population)
    from country)
```

```
select countryCode, countryName
from Country
where population =
    (select min (population)
    from country)
```

The queries return the following results.

countryCode	*countryName*
CN	China

countryCode	*countryName*
AU	Australia

The following constructs produce equivalent results to those shown previously.

```
select countryCode, countryName
from Country
where population >= all
    (select max(population)
    from country)
```

```
select countryCode, countryName
from Country
    where population <= all
    (select min (population)
    from country)
```

An alternative approach is based on the use of a *quota query* that returns a limited number of rows. Most database systems provide this kind of functionality, although the syntax may vary, with constructs such as **top** *n*..., **limit** *n*..., and **fetch first** *n* **rows only** being available in different implementations. As an example, we'll use SQL Server's **top** clause. The key to finding an extremum with this approach is to order by the required value and then select the top 1 from the result. This gives either the maximum or the minimum value depending on whether the ordering was descending or ascending. The following constructs produce the same maximum and minimum results as those shown previously.

```
select top 1 countryCode, countryName
from Country
order by population desc
```

```
select top 1 countryCode, countryName
from Country
order by population -- default is asc
```

Group Extrema Queries

As a further application of the group-by clause, we now consider group extrema queries. Group extrema queries take one of the following two forms:

List those *groups* that have the *maximum* value of some expression.
List those *groups* that have the *minimum* value of some expression.

If *expn* denotes the relevant expression, group extrema queries may be formulated using one of the following patterns:

Group max:
```
select ... from ...
group by ...
having expn >= all
    (select expn from ...
    group by ...)
```

Group min:
```
select ... from ...
group by ...
having expn <= all
    (select expn from ...
    group by ...)
```

For example, suppose the following query is applied to the table *Eats* (person, food), a sample population for which is shown in Table D.2.

Q1: Who eats the *most* foods?

This may be expressed as the SQL query S1. For the sample data in Table D.2, the subquery returns the bag (3, 2, 2, 1, 6, 4). The outer query groups the table into 5 groups, one for each person, compares the food count for each person with this bag, and selects just those people whose food count is greater than or equal to each value in the bag. This ensures selection of the person(s) with the maximum number of foods (in this case Humphrey).

Table D.2 Relational table storing facts about who eats what foods.

Eats:

person	*food*
Ann	apple
Ann	beef
Ann	potato
Bill	apple
Bill	potato
Chris	apple
Chris	potato
Fred	peas
Humphrey	apple
Humphrey	beef
Humphrey	chicken
Humphrey	orange
Humphrey	peas
Humphrey	potato
Sue	apple
Sue	chicken
Sue	orange
Sue	peas

S1:
```
select person
from  Eats
group by person
having count(*) >= all
      (select count(*)
       from  Eats
       group by person)   →   {'Humphrey'}
```

As an example of a group minimum, consider the query Q2. By analogy with the previous example, the SQL formulation in S2 should be easy to follow.

Q2: Who eats the *fewest* foods, and what is this number?

S2:
```
select person, count(*)
from  Eats
group by person
having count(*) <= all
      (select count(*)
       from  Eats
       group by person)   →   {'Fred', 1}
```

Quota queries

Quota queries were used in the previous section to find extrema. They provide useful functionality in their own right in returning a limited set of results from a query. Again, we'll use SQL Server's **top** clause as an illustration.

In SQL Server the **top** expression (which defines a number of rows) is placed a little strangely before the start of the select statement's column list. Retrieval of *n* rows is accomplished easily by specifying **select top** *n* … . It's also possible to select a percentage of

the rows, by using **select top** *n* **percent**…. , where *n* is a number (possibly a decimal fraction) between 0 and 100.

A critical question to resolve when using quotas is what to do in the event of a tie for the last place in a list. Table D.3 shows a selection of persons and their respective ages.

Table D.3 A relational table showing persons and their ages.

Person:

personName	*age*
Ann	25
Eve	18
Carol	19
David	21
Bill	25
Fred	19

If we wanted to find the two youngest people listed, we could use a construct such as

```
select top 2 personName, age
from Person
order by age -- default is asc
```

This would give a result such as

personName	*age*
Eve	18
Carol	19

Whether we get Carol or Fred in second place would depend on exactly how the table had been defined and populated. Also, we might have good business reasons to see if there are any other persons in a tie for second place. In such cases we need to specify the statement with ties, as shown here.

```
select top 2 with ties personName, age
from Person
order by age -- default is asc
```

This gives us the expected result.

personName	*age*
Eve	18
Carol	19
Fred	19

Note that ties are only an issue for the last place in the list. If the age of Carol had been 18 instead of 19, the query would return only two rows whether or not we specified **with ties**.

Ranking queries

Sometimes we may want to indicate the position of an item in some list. Ordinal numbers designate the place occupied by an item in an ordered sequence (first, second, third, and so on). SQL provides several *ranking* functions that return the ordinal position of an item. A typical application is to give the ordinal position of a row in a result table according to some ordering specification. Most SQL implementations provide functions of this kind, although, as usual, the syntax may vary a little from one to another.

To provide a concrete context, we'll assume that we want to use SQL Server to provide a ranking for the persons shown in Table D.3. The ranking will be shown in a column specified by the following simplified syntax:

<rankFn> **over** (**order by** <column> [**desc**])

where <rankFn> is one of the available ranking functions and <column> is the column that forms the basis for the ranking. SQL Server provides following four ranking functions.

row_number( )	gives the position in sequential order
rank( )	gives the ordinal position (1^{st}, 2^{nd}, 3^{rd}, etc.) but leaves gaps in the sequence where there are there are ties for the same place
dense_rank( )	similar to rank() but closes up the gaps in the sequence
ntile(*n*)	divides the list into *n* "buckets", each containing approximately the same number of elements

The simplest of the ranking functions is **row_number**( ), which can be used in the following way.

```
select personName, age,
       row_number( ) over ( order by age ) as rowNum
from Person
order by age
```

This query produces the following result. The last column has been specified with an alias for to aid readability. Note that the row numbering is not affected by ties.

personName	*age*	*rowNum*
Eve	18	1
Fred	19	2
Carol	19	3
David	21	4
Ann	25	5
Bill	25	6

The **rank**( ) function can be used in a similar way.

```
select personName, age,
        rank( ) over ( order by age ) as rank
from Person
order by age
```

This query produces the following result. The second and third rows rank equally for second place (both have an age of 19) and so they both receive a rank of 2. Since there are three preceding values, the next row receives a rank of 4. The last two rows tie and so both receive a rank of 5.

personName	*age*	*rank*
Eve	18	1
Fred	19	2
Carol	19	2
David	21	4
Ann	25	5
Bill	25	5

The **dense_rank**( ) function is also similar, except for the result.

```
select personName, age,
        dense_rank( ) over ( order by age ) as denseRank
from Person
order by age
```

This produces the same clustering as the previous query, but with no gaps in the numbering sequence. One way of thinking about this is to see David as being 4th in the list, but 3rd in the ranks.

personName	*age*	*denseRank*
Eve	18	1
Fred	19	2
Carol	19	2
David	21	3
Ann	25	4
Bill	25	4

The last ranking function is **ntile**(*n*), which operates slightly differently. The parameter *n* specifies a number of "buckets" into which the elements are to be allocated. The first bucket contains the first 1/*n* elements, the second bucket contains the second 1/*n* elements, and so on. The value produced by the function is the number of the bucket into which that element will be placed for the given ordering. Obviously, if the number of elements is not exactly divisible by *n*, some groups will have one fewer member than the largest group. Here is an example.

```
select personName, age,
        ntile( 4 ) over ( order by age ) as quartile
from Person
order by age
```

For such a small table the results are not particularly impressive. Since we only have six rows, each quartile will only contain one or two elements, but the function is much more useful on larger tables.

personName	*age*	*quartile*
Eve	18	1
Fred	19	1
Carol	19	2
David	21	2
Ann	25	3
Bill	25	4

One point to note about the ranking functions is that they have an independent specification of ordering. This makes for great flexibility, but can cause confusion if not handled carefully. As an example, here is a query that uses ranking functions with one ordering, but displays the table rows with a different ordering.

```
select personName, age,
        row_number( ) over ( order by age ) as rowNum,
        rank(  ) over ( order by age ) as rank,
        dense_rank( ) over ( order by age ) as denseRank,
        ntile( 4 ) over ( order by age ) as quartile
from Person
order by personName -- different order to ranking functions
```

The result may not seem particularly intuitive to a user.

personName	*age*	*rowNum*	*rank*	*denseRank*	*quartile*
Ann	25	5	5	4	3
Bill	25	6	5	4	4
Carol	19	3	2	2	2
David	21	4	4	3	2
Eve	18	1	1	1	1
Fred	19	2	2	2	1

Accessing the values produced by a ranking function also takes a little care. Let's say that we want to retrieve all rows in the second or third quartile. We might be tempted to write something like the following.

```
select personName, age,
      ntile( 4 ) over (order by age) as myQuartile
from Person
where myQuartile = 2 or myQuartile = 3
order by age
```

This does not work, since myQuartile is not a column in the base table. We can resolve the problem by using a Common Table Expression to create a temporary named result table, and then run a query on that to obtain the required result. The necessary code would look like the following.

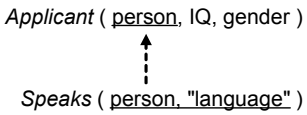
```
with TempRank as
      (select personName, age,
            ntile( 4 ) over (order by age) as myQuartile
      from Person)
-- now we can query this temporary "table"
select * from TempRank
where myQuartile = 2 or myQuartile = 3
order by age
```

This produces the expected result.

personName	*age*	*myQuartile*
Carol	19	2
David	21	2
Ann	25	3

Exercise D1

1. The following relational schema is designed to store data about applicants for positions as language interpreters. A sample population is shown.



Applicant:

person	*IQ*	*gender*
Ann	120	F
Bill	135	M
Chris	130	F
David	125	M
Ernie	115	M
Fumie	120	F

Speaks:

person	*"language"*
Ann	English
Ann	Japanese
Bill	English
Bill	Japanese
Bill	Russian
Chris	English
Chris	Russian
David	English
David	Japanese
David	Russian
Ernie	English
Ernie	French
Ernie	Japanese
Fumie	Japanese
Fumie	Korean

Formulate the following queries in SQL.

(a) Which language(s) is/are spoken by the fewest applicants?
(b) Who speaks the most languages?
(c) For the applicants, which gender has the highest average IQ?
(d) For the applicants, which gender has the greatest range of IQs?

2. The following questions refer to the schema given in Question 1.
   (a) List the Japanese speakers with the two highest IQs.
   (b) Which applicants have the highest IQ for their gender?
   (c) Of the applicants with above-average IQ, who speaks the most languages?

3. The following table is used to store information about products.

   Product (productId, productName, unitPrice, unitsInStock)

   Formulate the following queries in SQL.
   (a) List the productId, productName and unitsInStock for the products with the five highest numbers of units in stock (i.e. five output rows).
   (b) As (a), but include any ties for the last place.
   (c) List the productId, productName and unitPrice for the products having the five highest distinct prices.
   (d) List the productId, productName and unitsInStock for the highest 10% of products which have units in stock.
   (e) List the productId, productName and unitsInStock in decreasing order of unitsInStock, for products with more than 100 units in stock, along with the row number, rank and dense rank (using the same ordering).

4. The following questions refer to the schema given in Question 3.
   (a) List the products in the second lowest price decile (i.e., the second lowest of ten equally sized groups).
   (b) List the products not ranked 1^{st}, 2^{nd}, or 3^{rd} when products are ranked in decreasing order of numbers of units in stock.