Important Functions

Use of DISTINCT

List the property numbers of all properties that have been viewed.

```
SELECT propertyNo
FROM Viewing;
```

Notice that there are several duplicates, because unlike the relational algebra Projection operation

Result:

propertyNo
PA14
PG4
PG4
PA14
PG36

SELECT does not eliminate duplicates when it projects over one or more columns. To eliminate the duplicates, we use the DISTINCT keyword. Rewriting the query

```
SELECT DISTINCT propertyNo
FROM Viewing;
```

Result:

propertyNo
PA14
PG4
PG36

Comparison search condition

List all staff with a salary greater than £10,000.

```
SELECT staffNo, fName, lName, position, salary
FROM Staff
WHERE salary > 10000;
```

Result:

staffNo	fName	lName	position	salary
SL21	John	White	Manager	30000.00
SG37	Ann	Beech	Assistant	12000.00
SG14	David	Ford	Supervisor	18000.00
SG5	Susan	Brand	Manager	24000.00

In SQL, the following simple comparison operators are available:

=	equals		
<>	is not equal to (ISO standard)	! =	is not equal to (allowed in some dialects)
<	is less than	< =	is less than or equal to
>	is greater than	> =	is greater than or equal to

More complex predicates can be generated using the logical operators **AND**, **OR**, and **NOT**, with parentheses (if needed or desired) to show the order of evaluation. The rules for evaluating a conditional expression are:

- an expression is evaluated left to right;
- subexpressions in brackets are evaluated first;
- NOTs are evaluated before ANDs and ORs;
- ANDs are evaluated before ORs.

The use of parentheses is always recommended, in order to remove any possible ambiguities.

Compound comparison search condition

List the addresses of all branch offices in London or Glasgow.

```
SELECT *
FROM Branch
WHERE city = 'London' OR city = 'Glasgow';
```

In this example the logical operator OR is used in the WHERE clause to find the branches in London (city = 'London') or in Glasgow (city = 'Glasgow').

Result:

branchNo	street	city	postcode
B005	22 Deer Rd	London	SW1 4EH
B003	163 Main St	Glasgow	G11 9QX
B002	56 Clover Dr	London	NW10 6EU

Range search condition (BETWEEN/NOT BETWEEN)

List all staff with a salary between £20,000 and £30,000.

```
SELECT staffNo, fName, lName, position, salary
FROM Staff
WHERE salary BETWEEN 20000 AND 30000;
```

The BETWEEN test includes the endpoints of the range, so any members of staff with a salary of £20,000 or £30,000 would be included in the result.

Result:

staffNo	fName	lName	position	salary
SL21	John	White	Manager	30000.00
SG5	Susan	Brand	Manager	24000.00

There is also a negated version of the range test (NOT BETWEEN) that checks for values outside the range. The BETWEEN test does not add much to the expressive power of SQL, because it can be expressed equally well using two comparison tests. We could have expressed the previous query as:

```
SELECT staffNo, fName, lName, position, salary
FROM Staff
WHERE salary >= 20000 AND salary <= 30000;
```

However, the BETWEEN test is a simpler way to express a search condition when considering a range of values.

Set membership search condition (IN/NOT IN)

List all managers and supervisors.

```
SELECT staffNo, fName, lName, position
FROM Staff
WHERE position IN ('Manager', 'Supervisor');
```

The set membership test (IN) tests whether a data value matches one of a list of values, in this case either 'Manager' or 'Supervisor'.

Result:

staffNo	fName	lName	position
SL21	John	White	Manager
SG14	David	Ford	Supervisor
SG5	Susan	Brand	Manager

There is a negated version (NOT IN) that can be used to check for data values that do not lie in a specific list of values. Like BETWEEN, the IN test does not add much to the expressive power of SQL. We could have expressed the previous query as:

```
SELECT staffNo, fName, lName, position
FROM Staff
WHERE position = 'Manager' OR position = 'Supervisor';
```

However, the IN test provides a more efficient way of expressing the search condition, particularly if the set contains many values.

NULL search condition (IS NULL/IS NOT NULL)

List the details of all viewings on property PG4 where a comment has not been supplied.

```
SELECT clientNo, viewDate
FROM Viewing
WHERE propertyNo = 'PG4' AND comment IS NULL;
```

Result:

clientNo	viewDate
CR56	26-May-13

Sorting Results (ORDER BY Clause)

Single-column ordering

Produce an abbreviated list of properties arranged in order of property type.

```
SELECT propertyNo, type, rooms, rent
FROM PropertyForRent
ORDER BY type;
```

Result:

propertyNo	type	rooms	rent
PL94	Flat	4	400
PG4	Flat	3	350
PG36	Flat	3	375
PG16	Flat	4	450
PA14	House	6	650
PG21	House	5	600

Multiple column ordering

```
SELECT propertyNo, type, rooms, rent
FROM PropertyForRent
ORDER BY type, rent DESC;
```

Result:

propertyNo	type	rooms	rent
PG16	Flat	4	450
PL94	Flat	4	400
PG36	Flat	3	375
PG4	Flat	3	350
PA14	House	6	650
PG21	House	5	600

Using the SQL Aggregate Functions

As well as retrieving rows and columns from the database, we often want to perform some form of summation or **aggregation** of data, similar to the totals at the bottom of a report. The ISO standard defines five **aggregate functions**:

- COUNT – returns the number of values in a specified column
- SUM – returns the sum of the values in a specified column
- AVG – returns the average of the values in a specified column
- MIN – returns the smallest value in a specified column
- MAX – returns the largest value in a specified column

These functions operate on a single column of a table and return a single value. COUNT, MIN, and MAX apply to both numeric and nonnumeric fields, but SUM and AVG may be used on numeric fields only. Apart from COUNT(*), each function eliminates nulls first and operates only on the remaining nonnull values. COUNT(*) is a special use of COUNT that counts all the rows of a table, regardless of whether nulls or duplicate values occur.

If we want to eliminate duplicates before the function is applied, we use the keyword DISTINCT before the column name in the function. The ISO standard allows the keyword ALL to be specified if we do not want to eliminate duplicates, although ALL is assumed if nothing is specified. DISTINCT has no effect with the MIN and MAX functions. However, it may have an effect on the result of SUM or AVG, so consideration must be given to whether duplicates should be included or excluded in the computation. In addition, DISTINCT can be specified only once in a query.

Use of COUNT(*)

How many properties cost more than £350 per month to rent?

```
SELECT COUNT(*) AS myCount
FROM PropertyForRent
WHERE rent > 350;
```

Restricting the query to properties that cost more than £350 per month is achieved using the **WHERE** clause. The total number of properties satisfying this condition can then be found by applying the aggregate function **COUNT**.

Use of COUNT(DISTINCT)

How many different properties were viewed in May 2013?

```
SELECT COUNT(DISTINCT propertyNo) AS myCount
FROM Viewing
WHERE viewDate BETWEEN '1-May-13' AND '31-May-13';
```

Again, restricting the query to viewings that occurred in May 2013 is achieved using the **WHERE** clause. The total number of viewings satisfying this condition can then be found by applying the aggregate function **COUNT**. However, as the same property may be viewed many times, we have to use the **DISTINCT** keyword to eliminate duplicate properties.

Use of COUNT and SUM

Find the total number of Managers and the sum of their salaries.

```
SELECT COUNT(staffNo) AS myCount, SUM(salary) AS mySum
FROM Staff
WHERE position = 'Manager';
```

Restricting the query to **Managers** is achieved using the **WHERE** clause. The number of **Managers** and the sum of their salaries can be found by applying the **COUNT** and the **SUM** functions respectively to this restricted set.

Result:

myCount	mySum
2	54000.00

Use of MIN, MAX, AVG

Find the minimum, maximum, and average staff salary.

```
SELECT MIN(salary) AS myMin, MAX(salary) AS myMax, AVG(salary) AS myAvg
FROM Staff;
```

In this example, we wish to consider all staff and therefore do not require a WHERE clause. The required values can be calculated using the MIN, MAX, and AVG functions based on the salary column.

Result:

myMin	myMax	myAvg
9000.00	30000.00	17000.00

Grouping Results (GROUP BY Clause)

The previous summary queries are similar to the totals at the bottom of a report. They condense all the detailed data in the report into a single summary row of data. However, it is often useful to have subtotals in reports. We can use the GROUP BY clause of the SELECT statement to do this. A query that includes the GROUP BY clause is called a **grouped query**, because it groups the data from the SELECT table(s) and produces a single summary row for each group. The columns named in the GROUP BY clause are called the **grouping columns**. The ISO standard requires the SELECT clause and the GROUP BY clause to be closely integrated. When GROUP BY is used, each item in the SELECT list must be **single-valued per group**. In addition, the SELECT clause may contain only:

- column names;
- aggregate functions;

- constants;
- an expression involving combinations of these elements.

All column names in the SELECT list must appear in the GROUP BY clause unless the name is used only in an aggregate function. The contrary is not true: there may be column names in the GROUP BY clause that do not appear in the SELECT list. When the WHERE clause is used with GROUP BY, the WHERE clause is applied first, then groups are formed from the remaining rows that satisfy the search condition.

The ISO standard considers two nulls to be equal for purposes of the GROUP BY clause. If two rows have nulls in the same grouping columns and identical values in all the nonnull grouping columns, they are combined into the same group.

Use of GROUP BY

Find the number of staff working in each branch and the sum of their salaries.

```
SELECT branchNo, COUNT(staffNo) AS myCount, SUM(salary) AS mySum
FROM Staff
GROUP BY branchNo
ORDER BY branchNo;
```

It is not necessary to include the column names **staffNo** and salary in the **GROUP BY** list, because they appear only in the **SELECT** list within aggregate functions. On the other hand, **branchNo** is not associated with an aggregate function and so must appear in the **GROUP BY** list.

Result:

branchNo	myCount	mySum
B003	3	54000.00
B005	2	39000.00
B007	1	9000.00

Conceptually, SQL performs the query as follows:

(1) SQL divides the staff into groups according to their respective branch numbers. Within each group, all staff have the same branch number. In this example, we get three groups:

branchNo	staffNo	salary
B003	SG37	12000.00
B003	SG14	18000.00
B003	SG5	24000.00
B005	SL21	30000.00
B005	SL41	9000.00
B007	SA9	9000.00

COUNT(staffNo)	SUM(salary)
3	54000.00
2	39000.00
1	9000.00

(2) For each group, SQL computes the number of staff members and calculates the sum of the values in the salary column to get the total of their salaries. SQL generates a single summary row in the query result for each group.

(3) Finally, the result is sorted in ascending order of branch number, branchNo.

Restricting groupings (HAVING clause)

The HAVING clause is designed for use with the GROUP BY clause to restrict the **groups** that appear in the final result table. Although similar in syntax, HAVING and WHERE serve different purposes. The WHERE clause filters individual rows going into the final result table, whereas HAVING filters **groups** going into the final result table. The ISO standard requires that column names used in the HAVING clause must also appear in the GROUP BY list or be contained within an aggregate function. In practice, the search condition in the HAVING clause always includes at least one aggregate function; otherwise the search condition could be moved to the WHERE clause and applied to individual rows. (Remember that aggregate functions cannot be used in the WHERE clause.)

The HAVING clause is not a necessary part of SQL—any query expressed using a HAVING clause can always be rewritten without the HAVING clause.

Use of HAVING

For each branch office with more than one member of staff, find the number of staff working in each branch and the sum of their salaries.

```
SELECT branchNo, COUNT(staffNo) AS myCount, SUM(salary) AS mySum
FROM Staff
GROUP BY branchNo
HAVING COUNT(staffNo) > 1
ORDER BY branchNo;
```

This is similar to the previous example, with the additional restriction that we want to consider only those groups (that is, branches) with more than one member of staff. This restriction applies to the groups, so the HAVING clause is used.

Result:

branchNo	myCount	mySum
B003	3	54000.00
B005	2	39000.00

END