SQL

(Note that the character '#' is used in examples throughout but in most cases is not legal in SQL.]

Simple SELECT

```
select    some data              (column name(s))
from      some table(s)          (table names(s))
where     condition              (retrieved tuple(s))
```

```
select s#, sname, city
from s
where status >= 20;
```

```
select * from s;
```

(we use * as shorthand for all columns of a table)

Creating a Table

```
create table tablename
     (columnname format {, columnname format})
```

```
create table s
     (s# char(4), sname varchar(20),
      status smallint, city varchar(10));
```

```
create table s
     (s# char(4),
      sname varchar(20) not null,
      status smallint default 10,
      city varchar(10)
     constraint spk
      primary key (s#));
```


For the table SPJ we might include

```
     constraint spjpk
      primary key (s#, p#,j#)
     constraint spjfk
      foreign key (s#) references s(s#)
```

Datatypes

Numeric

```
e.g. INTEGER, INT, FLOAT, REAL, DOUBLE, NUMERIC(i, j)
where i is precision (no. of decimal digits)and
j is scale (no. of digits after the decimal point)
```

Character-string

```
e.g. CHARACTER(n), CHAR(n)
     CHARACTER VARYING(n), CHAR VARYING(n), VARCHAR(n)
```

Boolean

```
has values TRUE and FALSE (and UNKNOWN represented as NULL !?)
```

Date/Time/Timestamp/Interval

```
e.g. DATE    yyyy-mm-dd
```

```
TIME   hh:mm:ss
DATETIME e.g. '2004-02-24 11:35:16.999'
TIMESTAMP
```

Bit-string

```
e.g. BIT(n), (BLOB, IMAGE)
```

Inserting rows into a table

```
insert into s
     values ('S8', 'STEIN', 20, 'SWANSEA');

insert into s (s#, sname)
     values ('S9', 'STEIN');
```

[Some DBMS implementations provide a **copy** command or similar function to allow an easy method to load data into a table from a file. Be careful with some as they may bypass the normal integrity constraint checks!]

Selecting specific rows and columns from a table

selecting columns

```
select s#, sname, city from s;
```

selecting rows

```
select * from s where city = 'LONDON';
```


The *where* clause causes the table to be searched and the data satisfying the *search-condition* is retrieved.

We can use **AND** and **OR** for multiple conditions (also **NOT**)

and we can use **= <> < <= > >=**
(and we can use **+ - * / **** in arithmetic expressions)

```
select sname, city from s
     where (city = 'LONDON' or city = 'PARIS')
     and status > 20;

     not (city = 'LONDON');
     city <> 'LONDON';
```

Functions

[A large number are usually available in most DBMS implementations.]

Type conversion

```
e.g.   cast(expr as datatype)
```

Numeric

```
e.g.   abs(n), sin(n), sqrt(n)
```

String

```
e.g.   concat(c1,c2), lower(c1), left(c1, n)
```

Date

```
e.g.   day(date), datediff(datepart, startdate, enddate)
```

Set (Aggregate)

```
e.g.   count, sum, avg, max, min
```

LIKE Operator

% matches any string of zero or more characters
_ matches a single character

```
... where sname like 'R%'
... where sname like '_R%'
... where module_code like 'CS-M__'

... where notes like '%20\% discount%' escape '\'
... where notes like '%20%% discount%' escape '%'
```

BETWEEN Operator

```
select s#, sname, city
from s
where status between 20 and 40;

[where status >= 20 and status <=40]
```

IN Operator

```
select s#, sname, city
from s
where status in (10, 30, 50);
```


IS NULL Operator

```
... where sname is null
... where sname is not null
[... where not (sname is null)]
```


Ordering Rows


```
select sname, status, city
from s
where status >= 20
order by status asc, city desc;
```

Distinct Rows

```
select city
from s;
```

```
select distinct city
from s;
```

Table Correlation Names (Range Variables)

```
select sx.sname
from s sx
where sx.status >= 20;
```

[By default, a range variable exists with the same name as each table in the query.]

Find the name of suppliers whose status is greater than the status of supplier S5

```
select s1.sname
from s s1, s s2
where s1.status > s2.status
and s2.s# = 'S5';
```

Querying Multiple Tables (Join Query)

What are the names of parts supplied by supplier S3?

(we could think of the solution like this:

```
select p#                          P#
from spj                  --->     P3
where s# = 'S3';                   P4
```

and then

```
select p#, pname                   P#      pname
from p                    --->     P3      SCREW
where p# = 'P3'                    P4      SCREW
or p# = 'P4';
```

But rather than two queries, we can use just one:

```
select p.p#, p.pname
from spj, p
where spj.s# = 'S3'
and spj.p# = p.p#;
```

This illustrates how SQL is non-procedural (like relational calculus); we simple state the conditions which define the data required.

Subqueries are, however, allowed


```
select p#, pname
from p
where p# =                         (we could have used in instead of =)
     (select p#
      from spj
      where s# = 'S3');
```


Set Operations – UNION, INTERSECT and DIFFERENCE (EXCEPT/MINUS)

```
select city from s union select city from j;

select city from s intersect select city from j;

select city from s except select city from j;
```

By default duplicates are removed from the result, however, duplicates are not removed when the word **all** is added to the operator e.g.

```
select city from s union all select city from j;
```

Renaming attributes in result relation

```
select city as supplier_city from s;
```

Aggregate Functions

[Several further aggregate operators were added by the "online analytical processing" (OLAP) amendment.]

```
select avg(qty) as average_quantity
from spj;
```

```
select city, count(s#) as howmany
from s
group by city;

select s#, avg(qty) as average_quantity
from spj
group by s#;

select *
from spj
where qty >
     (select avg(qty)
      from spj;
```

Get part numbers for parts supplied by more than one supplier.

```
select p#
from spj
group by p#
having count(distinct s#) > 1;
```

The **HAVING** clause is used to eliminate/select groups (just as the WHERE clause is used to eliminate/select rows).

DELETE Command

```
delete from s where s# = 'S5';

delete from s where city = 'PARIS';
```



```
delete from s;
```

(Ooops!, we just deleted all the rows from S!)

To remove a table completely from the database we use the **DROP TABLE** statement:


```
drop table s;
```

UPDATE Command

```
update s
set status = 5
where city = 'LONDON';

update s
set status = status + 5,
    city = 'SWANSEA'
where s# <> 'S3' or s# <> 'S5';
```

What exactly does that last update do? (Just testing you are awake?)
What would it do if the "`or`" was replaced with an "`and`"?

ALTER TABLE Command

```
alter table s
     add address varchar(100);

alter table s
     drop address;
```

Use of Subselects with Create Table and Insert

```
create table highstatus
     as select * from s where status >= 20;

create table spnames (suppname, partname)
     as select distinct sname, pname from s, spj, p
     where s.s# = spj.s# and spj.p# =p.p#;

insert into highstatus
     select * from s where status = 10;
```

Insert a supplier S6 STEIN SWANSEA with the same status as supplier S2

```
insert into s (S#, sname, status, city)
     select 'S6', 'STEIN', status, 'SWANSEA'
     from s where s# ='S2';
```

EXISTS Operator

The exists operator (corresponding to the existential quantifier from relational calculus) takes the form `exists (`*`subquery`*`)` and it evaluates to "true" if and only if the set represented by *`subquery`* is non-empty.


```
select distinct s.sname
from s
where exists (
     select * from spj
     where spj.s# = s.s# and spj.p# = 'P2');
```


(i.e. get the names of suppliers who supply part P2)

We can also use **`not exists`**.

SQL does not include direct support for the universal quantifier, FORALL, hence "FORALL-queries" typically have to be expressed in terms of EXISTS and double negation e.g. Get supplier names for suppliers who supply all the parts

```
select distinct s.sname
from s
where not exists (
     select * from p
     where not exists (
          select * from spj
          where spj.s# = s.s# and spj.p# = p.p#));
```

(i.e. get the names of suppliers such that there is not a part they do not supply)

ALL or ANY Conditions

(a) Which supplier makes at least one shipment with quantity >= 300?
(b) Which supplier makes all shipments with quantity >= 300?

```
select s# from spj spj1
where 300 <= any       [<= all  for (b)]
     (select qty from spj spj2
      where spj2.s# = spj1.s#);
```

Operator `some` is a synonym for `any`. The operator `= any` is equivalent to the operator `in`.

Joins

```
select s.s#, s.sname, spj.p#, spj.qty
from s, spj where s.s# = spj.s#
```

We could lose information if a supplier does not currently supply a part

```
select s.s#, s.sname, spj.p#, spj.qty
from s left join spj on s.s# = spj.s#;
```

We also have **`right join`** and **`full join`** (with optional use of word **`outer`**).
(First example above is an *inner* join.)

Views

Views (derived relations) are virtual tables. They operate like tables but hold no data of their own.
Views serve three main purposes:

- they simplify data access
- they provide data independence
- they provide data privacy

```
create view ls as
     select s#, sname, status
     from s
     where city = 'LONDON';
```


We can now use SQL to operate on `ls` as though it were a table


```
create view pqty (p#, pname, totqty) as
     select spj.p#, p.pname, sum(spj.qty)
     from spj, p
     where p.p# =spj.p#
     group by spj.p#, p.pname;
```



```
create view biqpqty as
     select *
     from pqty
     where totqty > 1500;
```

Views allow the **same** data to be seen by **different** users in **different** ways (at the **same** time).

To delete a view definition use `drop view` *`viewname`*

One approach to view implementation is *query modification*, for example,

```
select sname from ls where status > 10;
```

becomes

```
select sname from s where status > 10 and city = 'LONDON';
```

Updating Views
Updating of views is complicated and can be ambiguous. For example, … (see later handout on Updating Views).

Snapshots
(Commonly known as **materialized views** (and also indexed views) – a contradiction in terminology!?) They are real, not virtual. Normally held as a 'read only' relation and *refreshed* at regular intervals.

Null Values in Tuples

[The SQL 2003 Standard says the following about `NULL`.

The null value: Every data type includes a special value, called the *null value*, sometimes denoted by the keyword NULL. This value differs from other values in the following respects:

– Since the null value is in every data type, the data type of the null value implied by the keyword NULL cannot be inferred; hence NULL can be used to denote the null value only in certain contexts, rather than everywhere that a literal is permitted.

– Although the null value is neither equal to any other value nor not equal to any other value – it is *unknown* whether or not it is equal to any given value – in some contexts, multiple null values are treated together; for example, the <group by clause> treats all null values together.]

Nulls can have multiple interpretations, for example:

- the attribute *does not apply* to this tuple
- the attribute values for this tuple is *unknown*
- the value is *known but absent*, i.e. it has not been recorded yet.

not applicable e.g. house name in an address
unknown e.g. height in a person table
missing e.g. telephone number in a person table
(all three interpretations could apply to the telephone number)

Nulls can lead to problems with

interpretation

specifying *join* operations

accounting for them when *aggregate operators* (e.g. Count, Sum) are used

SQL does not differentiate between the different meanings of null. In general each null is considered to be different from every other null in the database. When a null is involved in a comparison operation, the result is considered to be *unknown* (it may be *true* or it may be *false*). Hence, SQL uses a three-valued logic with values true, false and unknown instead of the normal two-valued logic (with values true and false). It is therefore necessary to define the results of three-valued logical expressions when the logical connectives *and*, *or* and *not* are used.

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

NOT	
T	F
F	T
U	U

In select/project/join queries, the general rule is that only those combinations of tuples that evaluate the logical expression of the query to true are selected. Tuple combinations that evaluate to false or unknown are not selected. However, there are exceptions to that rule for certain operations, such as outer joins.

SQL allows queries that check whether an attribute value is null. Rather than using = or <> to compare an attribute value to null, SQL uses **is** or **is not**. [Many database implementations also provide a function such as *ifnull(a, b)* which returns the value *b* if the value of *a* is null, otherwise it returns the value of *a*.]