

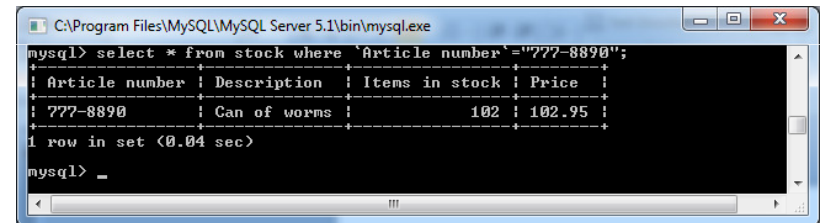
Lecture 2

- Finding stuff in tables
- the `where` clause
- keys

Finding records with `where`

- To find an individual record or group of records, we must tell `SELECT` how to identify them:

```
select * from stock
  where `Article number`="777-8890";
```



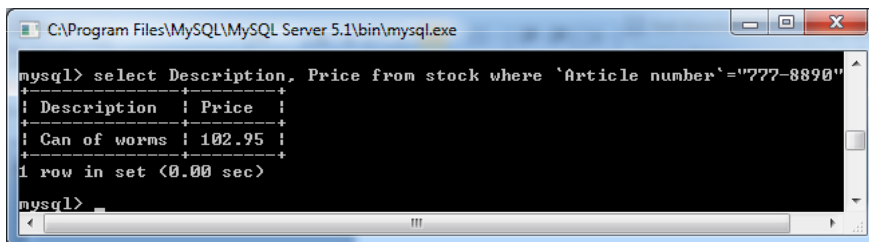
```
C:\Program Files\MySQL\MySQL Server 5.1\bin\mysql.exe
mysql> select * from stock where `Article number`="777-8890";
+-----+-----+-----+-----+
| Article number | Description | Items in stock | Price |
+-----+-----+-----+-----+
| 777-8890      | Can of worms | 102           | 102.95 |
+-----+-----+-----+-----+
1 row in set (0.04 sec)
mysql> _
```

- Note that the article number is a string and goes into straight quotes, and the equals operator is `=`, not `==`

More ways to use the `where` clause

- In combination with just a few fields:

```
select Description, Price from stock
  where `Article number`="777-8890";
```



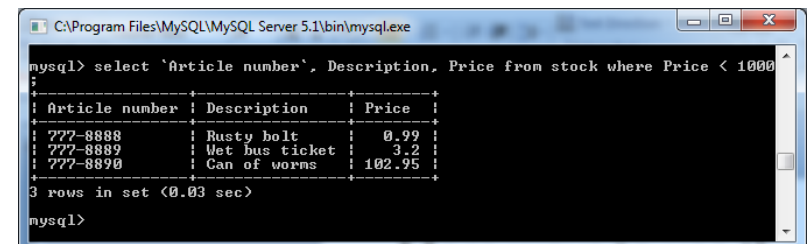
```
C:\Program Files\MySQL\MySQL Server 5.1\bin\mysql.exe
mysql> select Description, Price from stock where `Article number`="777-8890"
+-----+-----+
| Description | Price |
+-----+-----+
| Can of worms | 102.95 |
+-----+-----+
1 row in set (0.00 sec)
mysql> _
```

- Note: fields used in the `where` clause do not need to be on the select list!

More ways to use the `where` clause

- With different operators:

```
select
  `Article number`, Description, Price
from stock
  where Price < 1000;
```

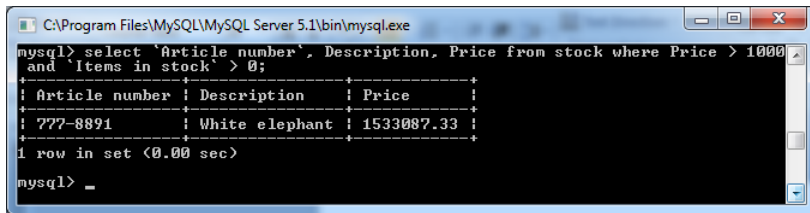


```
C:\Program Files\MySQL\MySQL Server 5.1\bin\mysql.exe
mysql> select `Article number`, Description, Price from stock where Price < 1000
+-----+-----+-----+
| Article number | Description | Price |
+-----+-----+-----+
| 777-8888      | Rusty holt | 0.99  |
| 777-8889      | Wet bus ticket | 3.2   |
| 777-8890      | Can of worms | 102.95 |
+-----+-----+-----+
3 rows in set (0.03 sec)
mysql>
```

More ways to use the where clause

- With more than one condition in the where clause:

```
select
  `Article number`, Description, Price
from stock
where Price > 1000
      and `Items in stock` > 0;
```

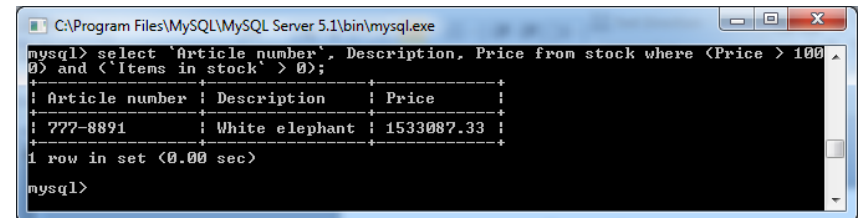


```
C:\Program Files\MySQL\MySQL Server 5.1\bin>mysql.exe
mysql> select `Article number`, Description, Price from stock where Price > 1000
and `Items in stock` > 0;
+-----+-----+-----+
| Article number | Description | Price |
+-----+-----+-----+
| 777-8891      | White elephant | 1533087.33 |
+-----+-----+-----+
1 row in set (0.00 sec)
mysql> _
```

Hate having to memorise operator precedence? Use parentheses!

- This works just as well in the where clause:

```
select
  `Article number`, Description, Price
from stock
where (Price > 1000)
      and (`Items in stock` > 0);
```



```
C:\Program Files\MySQL\MySQL Server 5.1\bin>mysql.exe
mysql> select `Article number`, Description, Price from stock where (Price > 1000)
and (`Items in stock` > 0);
+-----+-----+-----+
| Article number | Description | Price |
+-----+-----+-----+
| 777-8891      | White elephant | 1533087.33 |
+-----+-----+-----+
1 row in set (0.00 sec)
mysql>
```

Notes on quotes

- Different dialects of SQL use different types of quotes for strings
- I have so far used double quotes for strings to make it easier for you to distinguish them on the screen from the single quote backticks that denote field and table names, like here:

```
select * from stock
  where `Article number`="777-8890";
```
- However, this only works in MySQL and some other servers. Single quotes are more portable (ANSI-SQL compliant). So from now on, we'll use single quotes:

```
select * from stock
  where `Article number`='777-8890';
```

Finding an individual record in a table

- Consider:

```
select * from stock
  where `Article number`='777-8890';
```
- This makes an assumption: That there are no two articles with the same number. But nobody stops us from doing this:

```
insert into stock
  (`Article number`,`Description`,
   `Items in stock`,`Price`)
  values ('777-8890','Copy cat',123,99.90);
```
- This would give us a second entry with the same article number. So the select statement above would return a data set with two rows.

Adding a primary key to a table

- The problem of duplicate entries can be avoided by giving each entry a unique identifier. E.g., if the article number should be unique, then we can declare it a *primary key*.
- This can be done in two ways. If the table has not yet been created, we can add the primary key to the CREATE TABLE statement:

```
CREATE TABLE stock (  
  `Article number` varchar(8),  
  Description varchar(100),  
  `Items in stock` int,  
  Price double,  
  PRIMARY KEY (`Article number`)  
);
```

Adding a primary key to a table

- If the table has already been created, we can change its definition to add the primary key:

```
ALTER TABLE stock  
  ADD  
  PRIMARY KEY (`Article number`);
```

- Whichever way we choose, the RDBMS now regards the field ``Article number`` as a unique identifier for each record – so trying to insert a duplicate article number will result in an error

Why (primary) keys are good for you

- Consider a large database table with N records, where N is very large
- Remember we cannot rely on the order in which records are stored in the table
- So what strategy can a RDBMS use to find a record with a given attribute value (field value)?
- Answer: If the field is not a key, the RDBMS has to check *each individual record* in the table. This takes time – $O(N)$ in fact!
- Having a key means that the RDBMS can build an index for the table, permitting fast location of entries with a particular key value. Lookup is now only $O(\log(N))$.

Why too many keys are bad for you

- So should you have a key for each field in a table? No!
- Building the index takes a little extra time whenever a record is inserted into a table, or when it is updated
- Rule of thumb: turn only such fields into keys that will actually foreseeably be used in lookups.
- How many times are we likely to have to find an individual item in stock based on the number of items in stock? Probably never – so this probably doesn't need to be a key!
- NB: There can only be one primary key per table!

Composite keys

- Consider the following table, designed to store the text in a book such that every record stores the text of exactly one subsection:

```
CREATE TABLE book (  
    chapter int,  
    section int,  
    subsection int,  
    subsectionText text  
);
```

- In a book, none of these fields are necessarily unique on their own. However, the combination of chapter, section, and subsection should be unique

Composite keys

- We can create a composite primary key like this:

```
CREATE TABLE book (  
    chapter int,  
    section int,  
    subsection int,  
    subsectionText text,  
    PRIMARY KEY  
        (chapter, section, subsection)  
);
```

- Note that such composite keys are not necessarily a good idea – often it is better to add an extra field with a unique ID for each record and use this as a primary key

Autoincrement fields

- Often, we don't have a natural candidate for a unique ID among our fields and want to create one. For the **book** table, we could do it like this:

```
CREATE TABLE book (  
    textSnippetId bigint unsigned autoincrement,  
    chapter int,  
    section int,  
    subsection int,  
    subsectionText text,  
    PRIMARY KEY (textSnippetId)  
);
```

- Whenever we insert a new record into the **book** table, the new record is automatically given a **textSnippetId**. We do not have to specify the field or the value in the INSERT statement.

INSERT with autoincrement

- Inserting into the **book** table with autoincrement is done as follows:

```
INSERT INTO book  
    (`chapter`, `section`,  
    `subsection`, `subsectionText`)  
VALUES  
    (1,1,1, 'Once upon a time, ...');
```

- Note that we do not specify a `textSnippetId` here!

Notes on autoincrement fields

- Note that I used a `bigint` rather than an `int` as the datatype. Why?
- Autoincrement values are generally not recycled if a record is deleted from a table
- So if the book table saw a lot of INSERT and DROP statements (have yet to discuss these!), then we could one day run out of numbers even if the table itself never gets that big.
- On MySQL: `int unsigned` goes up to 4294967295. `bigint unsigned` goes up to 18446744073709551615
- Autoincrement fields must generally also be keys

Today's lab sheet

- Finding a particular entry in a table with a WHERE clause
- Adding a field with a unique identifier and turning it into a primary key that will auto-increment
- Retrieving records that meet a particular condition