Section slides: <http://webdev.slides.com/coltsteele/mysql-97-98#/43>

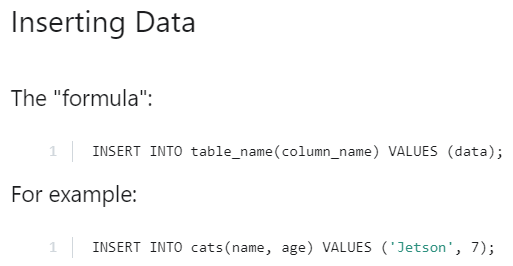
Inserting Data

* The INSERT INTO command is used to manually insert data into tables
  + When inserting data, one specifies the table name, the names of the columns that are being inserted into, and the values that are being inserted into those columns
  + The *order* of the column names and data inserted, as well as their datatypes, must match. You can declare and insert data in any order, as long as you are consistent between the names and data
  + In the example below, we insert into the *cats* table a cat named “Blue” whose age is 1. We’ll also add another cat named Draco, where we reverse the order in which we do the insertion





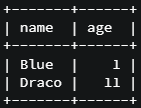
* Code for insertion of data:



Selecting Data

* How do we know that our data insertion worked? In most cases, the best way is to use the **SELECT** command, which we’ll dive more into later in the course
* For now, to view the data in a given table, use **SELECT \* from <table name>**



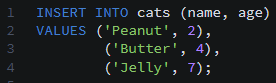


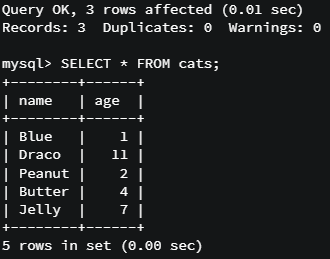
* Code review for data selection from a table



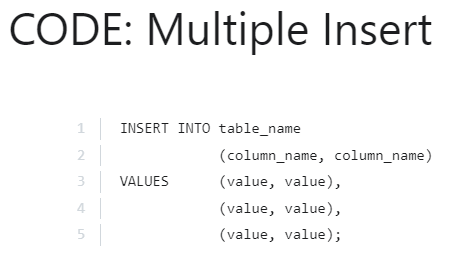
Multiple Inserts

* The INSERT command allows you to bulk insert data rather than just entering one at a time



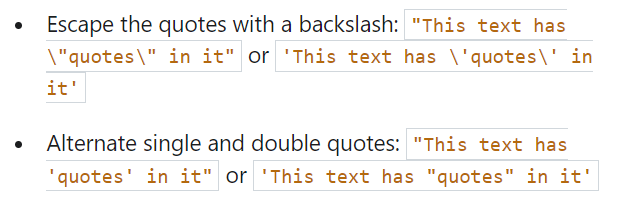


* Code summary for multiple inserts

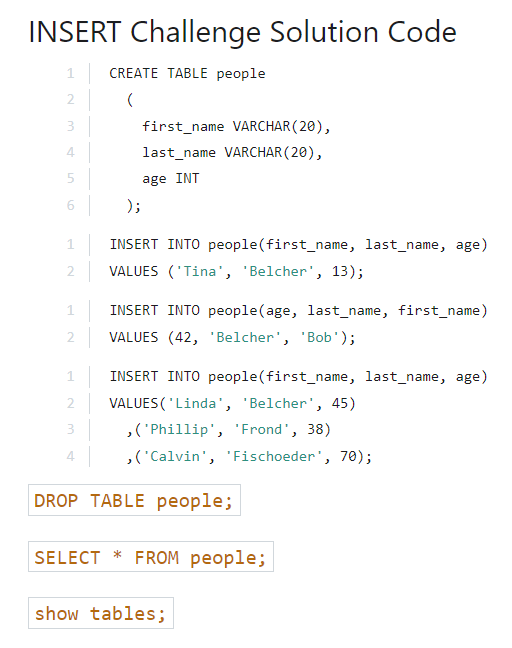


Inserting Literal Quotes into Values

* Normally, quotes are used to indicate that a value is to be a string. Sometimes however you want to have a literal single or double quote or quotes in your value. There are a couple ways to do this:



INSERT Challenge Code Solution



set sql\_mode='';

MySQL Warnings

* Let’s try the following insertion. Note that when the code is executed you get a warning

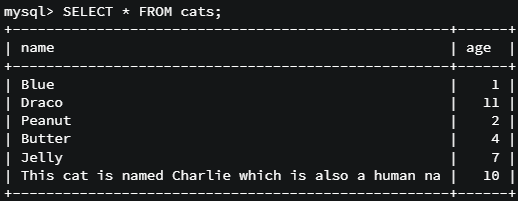




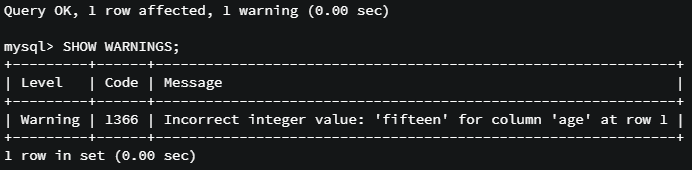
* Why did this happen? Let’s find out with the **SHOW WARNINGS;** command



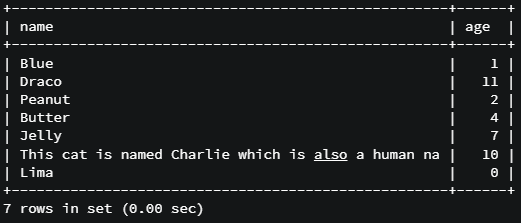
* + We see that some data was truncated for the “name” column. Let’s inspect the table to see what the truncation is.



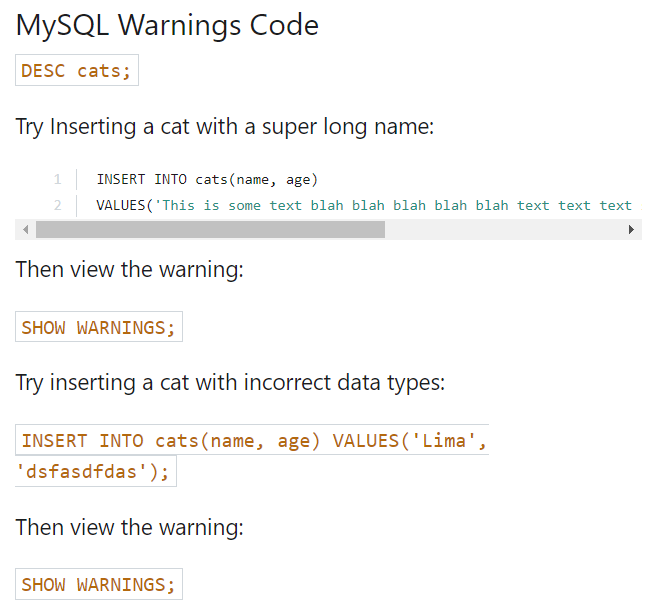
* + We see that the long text we tried to insert as the name for this cat has been truncated. This occurred because the datatype for this column is VARCHAR(50) and can only handle up to 50 characters
* Let’s look at another example where we insert a string for *age*:



* + This time we got a different warning, “Incorrect integer value”. So if we couldn’t properly insert an integer, what did we insert?

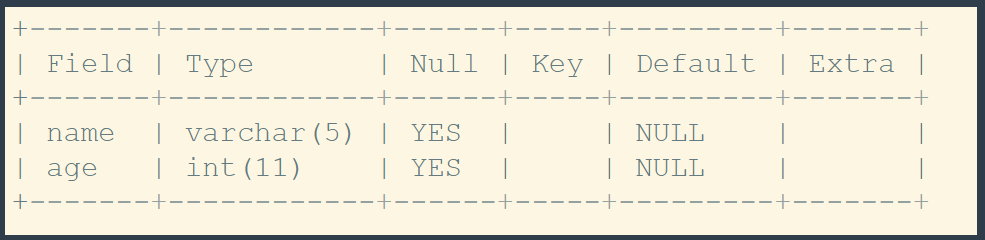


* + We see that the invalid value inserted for age was defaulted to 0 (zero)
* The take-home message is to view your warnings if you see them to figure out what went wrong
  + View warnings right away because they will disappear if you execute any valid commands afterward
  + When building apps in MySQL in, for example, JavaScript or Ruby, there will be warnings to let us know when things are going wrong. But in the shell you have to view these warnings manually
* Code summary: MySQL Warnings



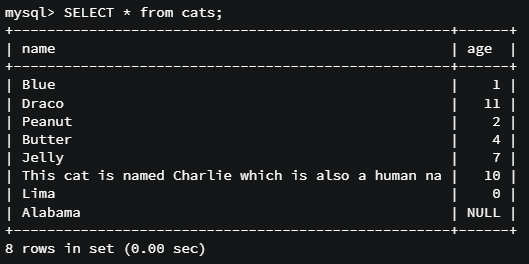
NULL and NOT NULL

* When we’ve created tables and examined them using **DESC;** we’ve seen something like this.



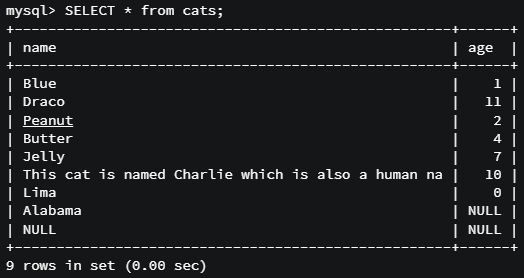
* What is this NULL column and what does it mean? **NULL** essentially means that the value is *not known*. Null does NOT mean zero.
  + In the description table, having “YES” under Null means that null values are permissible
* To illustrate, we can insert a cat name into the *cats* table without an age.
  + Note that we cannot declare something and not insert it. To insert only a name, we have to declare only a name



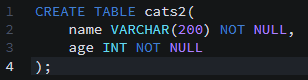


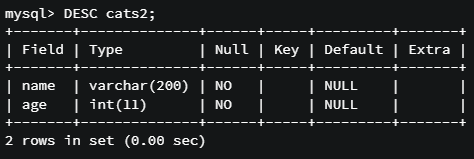
* + Check out that “age” value for the cat named Alabama. Note how it is NULL, which makes sense because we did not give it a value.
* A more extreme example – inserting nothing:





* How do we impose a requirement that a value *not* be null? We use **NOT NULL**
  + When defining a table, we specify that value as a NOT NULL

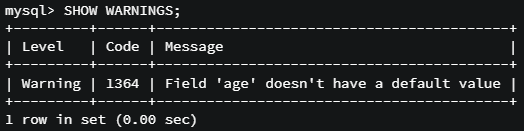




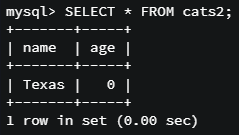
* + So now these two values are not allowed to be NULL. Let’s put it to the test by attempting to insert a name without an age





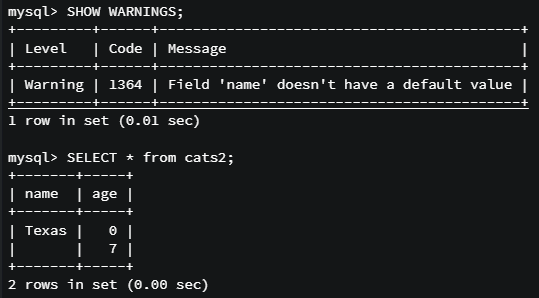


* + The value is not allowed to be NULL, and it does not have a default value (more on that during the next lecture). So what does it do?

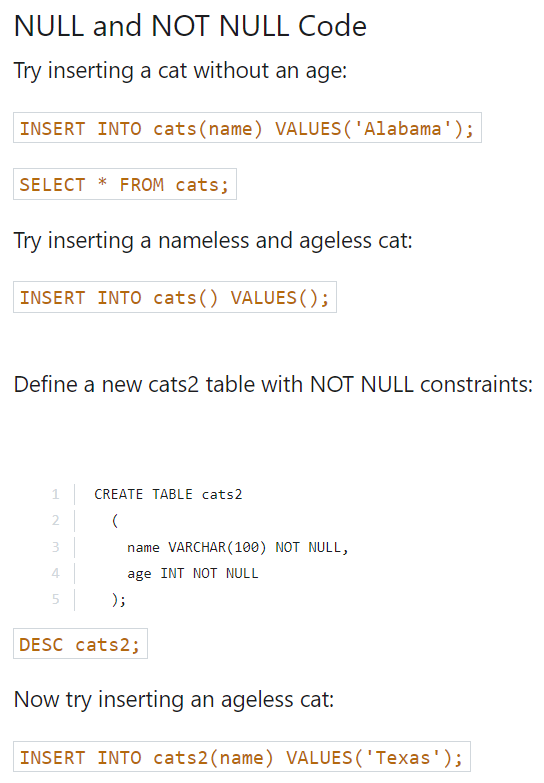


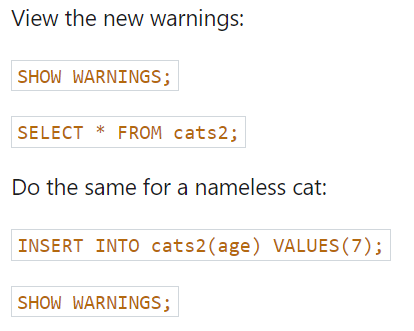
* + The age value defaulted to zero, as it cannot be NULL and the program did not know what else to do with it
* Another example: inserting age without a name





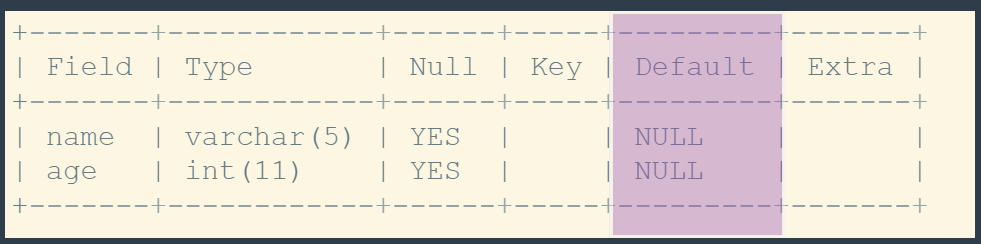
* + Note the difference here – the “name” field cannot be NULL, and so when we fail to provide the value to the INSERT INTO command, we get an empty string instead
    - An empty string is NOT a null value
* Code summary: NULL and NOT NULL



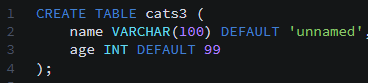


Setting Null Values

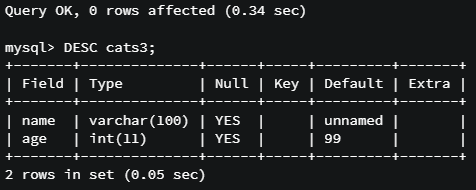
* It can be very helpful to specify default values that the SQL DBMS can fall back on if such values are not provided. Consider this description of our simple *cats* table, where the Default value is NULL



* To set default values, we do the following:

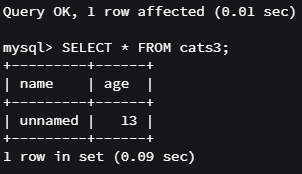


* + Using DESC, we sse this:

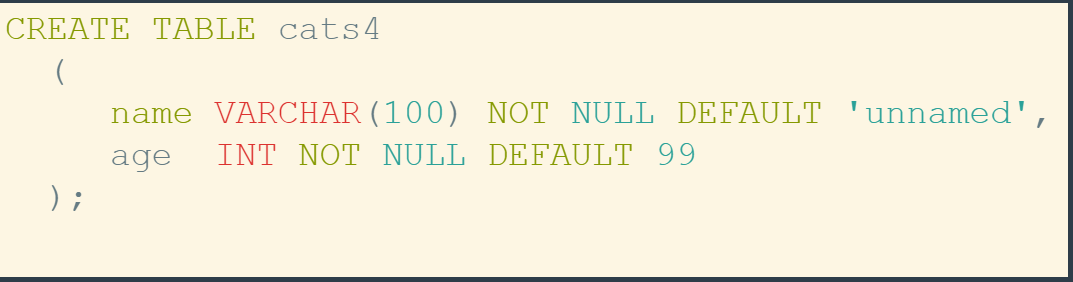


* Now let’s try an insertion where we provide only an age and no name:



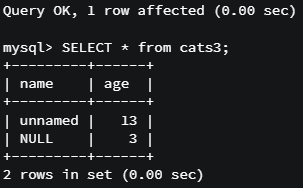


* + And viola, we see that the new entry has an age of 13 as we indicated, but the name has defaulted to “unnamed”
* Take a look at this other approach for declaring default values. Is there redundancy here? After all, if we declare a default value, wouldn’t it already be NOT NULL?

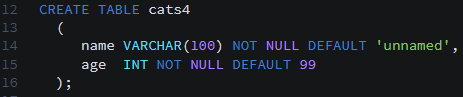


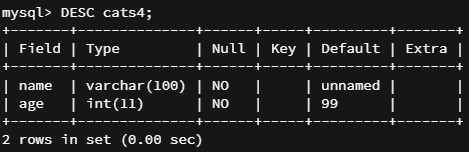
* + The answer is that the statement above is *not redundant* because you can manually and explicitly set a value to NULL so long as NULL values are allowed. For example:





* To prevent manual insertion of NULL values, we must declare columns as NOT NULL. Consider this example where we create a table called *cats4* that has default values on its NOT NULL columns



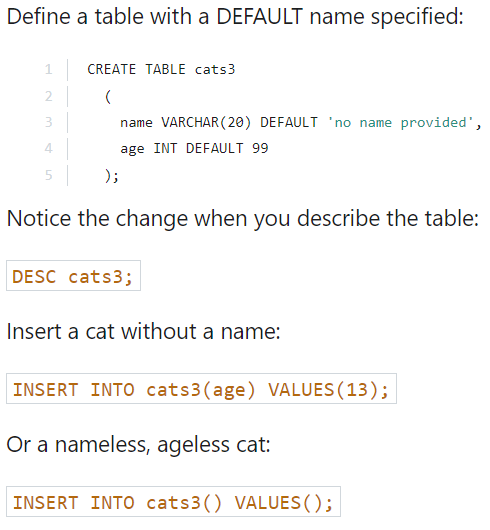


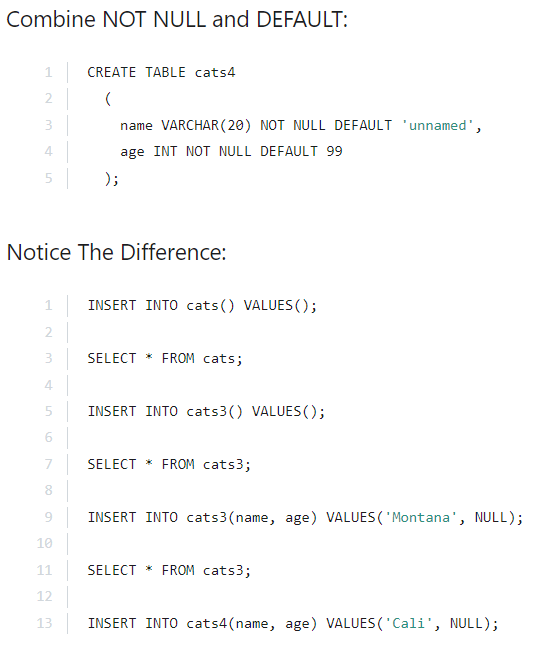
* + Now let’s try to insert a NULL value





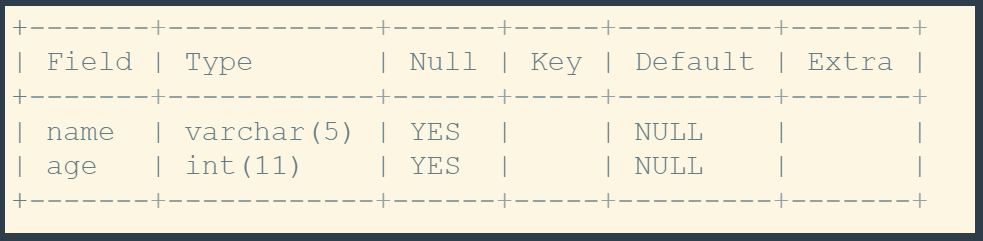
* + And there we have it. The “age” column cannot be a null value, and so SQL throws an error
* NOT NULL will be used pretty frequently because there will be instances where you never want a value to be NULL
* Code summary: setting default values





A Primer on Primary Keys

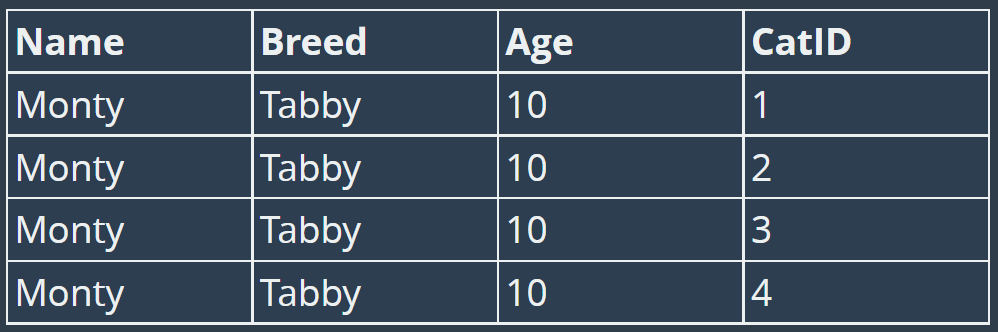
* You may have noticed the “key” column in the table description. What’s that all about?
  + This lecture is just an intro, we’ll be revisiting it often



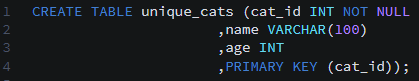
* Keys come into play for data integrity. So far in this course, we were able to repetitive data in our tables. In the table below, we have 4 separate cats that have the same name, breed, and age

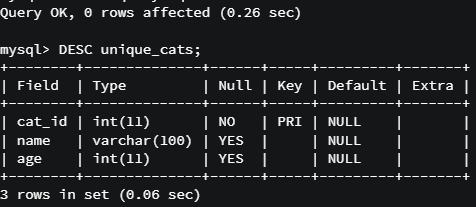


* The problem is that because we have 4 separate cats, we want each line of data to be *uniquely identifiable*
  + Uniquely identifiable data entries are a fundamental aspect of databases
    - Consider a database of usernames and passwords where some people have the same username
  + That does not mean that you cannot have data entries that have the same values. For instance, an animal shelter could easily have two animals of the same species that are the same age (e.g. a litter of strays)
* To get around this issue, a popular approach is to assign each data entry a unique ID. The precise term for a unique ID is a **primary key**



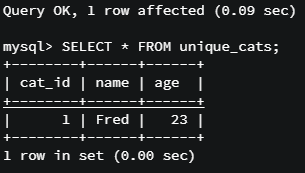
* + A **primary key** is a unique identifier that distinguishes a row of data in a table from all other rows within that table
  + We will see another type of key later called a **foreign key**, but more on that later
* We can assign primary keys using the following syntax. Here we create a column called “cat\_id” and assign it as the primary key



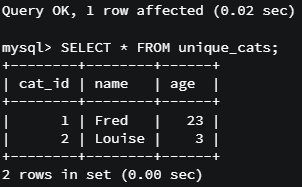


* + Notice now how the “Key” column in the description table has **PRI** associated with the “cat\_id” field, indicating that “cat\_id” is the primary key
  + Let’s now try an insertion into this table. We’ll insert two cats name Fred and Louise







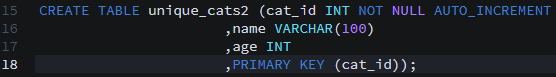


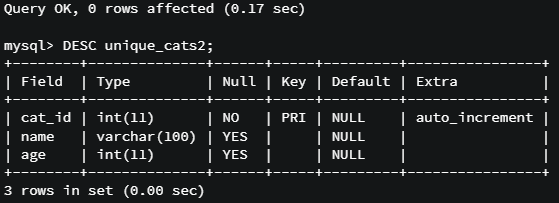
* + Now, what if we try to enter a new cat and re-use a cat\_id?





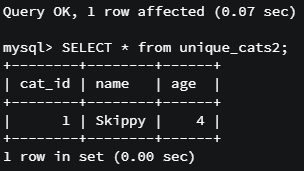
* + Ah ha! We get an error because the primary key cannot be duplicated
* Primary keys need not necessarily be numeric values. It can be any value that will be unique for each row of data
* It is impractical to manually specify a new ID for each new row of data. Imagine a database with tens of thousands of rows. Are you really going to remember where you left off in order to assign a new key to a new row? No way.
  + Instead, we can **auto-increment** our primary key so that it automatically increases the primary key each time a new entry is created
  + When doing this, you do not need to specify a value for the primary key – the system will handle that for you





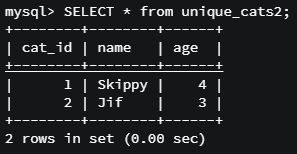
* + Now let’s try another insertion, where we do not specify the cat\_id:



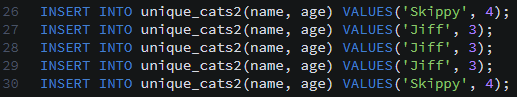


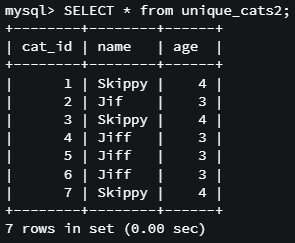
* + We see that cat\_id was automatically assigned a value of 1! Let’s add another cat to make sure the increment works:





* + Very nice. Also, if you were to use the same name and age for two different cats, the cat\_id would still be different





#### CODE: Primary Keys

Define a table with a PRIMARY KEY constraint:

1. CREATE TABLE unique\_cats
2. (
3. cat\_id INT NOT NULL,
4. name VARCHAR(100),
5. age INT,
6. PRIMARY KEY (cat\_id)
7. );

DESC unique\_cats;

Insert some new cats:

1. INSERT INTO unique\_cats(cat\_id, name, age) VALUES(1, 'Fred', 23);
3. INSERT INTO unique\_cats(cat\_id, name, age) VALUES(2, 'Louise', 3);
5. INSERT INTO unique\_cats(cat\_id, name, age) VALUES(1, 'James', 3);

Notice what happens:

SELECT \* FROM unique\_cats;

Adding in AUTO\_INCREMENT:

1. CREATE TABLE unique\_cats2 (
2. cat\_id INT NOT NULL AUTO\_INCREMENT,
3. name VARCHAR(100),
4. age INT,
5. PRIMARY KEY (cat\_id)
6. );

INSERT a couple new cats:

1. INSERT INTO unique\_cats2(name, age) VALUES('Skippy', 4);
2. INSERT INTO unique\_cats2(name, age) VALUES('Jiff', 3);
3. INSERT INTO unique\_cats2(name, age) VALUES('Jiff', 3);
4. INSERT INTO unique\_cats2(name, age) VALUES('Jiff', 3);
5. INSERT INTO unique\_cats2(name, age) VALUES('Skippy', 4);

Notice the difference:

SELECT \* FROM unique\_cats2;