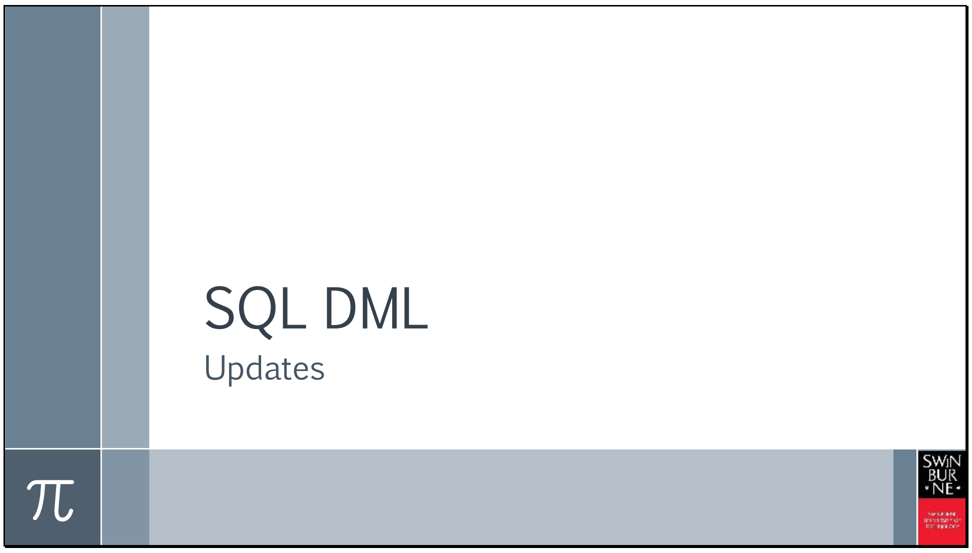
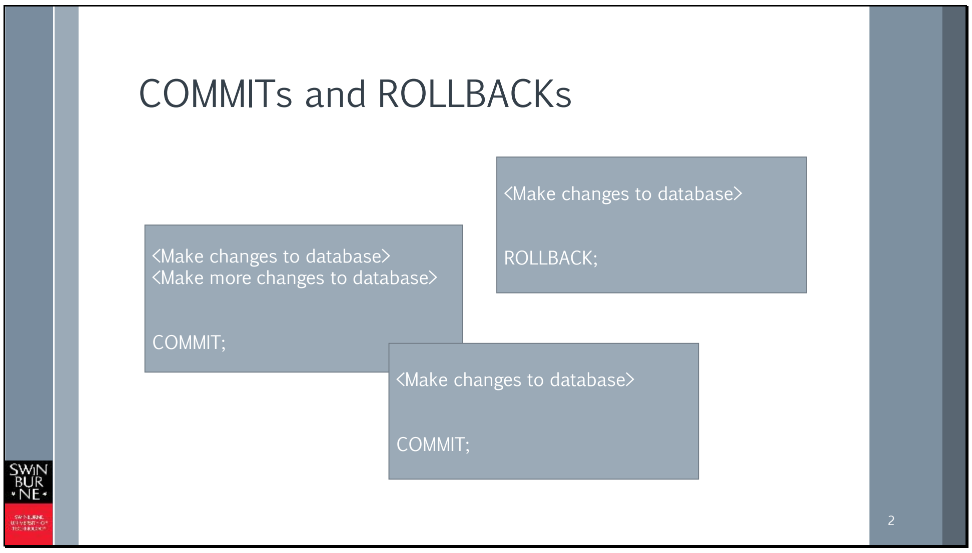


SQL DML

Updates

In this presentation we'll learn how to make changes to relational databases.

Slide 2



COMMITs and ROLLBACKs

<Make changes to database>
<Make more changes to database>

COMMIT;

<Make changes to database>

ROLLBACK;

<Make changes to database>

COMMIT;

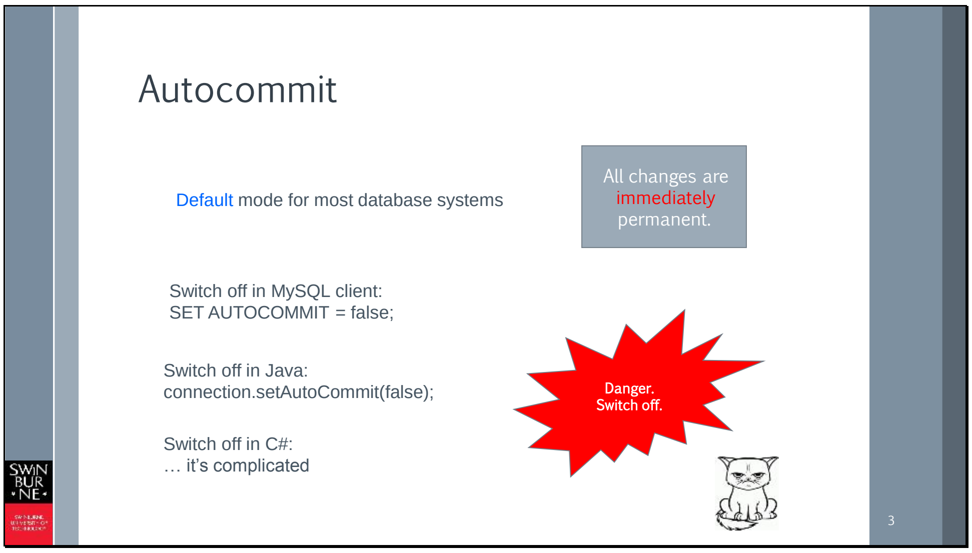
2

COMMIT and ROLLBACK are core concepts in the relational database world.
All applications we use these days come with an undo button, but this is quite different. With databases, every change you make is temporary until you issue a COMMIT.
This is because unlike your desktop files, any change you make to the database is actually visible to all other database users as soon as they have been made permanent. The changes to the data change the state of the database. For this reason, database systems give you the option of making changes on a tentative basis. When you are sure the effect is what you want, you make the changes permanent and everyone can see them. Until you issue a COMMIT or ROLLBACK, all your changes are only temporary and mostly visible only to yourself.

COMMIT means you commit to the changes. There is nothing more you can do about them after the commit, they are final. Commit is the "I do" of the database world.

ROLLBACK means the changes are discarded. They never actually existed. Nobody should ever have seen them.

Slide 3



Autocommit

Default mode for most database systems

All changes are immediately permanent.

Switch off in MySQL client:
SET AUTOCOMMIT = false;

Switch off in Java:
connection.setAutoCommit(false);

Switch off in C#:
… it's complicated

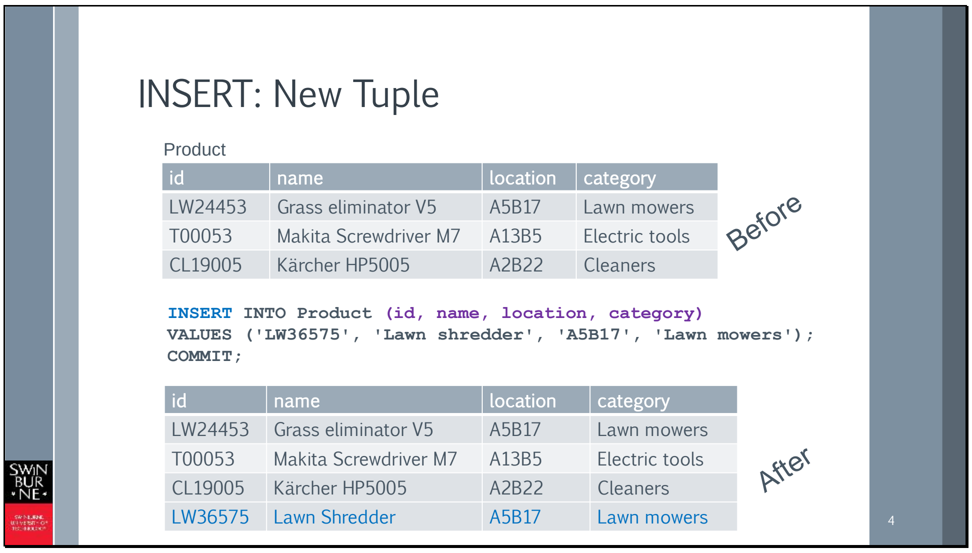
Danger.
Switch off.
3

Most databases immediately commit every change you make unless you tell them otherwise. This happens regardless of whether you are working from an SQL client like the SQL workbench or using SQL that is embedded in a program. Autocommit actually affects even SELECT statements, except that with queries you mostly don't notice the difference because queries make no changes to the database.

Whenever you make any changes, be sure to switch off autocommit first. If you don't do this, there is no undo button. If you make a mistake, correcting it can be time-consuming and even impossible. Think about working on a production database – if you make a mistake and this mistake is immediately committed, other users will make their changes based on your changes before you have time to correct the mistake. So do not work in autocommit mode unless you are running queries only.

Switching autocommit mode off is tantamount to using transactions. Transactions are an important but slightly complex concept we are going to look into in the next module. The .NET framework, for example, makes you use transactions to get out of autocommit mode. That's why the syntax is slightly more complicated than in Java or on the SQL console.

Slide 4



INSERT: New Tuple

Product

id	name	location	category
LW24453	Grass eliminator V5	A5B17	Lawn mowers
T00053	Makita Screwdriver M7	A13B5	Electric tools
CL19005	Kärcher HP5005	A2B22	Cleaners

```
INSERT INTO Product (id, name, location, category)
VALUES ('LW36575', 'Lawn shredder', 'A5B17', 'Lawn mowers');
COMMIT;
```

id	name	location	category
LW24453	Grass eliminator V5	A5B17	Lawn mowers
T00053	Makita Screwdriver M7	A13B5	Electric tools
CL19005	Kärcher HP5005	A2B22	Cleaners
LW36575	Lawn Shredder	A5B17	Lawn mowers

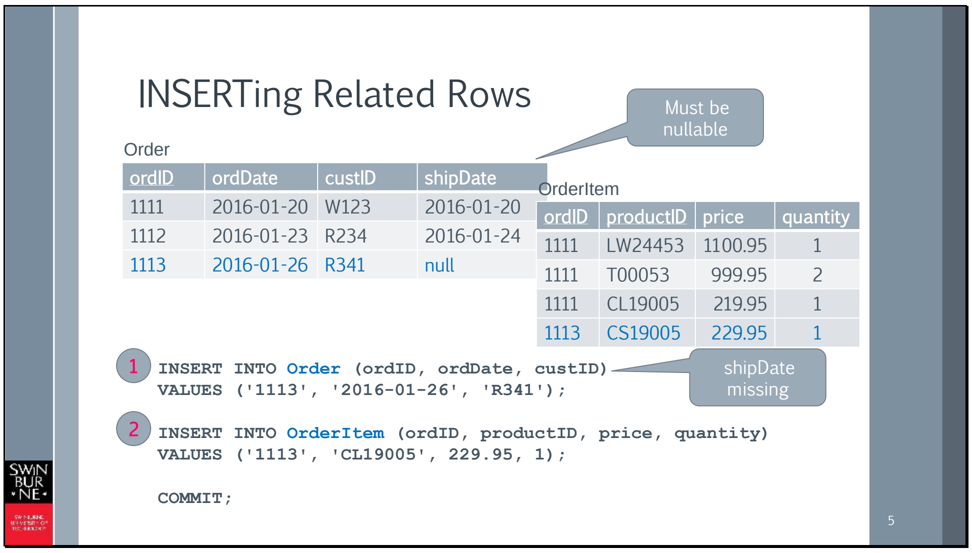
Whenever you need to add a new record to a database, you issue an INSERT into a table. INSERTs can only be issued against a single table at a time. If we want to insert into several tables, we have to issue several INSERTs.

The values are put into the columns named in the listing shown in purple. So the order of the attribute names and the attribute values has to be the same. Listing the attributes that we want to INSERT values for (shown in purple) is entirely optional. But if you don't state the attribute names, the database systems will put the values into the columns in the order the columns appear in the table. This depends on your Create table statement. If you get the values mixed up, and if the data types happen to match, the values will go in the wrong columns. You should therefore always list the attribute names in your INSERT statements and make sure the values are listed in the exact same order.

When we run only a single INSERT, you may wonder why we still have to switch off autocommit. If you are working in an SQL client, you can run a query that shows you the table after the INSERT. You may notice a problem then. So you can still decide whether to ROLL BACK when you have seen the effect of your INSERT. If you are working in autocommit mode, you have to issue a correction, which is messy.

When you need to INSERT new values into several tables, autocommit really is a problem, because the entries may be related, possibly even connected through a foreign key. In that case, a single entry in a table may be missing matching entries in another table. INSERTing related data into several tables first and then issuing a COMMIT at the end is a much better approach.

Slide 5



INSERTing Related Rows

Order

ordID	ordDate	custID	shipDate
1111	2016-01-20	W123	2016-01-20
1112	2016-01-23	R234	2016-01-24
1113	2016-01-26	R341	null

OrderItem

ordID	productID	price	quantity
1111	LW24453	1100.95	1
1111	T00053	999.95	2
1111	CL19005	219.95	1
1113	CS19005	229.95	1

Must be nullable

1
```
INSERT INTO Order (ordID, ordDate, custID)
VALUES ('1113', '2016-01-26', 'R341');
```
shipDate missing

2
```
INSERT INTO OrderItem (ordID, productID, price, quantity)
VALUES ('1113', 'CL19005', 229.95, 1);

COMMIT;
```

5

When you need to make a change to your database that affects many tables, having autocommit switched off is particularly important. Here you make all the additions to all the related tables and then commit everything at once. This ensures that no one can see an order that has no items in it.

Speaking of orders without items, do you think it matters which of these INSERTs is carried out first? Hopefully it does, because if it doesn't, it means you haven't implemented the foreign key relationships properly. First, let's think about which one is the parent and which is the child table. Both tables have the ordID column, but in which table is it the primary key? The fact that some ordID values have duplicates in the OrderItem table is a dead giveaway. Primary keys can't have duplicates, so ordID in Order is the primary key and ordID in OrderItem is the foreign key. But what could be the primary key in OrderItem? I'm sure you can figure it out based on what you already know.

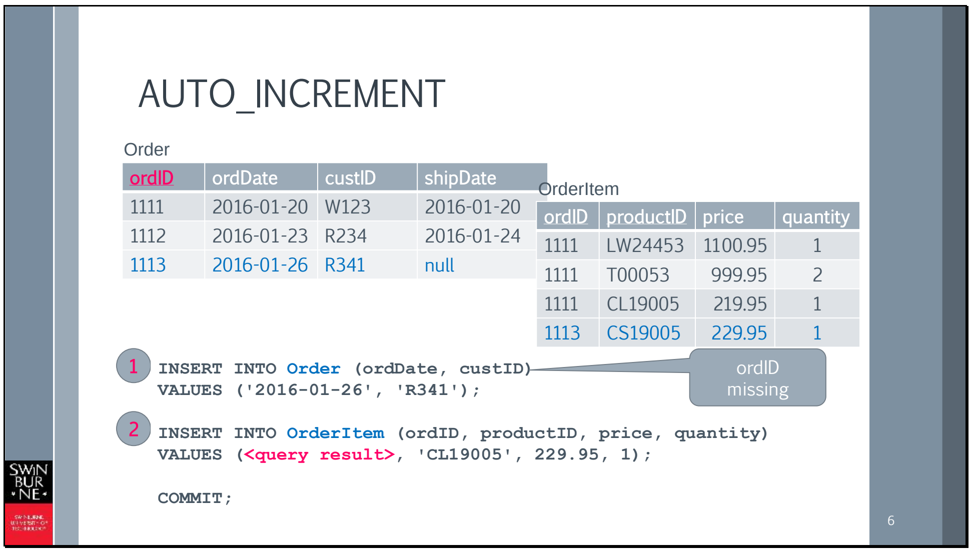
If we have implemented a proper foreign key relationship, the parent table entry has to be made first, so that there is an entry in the primary key column that the foreign key can reference. If there is a proper foreign key constraint in the child table, and a not null condition on the foreign key column, it is not possible to make the INSERT into the child table before the entry into the parent table.

Another consideration here is that the record for the Order table does not include a value for the shipDate. If you have declared the shipDate as not null, this INSERT statement will fail. With the Order table, we are assuming that we may not know the shipping date at the time we make the order, so the shipDate has to allow nulls.

Every INSERT statement has to have values for all fields that are not nullable, or the statement will fail.

When we have committed, the tables look like this. Note the primary keys are now underlined.

Slide 6



AUTO_INCREMENT

Order

ordID	ordDate	custID	shipDate
1111	2016-01-20	W123	2016-01-20
1112	2016-01-23	R234	2016-01-24
1113	2016-01-26	R341	null

OrderItem

ordID	productID	price	quantity
1111	LW24453	1100.95	1
1111	T00053	999.95	2
1111	CL19005	219.95	1
1113	CS19005	229.95	1

1
```
INSERT INTO Order (ordDate, custID)
VALUES ('2016-01-26', 'R341');
```
ordID missing

2
```
INSERT INTO OrderItem (ordID, productID, price, quantity)
VALUES (<query result>, 'CL19005', 229.95, 1);

COMMIT;
```

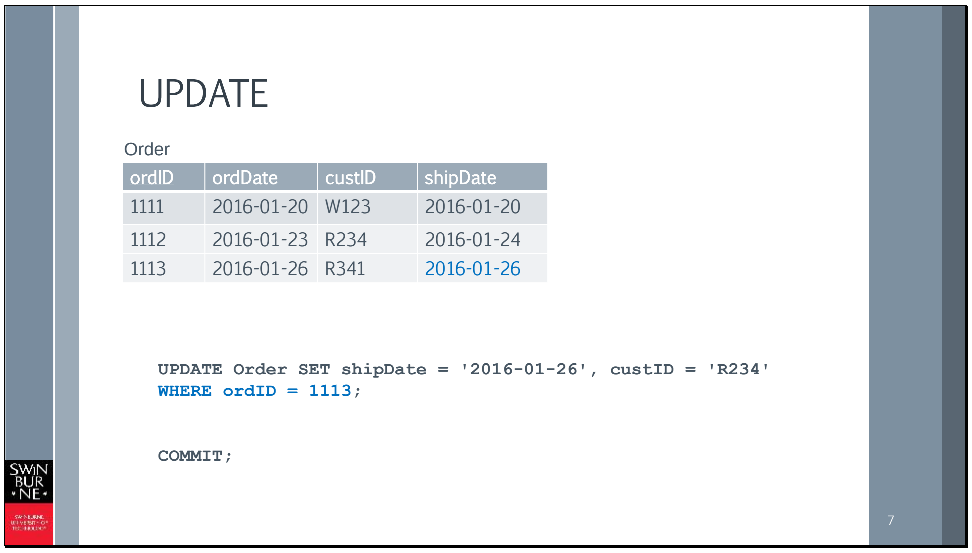
6

All full-fledged database products let you declare a column as an auto_increment column. This means the database system picks the value itself and ensures it is a consecutive number that is no duplicate. This is a convenient way of making unique primary keys.

A word of warning: Do not try to implement this yourself by using the MAX() function and increment the value. The problem with this approach is that it is very hard to get right with the concurrency models of most database systems. Over time you will have interference from other processes executing the same code. In the worst case this leads to duplicates. In the best case the INSERT will fail because it violates the primary key uniqueness constraint, if you have implemented one.

But if you use the auto_increment functionality, how do you find the autogenerated key for your second insert statement where it appears as a foreign key? The answer is that every database system has a mechanism that lets you retrieve the autogenerated number that was allocated in the same transaction (before the commit statement). You just have to execute this query to find the correct number for your foreign key.

Slide 7



UPDATE

Order

ordID	ordDate	custID	shipDate
1111	2016-01-20	W123	2016-01-20
1112	2016-01-23	R234	2016-01-24
1113	2016-01-26	R341	2016-01-26

```
UPDATE Order SET shipDate = '2016-01-26', custID = 'R234'
WHERE ordID = 1113;


COMMIT;
```

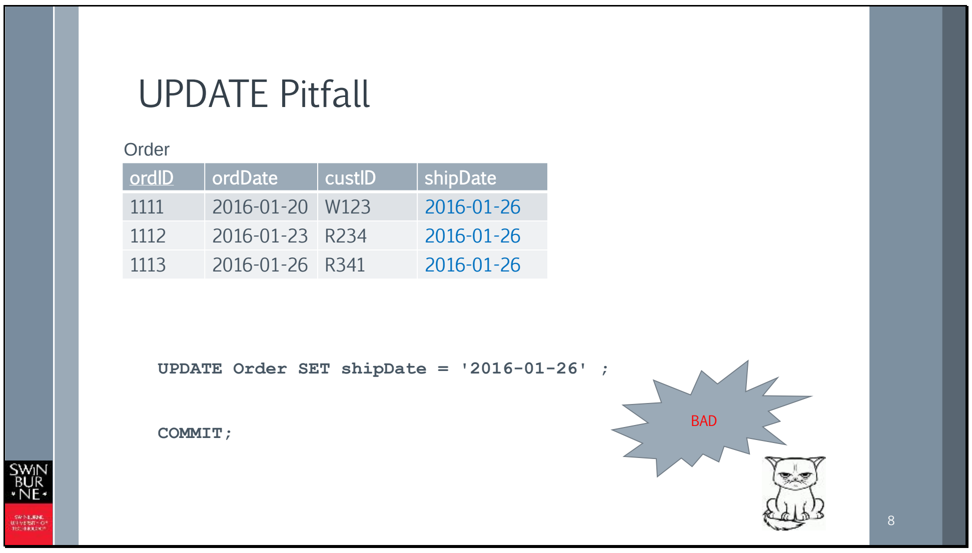
7

UPDATE is the statement you use for making changes to existing database entries. For example, when we made a new Order entry, we left the shipDate as null, because we didn't know yet when the order would be shipped. Once we have made the shipment, we can add the shipDate. To do this, we have to use an UPDATE statement.

A value doesn't have to be null to be updated. You could also update the customer ID if you had made a mistake there in the first place. Any value can be updated.

The important thing is that you don't forget the restriction.

Slide 8



UPDATE Pitfall

Order

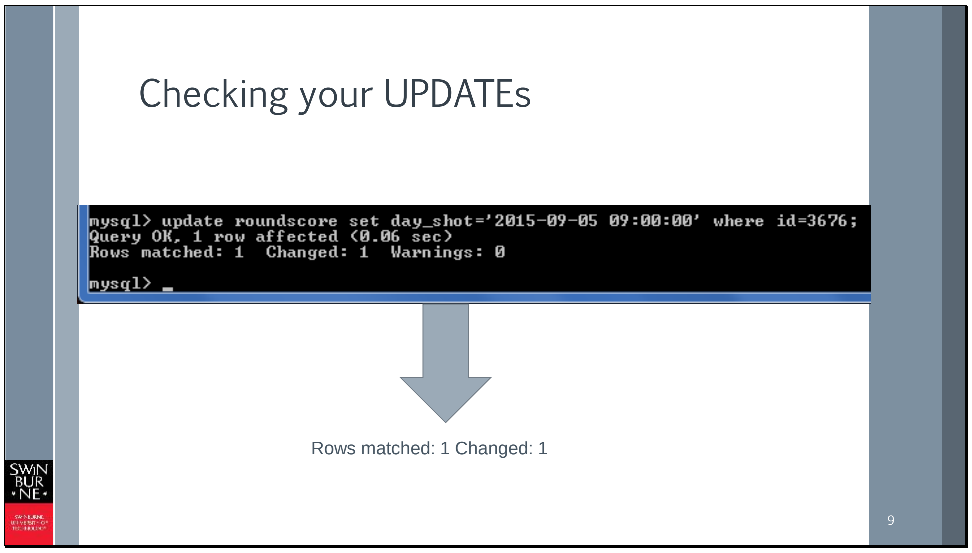
ordID	ordDate	custID	shipDate
1111	2016-01-20	W123	2016-01-26
1112	2016-01-23	R234	2016-01-26
1113	2016-01-26	R341	2016-01-26

```
UPDATE Order SET shipDate = '2016-01-26' ;

COMMIT;
```

With update statements, the most common mistake to make is to forget the restriction. How many shipDates will be updated with this statement? That's right, every shipDate in the table will be updated to this new date. This is really bad if you haven't disabled autocommit. How would you correct this mistake? How do you find all the actual shipDates? You would have to resort to a backup, which wouldn't have all the data – it would be missing the rows that were added since the backup was made. A nightmare.

UPDATES without restrictions are such a common problem MySQL actually asks you for confirmation if you forget the WHERE clause with an UPDATE statement. This is unusual – normally databases expect you to know what you are doing.

Slide 9



Checking your UPDATEs

```
mysql> update roundscore set day_shot='2015-09-05 09:00:00' where id=3676;
Query OK, 1 row affected (0.06 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> _
```

Rows matched: 1 Changed: 1

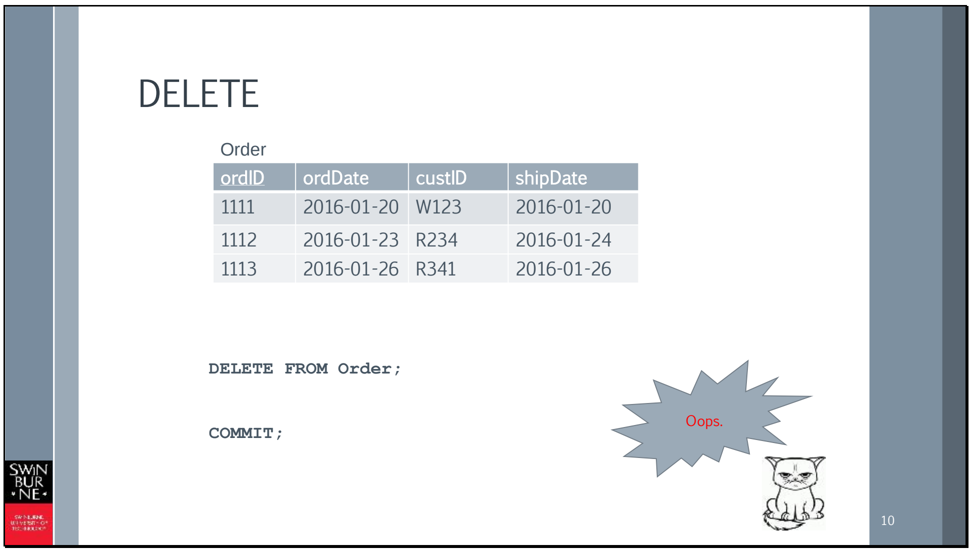
9

Ultimately, all your INSERTs and UPDATEs will probably go into some application code. But before you add them to the application, it's good practice to try them out from an SQL client. Ideally, you would set up a test environment rather than trying your commands on the production database.

Nonetheless, even if you are working on a test database, you want to observe carefully what your changes do to the database. If you treat the test database carelessly, it will change so much it becomes very different from the production database and you can't really tell from the test database what your SQL statements would do to the real database.

When working from an SQL client, before you commit an update, read what the database tells you about the result of your command. If you think too many rows have been matched, use a SELECT statement to view the result before you commit. Put the same restriction you had in the UPDATE statement into the SELECT statement to view what your updates look like.

Slide 10



DELETE

Order

ordID	ordDate	custID	shipDate
1111	2016-01-20	W123	2016-01-20
1112	2016-01-23	R234	2016-01-24
1113	2016-01-26	R341	2016-01-26

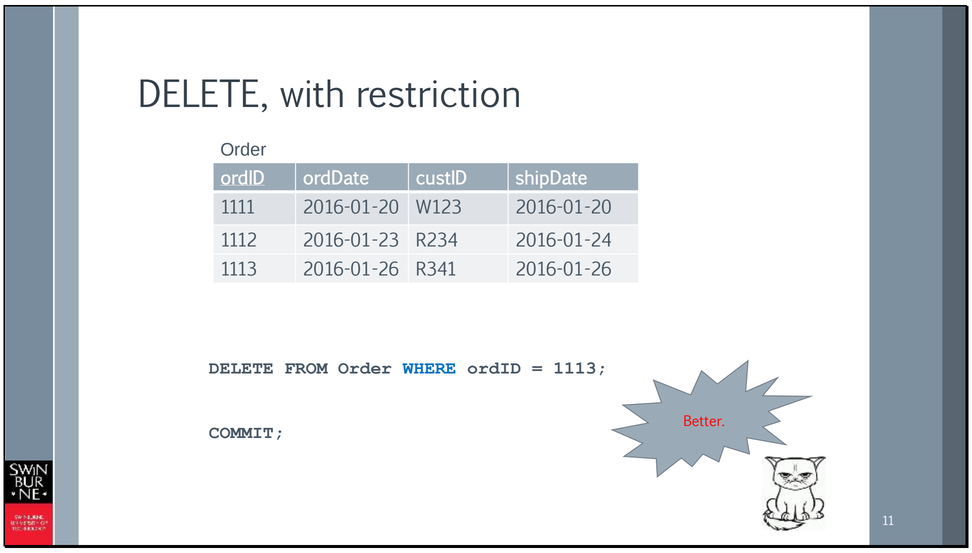
```
DELETE FROM Order;

COMMIT;
```

DELETE erases tuples from a table. If there is no restriction, no WHERE clause, every tuple goes and the table is empty.

However, do you think this statement would execute in a real database system? Order is the parent table for OrderItem. So if the foreign key constraint in OrderItem has been implemented, the DELETE would fail for those rows that have child rows in the OrderItem table. But you would still lose rows that don't have related rows in the OrderItem table.

Again, DELETE is not to be used in Autocommit mode – the outcome of DELETEs should be checked carefully before they are committed.

Slide 11



DELETE, with restriction

Order

ordID	ordDate	custID	shipDate
1111	2016-01-20	W123	2016-01-20
1112	2016-01-23	R234	2016-01-24
1113	2016-01-26	R341	2016-01-26

```
DELETE FROM Order WHERE ordID = 1113;

COMMIT;
```

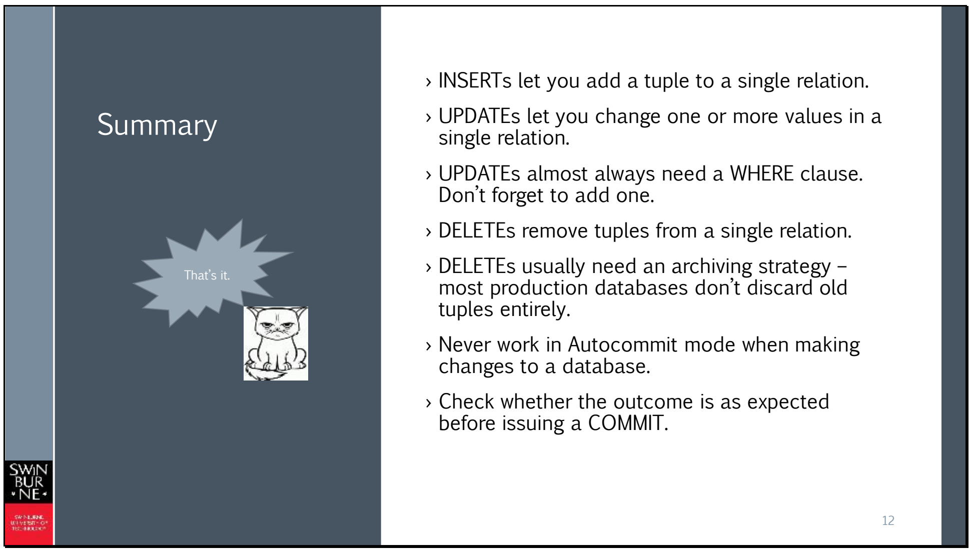
11

DELETE is a command that is used sparingly in practice. In many cases, rather than deleting rows that are not needed, production databases are often implemented with a column called deleted whose type is a timestamp. As long as the deleted column is either null or set to a date far in the future, the row is current. Once it is set to a date in the past, the row is considered deleted.

Eventually, rows have to be deleted when the table grows to an unmanageable size. Often, the rows are first copied into an archiving table before being deleted from the main table. This is mostly done by database administrators in batch processes, and not by user applications.

When you process DELETEs, the same applies as with updates: Don’t forget the restrictions and don’t work in autocommit mode.

Slide 12



Summary

› INSERTs let you add a tuple to a single relation.

› UPDATEs let you change one or more values in a single relation.

› UPDATEs almost always need a WHERE clause. Don't forget to add one.

› DELETEs remove tuples from a single relation.

› DELETEs usually need an archiving strategy – most production databases don't discard old tuples entirely.

› Never work in Autocommit mode when making changes to a database.

› Check whether the outcome is as expected before issuing a COMMIT.

12

Here is the summary of the most important points. You may want to read them before you go on to do the quiz.