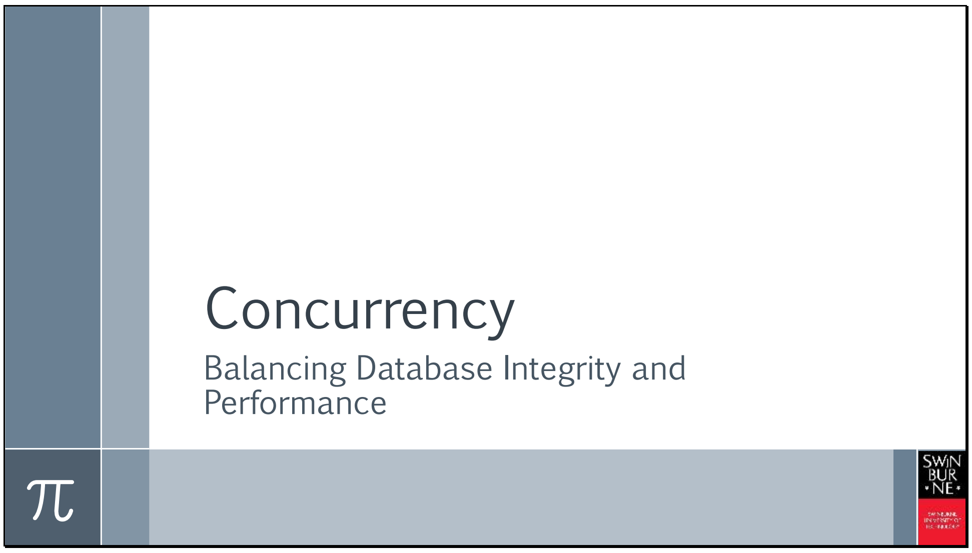
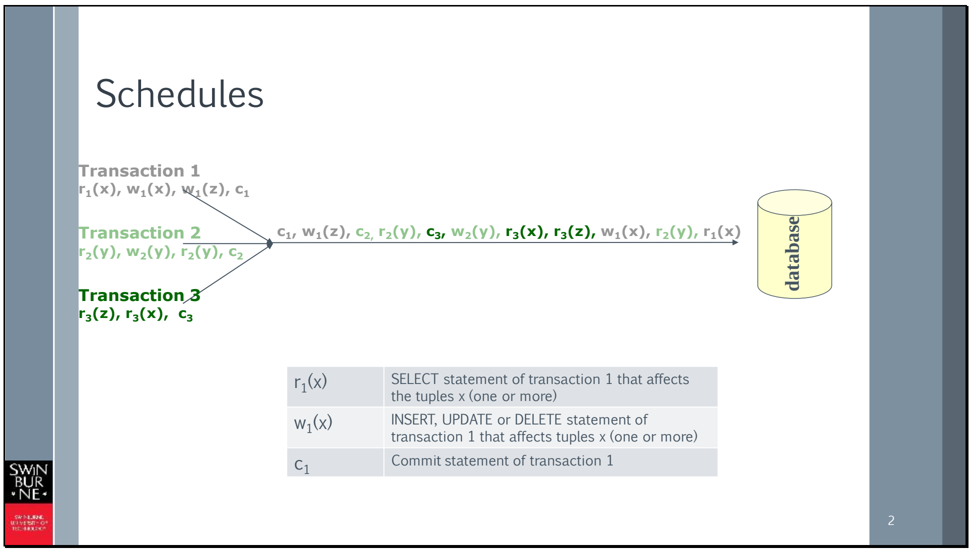


Concurrency

Balancing Database Integrity and Performance

This module explains more about why transactions are important and how the database management system can balance performance and isolation.

Slide 2



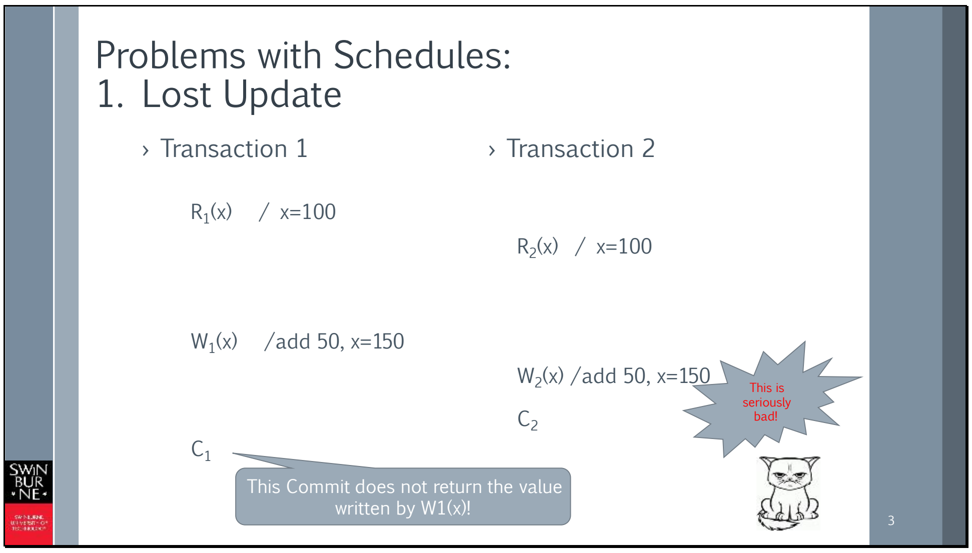
Schedules

Transaction 1
$r_1(x), w_1(x), w_1(z), c_1$
Transaction 2
$r_2(y), w_2(y), r_2(y), c_2$
Transaction 3
$r_3(z), r_3(x), c_3$
$c_1, w_1(z), c_2, r_2(y), c_3, w_2(y), r_3(x), r_3(z), w_1(x), r_2(y), r_1(x)$
database

$r_1(x)$	SELECT statement of transaction 1 that affects the tuples x (one or more)
$w_1(x)$	INSERT, UPDATE or DELETE statement of transaction 1 that affects tuples x (one or more)
c_1	Commit statement of transaction 1

Statements of different transactions arrive at the database, usually from many users over the network. In this notation, an r is a read statement, a w is a write statement and the subscript number defines the transaction it is part of. The letter in brackets is a variable that stands for the resource that is accessed. This is usually one or more tuples, also called rows.
The sequence of statements in the order in which they are executed are known as a schedule. You can see that the statements of different transactions alternate in the schedule – databases do not try to execute statements sequentially.
This is because many times, the data a statement needs is not yet in memory and the database management system has to fetch it from the disk. Disk operations are slow, and while we are waiting for the data, the processor can be used to process other statement.

Slide 3



Problems with Schedules:
1. Lost Update

› Transaction 1

› Transaction 2

$R_1(x)$ / x=100

$R_2(x)$ / x=100

$W_1(x)$ /add 50, x=150

$W_2(x)$ /add 50, x=150

C_2

C_1

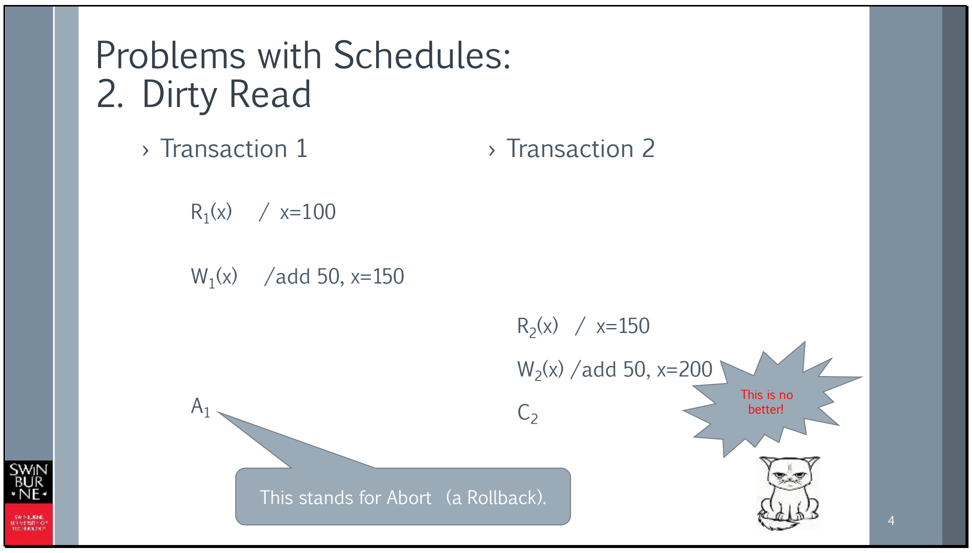
This Commit does not return the value written by W1(x)!

This is seriously bad!
3

In this example, we have two transactions, T1 and T2. T1 reads a value x in the database. Then T2 reads the same value. Then T1 updates the value based on the initial value it read. T2 does the same, using the same initial value. T2 commits first, but the fact that T1 commits last does not mean the value it wrote is reapplied.

Since both transactions read the same initial value, they both calculate the new value using the value that was valid before both transactions started. Since T2 applies its change last, the update of T1 is lost.
Whenever a transaction reads a value first and subsequently changes it, any other transaction that does the same can potentially overwrite the value. We say that these are conflicting statements because they belong to different transactions and access the same resource.

Slide 4



Problems with Schedules:
2. Dirty Read

› Transaction 1

› Transaction 2

$R_1(x)$ / x=100

$W_1(x)$ /add 50, x=150

$R_2(x)$ / x=150

$W_2(x)$ /add 50, x=200

A_1

C_2

This stands for Abort (a Rollback).

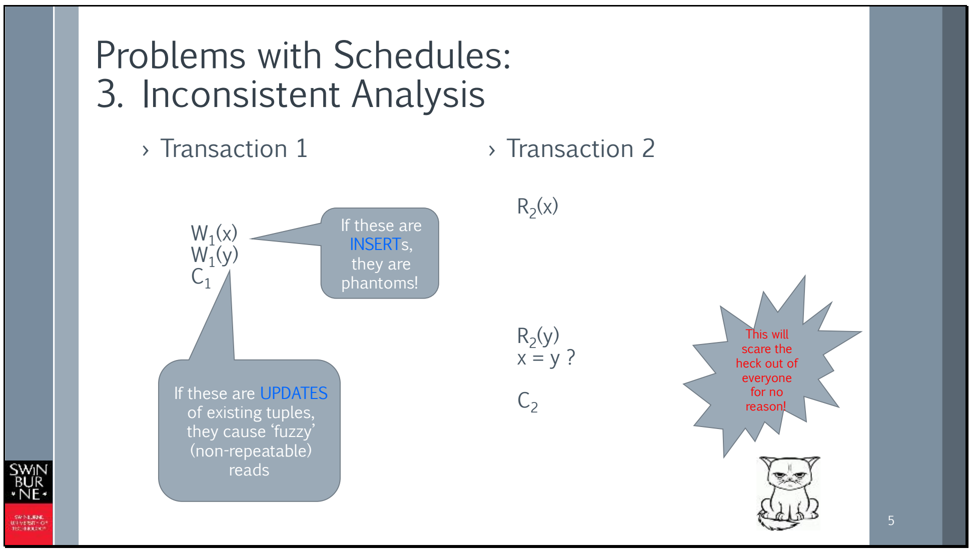
This is no better!

In this example T1 reads a value x in the database, then adds 50 to the value. T2 reads the value before it is committed. This is clearly better than the lost update we had on the previous slide.

Here, we only have a problem if T1 aborts and rolls back the changes. This would mean that the value read by T2 was only tentative and never really existed.

This is particularly serious when other write operations are based on the temporary value. When T2 commits, we have an incorrect value of x in the database. This is serious because it will be very hard to find out how this happened. Someone will have to read through the log file.

Slide 5



Problems with Schedules: 3. Inconsistent Analysis

› Transaction 1

› Transaction 2

$R_2(x)$

$W_1(x)$
$W_1(y)$
C_1

If these are INSERTs, they are phantoms!

$R_2(y)$
$x = y$?

C_2

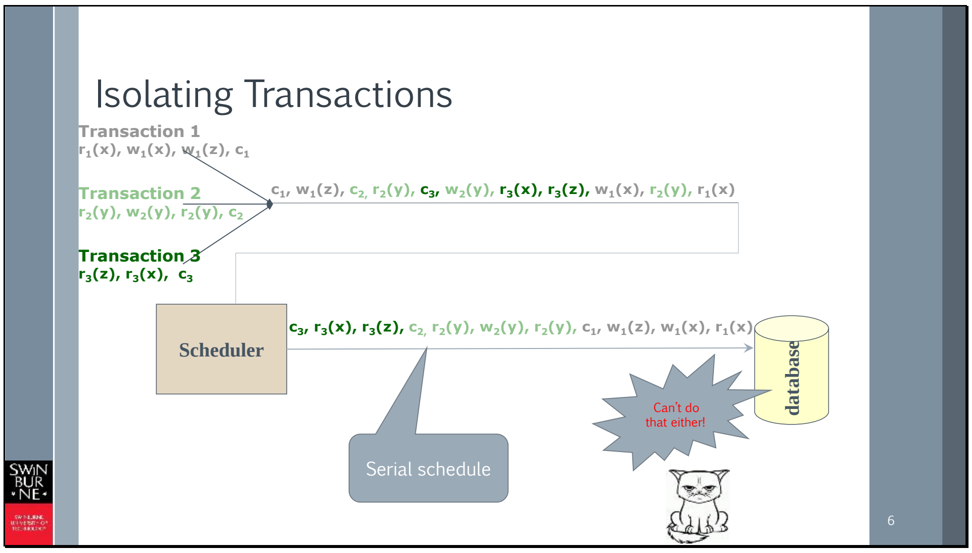
If these are UPDATES of existing tuples, they cause 'fuzzy' (non-repeatable) reads

This will scare the heck out of everyone for no reason!

5

In this example T2 reads two tuples x and y in order to compare them. T1 makes a change to both x and y before T2 can read y. When T2 finally reads y, the value has changed. If we expect x and y to match up in some way, for example if they are the sales and stock levels of a product, T2 sees a different value for x than for y, because they belong to two different states of the database (before and after T1).
This is not as serious as a lost update, because the database is still in a consistent state even though it appears not to be. Someone might just get scared that the stock level is too low compared to the sales, and so some products have disappeared in a black hole. But if people become seriously concerned, they are likely to repeat T2 and find that they have consistent values again..

Slide 6



Isolating Transactions

Transaction 1
$r_1(x), w_1(x), w_1(z), c_1$

Transaction 2
$r_2(y), w_2(y), r_2(y), c_2$

Transaction 3
$r_3(z), r_3(x), c_3$

$c_1, w_1(z), c_2, r_2(y), c_3, w_2(y), r_3(x), r_3(z), w_1(x), r_2(y), r_1(x)$

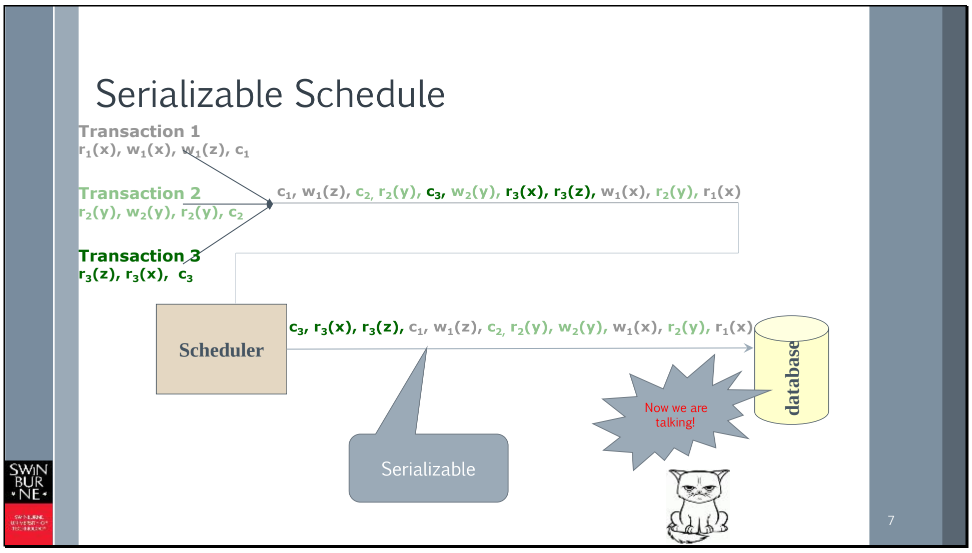
$c_3, r_3(x), r_3(z), c_2, r_2(y), w_2(y), r_2(y), c_1, w_1(z), w_1(x), r_1(x)$

Database management systems have a scheduler to manage the final schedule of the incoming transaction statements. A scheduler can delay a statement, which means that the following statements of the same transaction are also delayed. If statements of transactions were delayed until all other statements have ended, only one transaction could be processed at a time and we would have a serial schedule.

Serial schedules are always correct schedules, because there is no interference from other transactions. If there are still mistakes in the database, the transactions were badly designed.

Database management systems never enforce serial schedules, except by accident. They are too slow.

Slide 7



Serializable Schedule

Transaction 1
$r_1(x)$, $w_1(x)$, $w_1(z)$, c_1

Transaction 2
$r_2(y)$, $w_2(y)$, $r_2(y)$, c_2

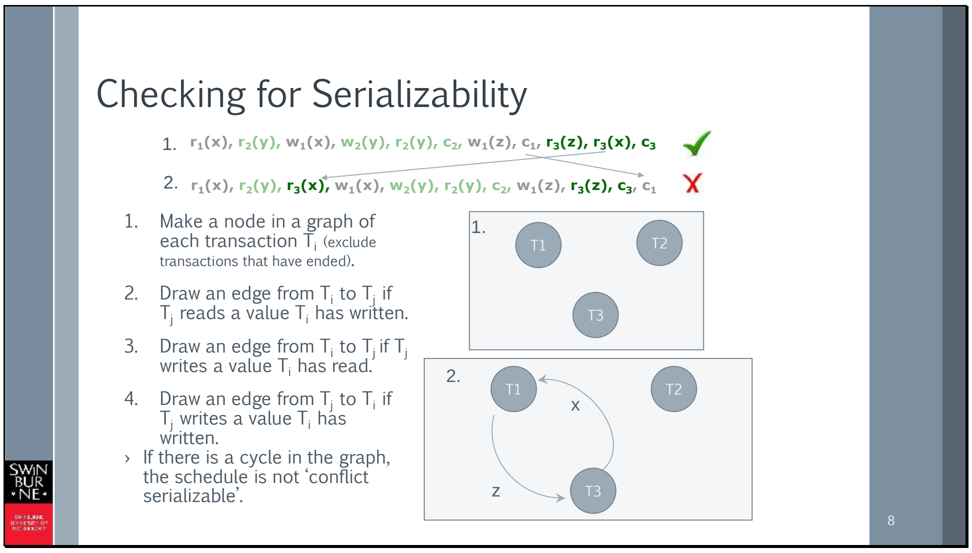
Transaction 3
$r_3(z)$, $r_3(x)$, c_3

c_1, $w_1(z)$, c_2, $r_2(y)$, c_3, $w_2(y)$, $r_3(x)$, $r_3(z)$, $w_1(x)$, $r_2(y)$, $r_1(x)$

c_3, $r_3(x)$, $r_3(z)$, c_1, $w_1(z)$, c_2, $r_2(y)$, $w_2(y)$, $w_1(x)$, $r_2(y)$, $r_1(x)$

Actually, the scheduler only has to identify conflicting operations – operations of different transactions that affect the same resource – and take care these do not interleave. All the other statements can be processed in any order, if they concern different resources.
If these rules are observed, the schedule has the same effect as a serial one, with much better performance. A schedule which is correct and produces the same result in the database as a serial one, while not executing the transactions serially one after the other, is called a serializable schedule.
In the example, if the statements arrive as in the line above (read right to left), T2 operations can be scheduled as they arrive because T2 does not conflict with the other transactions – it works with its own resource y. T3 has to wait until T1 completes because it reads the same resources as T1 but in different order, which could lead to a deadlock situation.

Slide 8

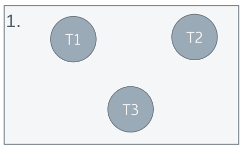


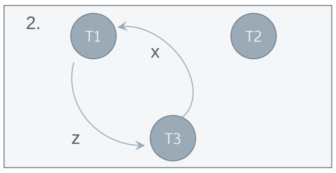
Checking for Serializability

1. $r_1(x), r_2(y), w_1(x), w_2(y), r_2(y), c_2, w_1(z), c_1, r_3(z), r_3(x), c_3$ ✓
2. $r_1(x), r_2(y), r_3(x), w_1(x), w_2(y), r_2(y), c_2, w_1(z), r_3(z), c_3, c_1$ ✗

1. Make a node in a graph of each transaction T_i (exclude transactions that have ended).
2. Draw an edge from T_i to T_j if T_j reads a value T_i has written.
3. Draw an edge from T_i to T_j if T_j writes a value T_i has read.
4. Draw an edge from T_j to T_i if T_j writes a value T_i has written.

› If there is a cycle in the graph, the schedule is not 'conflict serializable'.





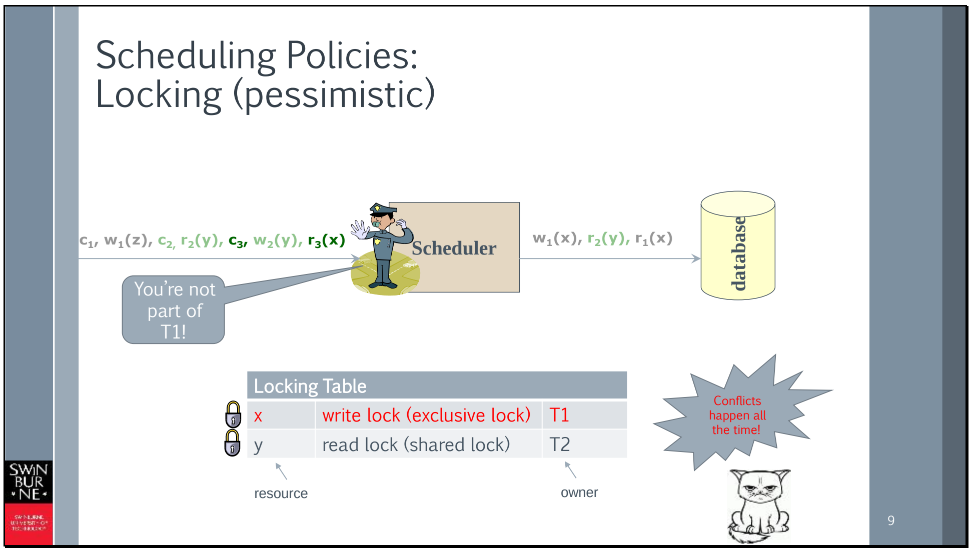
8

There are different definitions of serializability. Strictly speaking we are discussing conflict serializability, where transactions are serialised only when they access the same data. When two SQL statements access the same data, we say they are conflicting operations. When no two transactions attempt to access the same resource, the schedule is automatically conflict serializable.
By resource we mean the same piece of data – generally the same rows or the same attribute in a row.
When transactions attempt to access the same resource, we can use a dependency graph to determine whether the schedule is serializable. A dependency graph paints a node for each transaction. Whenever one transaction reads or writes the same resource as another transaction, we draw an arrow between them. If we end up with a circle between two transactions, we have a schedule that is not conflict serializable. Whether this is a real problem depends on the case.
The examples are the same transactions as in the previous slide, but now in more readable form left to right.
The first example is serializable – it has no edges in the dependency graph.

The second example is a variation of the first one, where the read operation of T3 and the commit of T1 are in a different location.

In the first example, the commit of T1 happens immediately after the write operation of z, and this concludes the transaction. This means transaction T1 and T3 are not concurrent and do not have to be considered for potential conflicts. It is normal for databases resources to be changed by different transactions – as long as transactions do not interfere with each other.

Slide 9



Scheduling Policies: Locking (pessimistic)

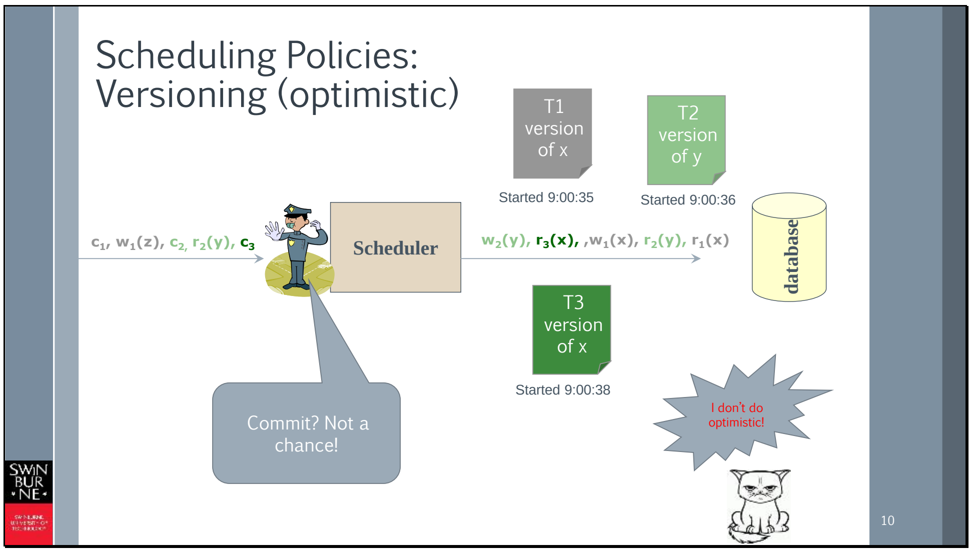
c_1, $w_1(z)$, c_2, $r_2(y)$, c_3, $w_2(y)$, $r_3(x)$
Scheduler
$w_1(x)$, $r_2(y)$, $r_1(x)$
database
You're not part of T1!
Conflicts happen all the time!
resource
owner

Locking Table		
x	write lock (exclusive lock)	T1
y	read lock (shared lock)	T2

Locking DBMSs have read and write locks. Read locks are also called shared locks because they can be shared across transactions. Everyone can read at the same time. Exclusive locks are for writing and can only be held by one transactions at a time. Read locks can be upgraded to write locks, but only if the upgrading transaction is the only one to hold the read lock.
This is a locking DBMSs best defence against lost updates: If a write operation is based on a read operation of the same resource, the transaction has to request an upgrade of the read lock to a write lock. If this is not possible, the scheduler can – depending on the isolation level – roll back the transaction.

This approach is called the pessimistic approach – thinking that there will be lots of conflicts and we have to prevent them from the start.

Slide 10

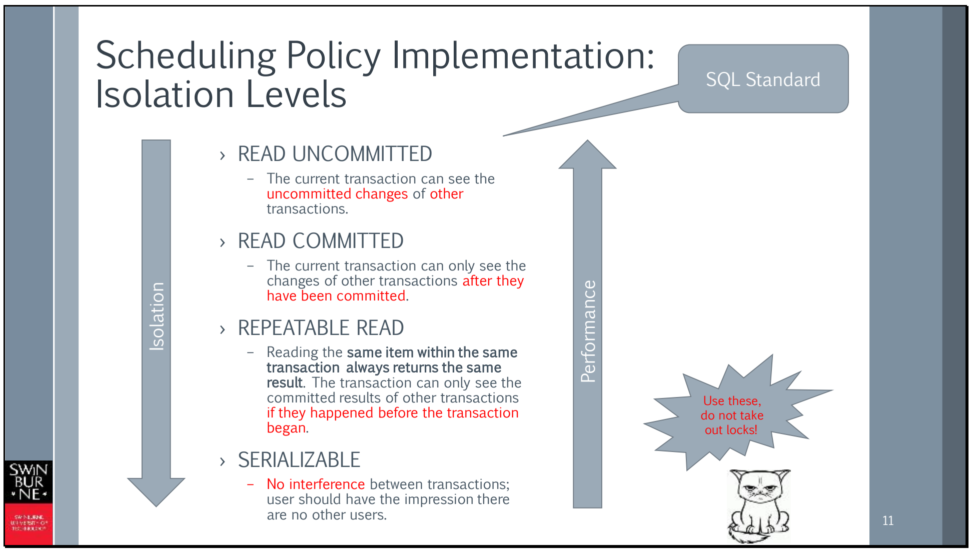


Scheduling Policies: Versioning (optimistic)

Versioning DBMSs let all operations make changes to the copies of the data held by their transactions and store them in separate versions tagged with a timestamp. When the transaction issues a commit, it checks whether there are conflicting versions and rolls back one of the transactions. DBMSs have different strategies how to decide which of the transactions to roll back. In the simplest approach, it chooses the transaction that has the later timestamp, so it is the younger one.
In the example, T1 started work on resource x earlier, but T3 wants to commit first – the DBMS can assume there is a potential for data corruption.
There are currently no purely versioning DBMSs on the market. Increasingly, DBMSs implement a mix of locking and multiversioning. Oracle has always applied this principle, SQL server followed suit in the last decade. MySQL has introduced multiversioning since it was taken over by Oracle. But as with Oracle, the approach is a mix of versioning and locking. No commercial RDBMS works purely with versioning at this time.

This is defined as an optimistic approach, because every transaction can make changes without checking for conflicts. The assumption is that conflicts don't happen very often. But the DBMS has to check later whether a conflict actually happened to prevent it from being made permanent.
Versioning is considered a faster approach than locking. Locking has to keep track of who has a lock for what, so a lot of updating of locking tables is going on behind the scenes.

Slide 11



Scheduling Policy Implementation: Isolation Levels

› READ UNCOMMITTED
 – The current transaction can see the uncommitted changes of other transactions.

› READ COMMITTED
 – The current transaction can only see the changes of other transactions after they have been committed.

› REPEATABLE READ
 – Reading the **same item within the same transaction always returns the same result**. The transaction can only see the committed results of other transactions if they happened before the transaction began.

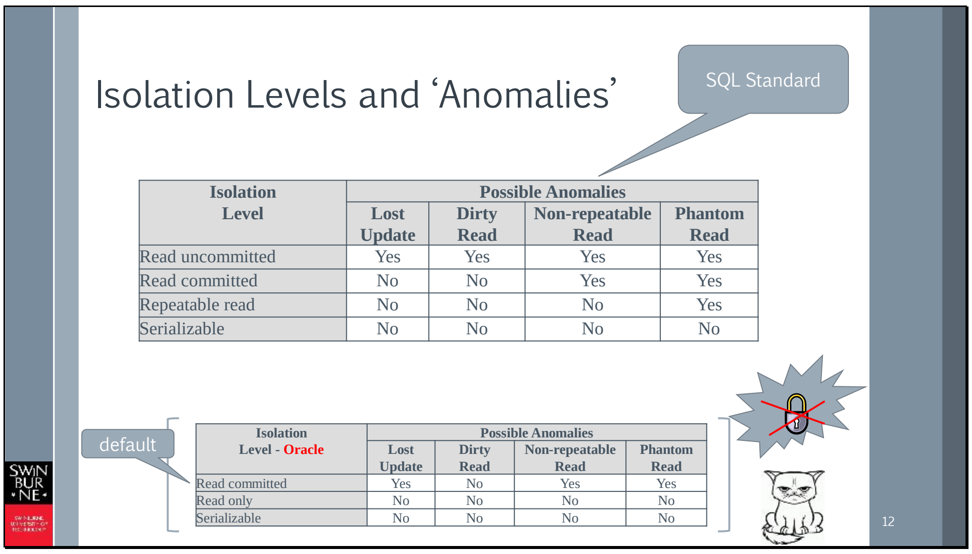
› SERIALIZABLE
 – No interference between transactions; user should have the impression there are no other users.

Most current DBMSs use a mix of optimistic and pessimistic approaches. When you program advanced features, such as reservations for a resource, say, a room booking system, you have to consider concurrency. According to the standard, the isolation levels listed should work the way they are described:
Read uncommitted is rather dangerous, because you can read a value that may never be committed. We have already seen that making updates based on such values can lead to inconsistencies.
Read committed is safer, and only lets other parties see the effects of a transaction when it has been committed. This means, though, that a value may change during a transaction, because other transactions can see the value immediately after it has been committed. Inconsistent reads are therefor possible. We have a potential inconsistent analysis problem.
Repeatable read differs from Serializable only very slightly. In RR isolation you can only ever see the same value for a given item. But new items may pop up when INSERTs are made in the middle of another transaction's reads. New rows that pop up in the middle of a transaction are called phantoms.

Naturally, the higher the isolation, the lower the performance. Even though we know full well that there are potential problems, DBMSs have default isolation levels that are generally lower than serializable. This is why this topic is important for programmers. Inexperienced programmers like to take out explicit locks on resources such as tables. This is very bad for performance and bypasses the mechanisms the DBMS offers to solve the problem.

Another problem is that the SQL standard is never fully respected by individual DBMSs. You will find great differences between the implementations. Whenever you identify a potential problem with the transactions you design, you have to check which isolation level is appropriate for this situation given your database product.

Slide 12



Isolation Levels and 'Anomalies'

SQL Standard

Isolation Level	Possible Anomalies			
	Lost Update	Dirty Read	Non-repeatable Read	Phantom Read
Read uncommitted	Yes	Yes	Yes	Yes
Read committed	No	No	Yes	Yes
Repeatable read	No	No	No	Yes
Serializable	No	No	No	No

default

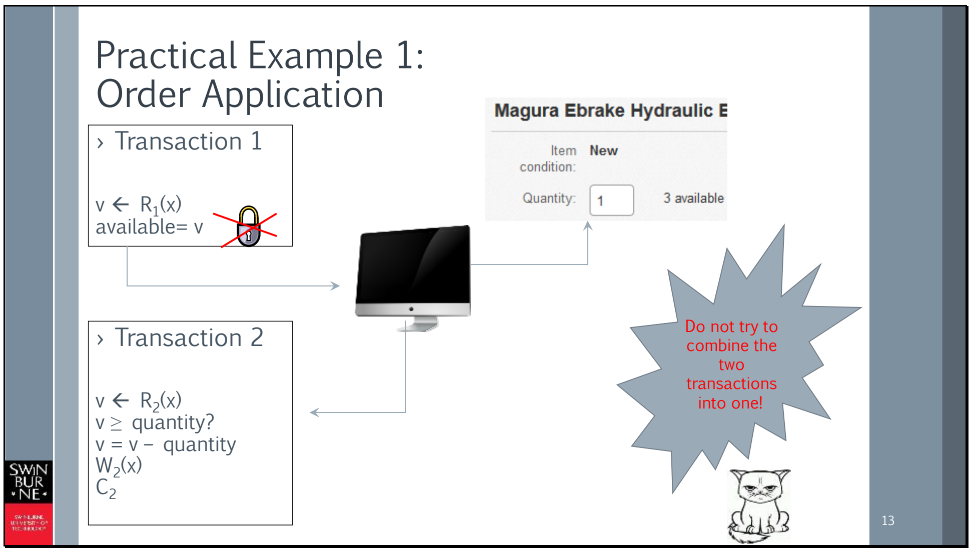
Isolation Level - Oracle	Possible Anomalies			
	Lost Update	Dirty Read	Non-repeatable Read	Phantom Read
Read committed	Yes	No	Yes	Yes
Read only	No	No	No	No
Serializable	No	No	No	No

12

As discussed earlier, serializable schedules are correct schedules and therefore safe in terms of database consistency. However, the SQL standard as well as DBMS manufacturers have realised that depending on the transaction, some interference can be allowed. Different isolation levels have been provided to give the programmer a chance to weigh performance against isolation. Interferences between transactions are known as 'anomalies' in schedules because in correct schedules, they do not happen. Whether an anomaly is actually a problem is for you to decide when you create your transactions.
The cumbersome news is that every DBMS provides slightly different implementations of isolation levels. Oracle is an extreme example: It only implements two isolation levels included in the standard. The third is proprietary. Even read committed does not work as promised by the standard: It allows lost updates! To make things worse, Oracle do not recommend you to switch to Serializable when you have to ensure lost updates do not happen. They recommend a workaround instead.

Most RDBMSs give you the possibility of taking out locks on resources such as tuples or relations. This practice usually indicates that the programmer does not understand the DBMS they are working with and its isolation levels. To work with DBMSs professionally, you have to study their isolation mechanisms.

Slide 13



Practical Example 1: Order Application

› Transaction 1

$v \leftarrow R_1(x)$
available= v

› Transaction 2

$v \leftarrow R_2(x)$
$v \geq$ quantity?
$v = v -$ quantity
$W_2(x)$
C_2

Magura Ebrake Hydraulic E

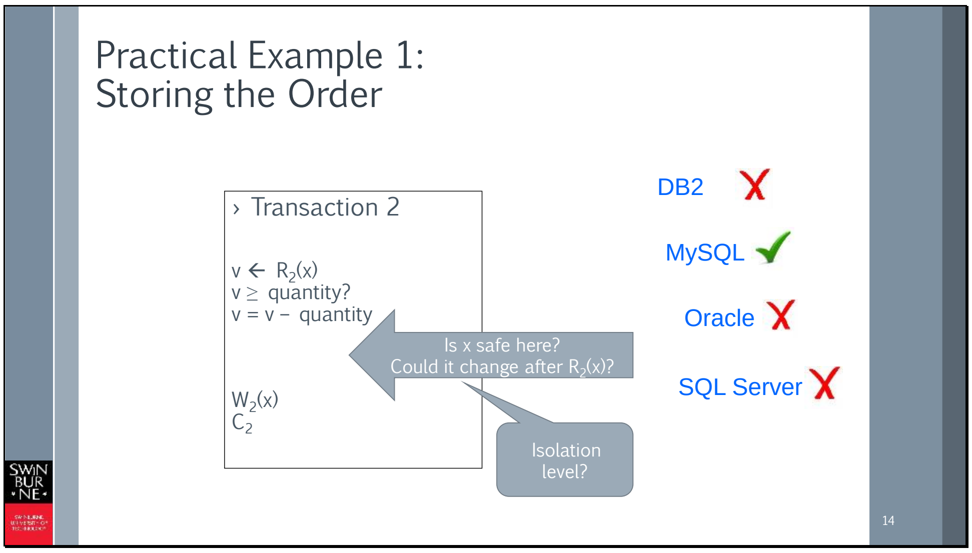
Item condition: New

Quantity: 1 3 available

Do not try to combine the two transactions into one!
13

Many applications on the Web let you buy things. When you have browsed to the item you want, you often get to choose a quantity up to the available maximum. How does the application know how many items are available?
Naturally, the available quantity has been looked up in the database.
While you are making up your mind and checking out, many other users may have ordered the same product. We cannot assume that the quantity is still the same as it was at the time of Transaction 1.
The application has to check again when it processes the order, which it does in Transaction 2.
Some people think 'locking' the quantity until the user has decided is a good idea. This will not work, because as we remember, the locks belong to the transaction, which ends here.
You may think we can combine T1 and T2 into a very long transaction to make the lock last. This is very bad – it assumes no one else can even browse to this item while a single user is looking at it! If the potential buyer decides to take a coffee break, this lock can remain for 15 minutes!

Slide 14



Practical Example 1: Storing the Order

› Transaction 2

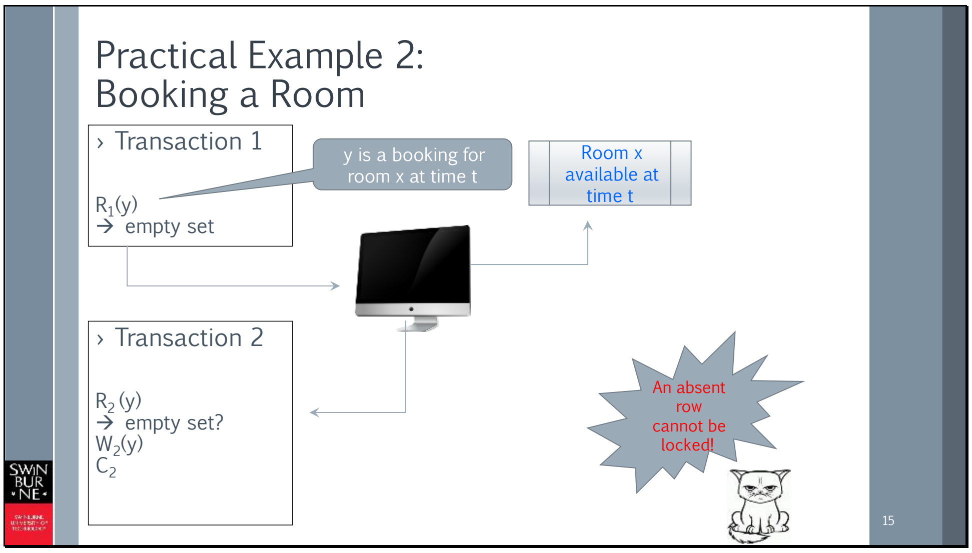
$v \leftarrow R_2(x)$
$v \geq$ quantity?
$v = v -$ quantity

$W_2(x)$
C_2

Is x safe here?
Could it change after $R_2(x)$?
Isolation level?
DB2 ✗
MySQL ✓
Oracle ✗
SQL Server ✗
14

Because we don’t know whether the available quantity has changed while the buyer was making up his mind and finally checking out, we have to re-check the available quantity to ensure the order can go ahead.
Naturally, the re-checking of the availability and the eventual update of the quantity have to happen in the same transaction, because we want to make sure that the quantity cannot change between the read and the update of the quanity.
The question is, will the database management system ensure that x does not change between the Read and the Write operations?
This depends on which DBMS we are using and its current isolation level.
Different DBMSs have different default isolation levels. In many cases, the default isolation level does not guarantee that other transactions cannot change resource x in the middle of this transaction 2. MySQL defaults to Repeatable Read, and is therefore the only one among the four DBMSs that handles this problem automatically.

Slide 15



Practical Example 2: Booking a Room

› Transaction 1

$R_1(y)$
→ empty set

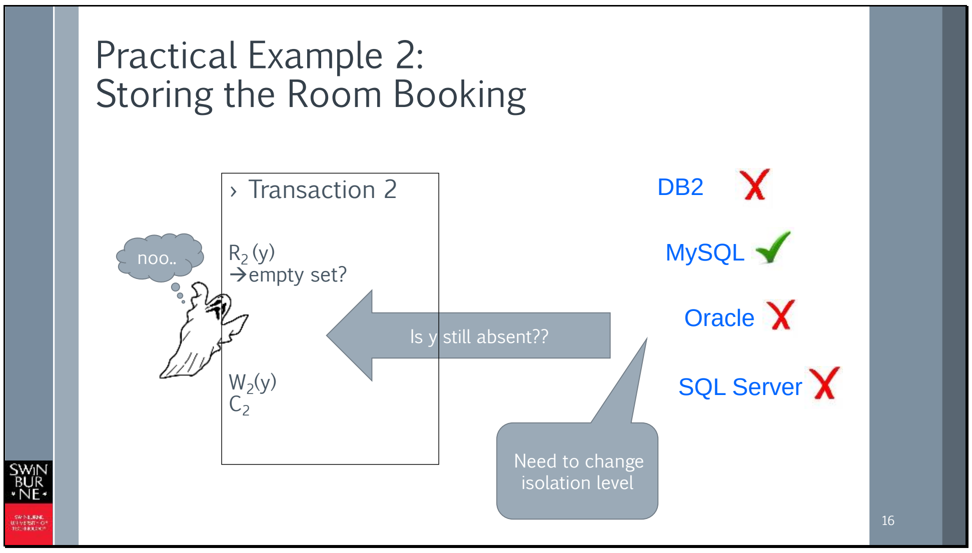
› Transaction 2

$R_2(y)$
→ empty set?
$W_2(y)$
C_2

y is a booking for room x at time t
Room x available at time t
An absent row cannot be locked!
15

Many booking applications have to establish the nonexistence of a reservation before offering the resource to a user. In this case, let's assume we are booking a room x at time t. We have established in T1 that no booking y exists for room x at that time, and the user books it.
In a second transaction T2 we would like to store the booking. As in example 1, we first have to re-check whether anyone has made a booking in the meantime. This problem is more complex than the previous one, because DBMSs have the habit of locking things that shouldn't be accessible to other transactions. It is hard – but not impossible – to lock things that are absent.
This is the reason why according to the standard, phantoms only have to be prevented at the SERIALIZABLE isolation level.

Slide 16



Practical Example 2: Storing the Room Booking

› Transaction 2

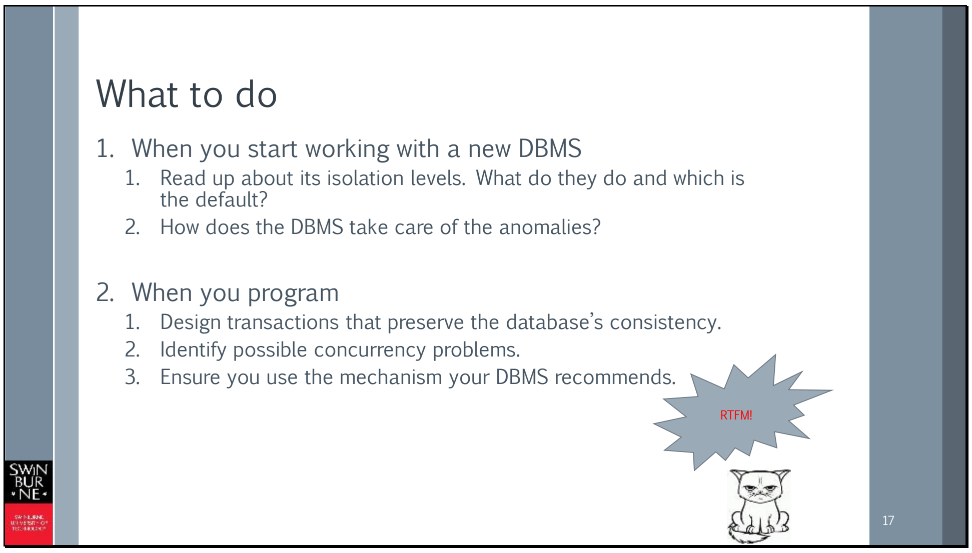
R_2 (y)
→empty set?

W_2(y)
C_2

noo..
Is y still absent??
Need to change isolation level
DB2 ✗
MySQL ✓
Oracle ✗
SQL Server ✗
16

When we read y again in T2 before committing, again we have to consider whether y can possibly change between the read operation and the write operation, in which we add the booking. In this case, a new row could appear in the middle of this T2, known as a phantom. If you remember the isolation levels and the anomalies they allow, you will conclude that only Serializable can save you from phantoms.
Unfortunately, life is slightly more complicated than this. You have to check whether your DBMS suggests to solve phantoms using the serializable isolation level. A web search for your database product and phantoms will tell you how professionals suggest you should solve this problem. The important thing is that you recognise the phantom problem in the first place.

Slide 17



What to do

1. When you start working with a new DBMS
   1. Read up about its isolation levels. What do they do and which is the default?
   2. How does the DBMS take care of the anomalies?

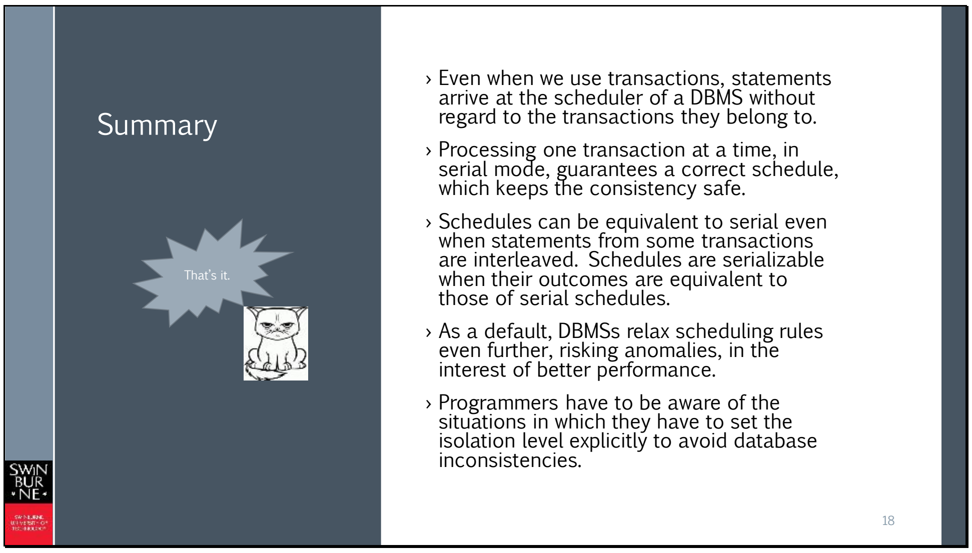
2. When you program
   1. Design transactions that preserve the database's consistency.
   2. Identify possible concurrency problems.
   3. Ensure you use the mechanism your DBMS recommends.

Fortunately for the programmer, there are only three main situations that require careful consideration.
The first is when an existing value, such as the availability, changes in the middle of the transaction, if your transaction depends on its value.
The second is when we are trying to book a resource and our actions depend on the absence of a previous booking.
The third is doing an analysis and having conflicting information.

Depending on the company you are working for and its business rules, having such conflicts may or may not be a problem. You just have to check.

When you start working with a new DBMS, get familiar with its behaviour at different isolation levels. You can also google for you database product and an anomaly, such as lost update. There will be lots of advice out there telling you how to deal with this in any given environment. The most important thing is that when you program, you know how to use transactions and understand how to design them and identify potential concurrency problems.

Slide 18



Summary

› Even when we use transactions, statements arrive at the scheduler of a DBMS without regard to the transactions they belong to.

› Processing one transaction at a time, in serial mode, guarantees a correct schedule, which keeps the consistency safe.

› Schedules can be equivalent to serial even when statements from some transactions are interleaved. Schedules are serializable when their outcomes are equivalent to those of serial schedules.

› As a default, DBMSs relax scheduling rules even further, risking anomalies, in the interest of better performance.

› Programmers have to be aware of the situations in which they have to set the isolation level explicitly to avoid database inconsistencies.

18

Here are the most important points discussed in this module. You may want to stop the recording to have a read through them. When you are ready, start the quiz about this module.