



Data concurrency and locking

IBM Information Management Cloud Computing Center of Competence
IBM Canada Labs

Agenda

- Transactions
- Concurrency & Locking
- Lock Wait
- Deadlocks

Supporting reading material & videos

- **Reading materials**

- Getting started with DB2 Express-C eBook
 - Chapter 13: Concurrency and locking

- **Videos**

- db2university.com course AA001EN
 - Lesson 8: Data concurrency and locking

05/26/2011

3

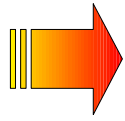
Template Documentation

© 2011 IBM Corporation

If you need more details about this topic, refer to this supporting material:

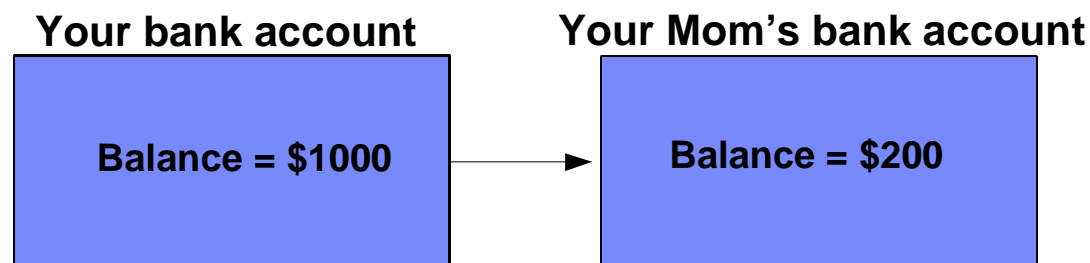
- Chapter 13: Concurrency and locking of the book “Getting started with DB2 Express-C 3rd Edition” and
- Lesson 8 of the db2university.com course AA001EN DB2 Academic Training

Agenda



- Transactions
- Concurrency & Locking
- Lock Wait
- Deadlocks

What is a transaction?



Transfer \$100 from your account to your Mom's account:

- Debit \$100 from your bank account (Subtract \$100)
- Credit \$100 to your mom's bank account (Add \$100)

05/26/2011

Template Documentation

5

© 2011 IBM Corporation

A transaction could be better explained using this simple example: Let's say, you have access to two accounts at a bank; one is your bank account, and the other one is your mom's bank account. In your bank account, there is a balance of one thousand dollars. In your mom's account, there is a balance of two hundred dollars. Now, let's say you want to transfer one hundred dollars from your account to your mom's account. Behind the scenes 100 hundred dollars will first be debited from your account and then 100 dollars will be credited to your mom's account. However, what would happen if the 100 dollars from your account is debited and all of a sudden there is a power outage and the operation is not completed?. You just lost 100 hundred dollars!. As you can tell, such scenario cannot be allowed to happen in any real life application. To solve this problem, both operations, the debit, and the credit should be treated as one single unit; as a transaction.

What is a transaction? (cont'd)

- One or more SQL statements altogether treated as one single unit
- Also known as a Unit of Work (UOW)
- A transaction starts with any SQL statement and ends with a COMMIT or ROLLBACK
- COMMIT statement makes changes permanent to the database
- ROLLBACK statement reverses changes
- COMMIT and ROLLBACK statements release all locks

05/26/2011

Template Documentation

6

© 2011 IBM Corporation

So, a transaction is basically treating one or more SQL statements as one unit. If one statement of a transaction fails, the entire transaction fails. A transaction is also known as Unit of work (UOW). As we will see in the next slide, a transaction starts with any SQL statement and ends with a COMMIT or ROLLBACK statement. A COMMIT statement makes changes permanent to the database while a ROLLBACK statement reverses changes. Both statements release all locks taken on rows.

Example of transactions

```
INSERT INTO employee VALUES (100, 'JOHN')
INSERT INTO employee VALUES (200, 'MANDY')
COMMIT
```

First SQL statement starts transaction

empID	name
100	JOHN
200	MANDY

No changes applied due to ROLLBACK

```
DELETE FROM employee WHERE name='MANDY'
UPDATE employee SET empID=101 where name='JOHN'
ROLLBACK
```

empID	name
100	JOHN
200	MANDY

```
UPDATE employee SET name='JACK' where empID=100
COMMIT
```

There is nothing to rollback

empID	name
100	JACK
200	MANDY

```
ROLLBACK
05/26/2011
```

Template Documentation

© 2011 IBM Corporation

For example in this slide you can see several transactions. For the first transaction, two rows are inserted into the employee table. The transaction ends with the COMMIT statement which guarantees these two rows (for 'JOHN' and 'MANDY') are stored in the table. The next transaction starts with the DELETE statement and ends with a ROLLBACK. Since it ends with a ROLLBACK, any changes made are not actually applied, and the table remains exactly as it was before the transaction started. Then we have another transaction that updates the name to be 'JACK' for empID=100. Since this transaction ends with a COMMIT, the change is applied as highlighted in yellow in the slide. Finally, there is one last short transaction consisting of a ROLLBACK. This transaction of one statement would have no effect, since nothing can be rolled back at this time.

Transactions – ACID rules

- **A**tomicity

- All statements in the transaction are treated as a unit.
- If the transaction completes successfully, everything is committed
- If the transaction fails, everything done up to the point of failure is rolled back.

- **C**onsistency

- Any transaction will take the data from one consistent state to another, so only valid consistent data is stored in the database

- **I**solation

- Concurrent transactions cannot interfere with each other

- **D**urability

- Committed transactions have their changes persisted in the database

05/26/2011

Template Documentation

8

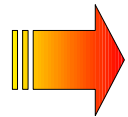
© 2011 IBM Corporation

ACID rules guarantee database transactions are processed in a reliable way.

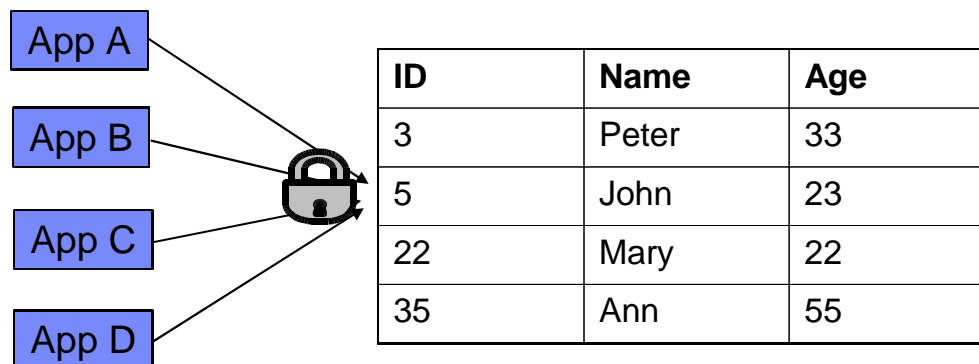
The “Isolation” rule is discussed in more detail in the next slides.

Consistency and Durability are also discussed in more detail in the presentation about “Backup and recovery”

Agenda

- Transactions
- • Concurrency & Locking
- Lock Wait
- Deadlocks

Concurrency and Locking



- Concurrency:
 - Multiple users accessing the same resources at the same time
- Locking:
 - Mechanism to ensure data integrity and consistency

05/26/2011

Template Documentation

10

© 2011 IBM Corporation

Let's move on, and provide an overview of concurrency and locking. Let's say there are several applications; application A, B, C and D trying to access the same row in the same table. Now if there was no concurrency control, then you may have that all of these applications could be doing an update on this same row at the same time. So at the end what will be the final result? It would be hard to tell. Concurrency, in general means that several applications can work at the same time on the same database, or same tables on the database while keeping the data consistent. Locking is how a database management system can ensure data integrity and consistency. So in this example, if application B is the first one to perform the update on the 2nd row, it would immediately take a lock on the row, and the other applications would have to wait until this lock is released.

Locking

- Locks are acquired automatically as needed to support a transaction based on “isolation levels”
- COMMIT and ROLLBACK statements release all locks
- Two basic types of locks:
 - Share locks (S locks) – acquired when an application wants to read and prevent others from updating the same row
 - Exclusive locks (X locks) – acquired when an application updates, inserts, or deletes a row

05/26/2011

Template Documentation

11

© 2011 IBM Corporation

Locks are normally acquired automatically for you based on isolation levels that you can set. Isolation levels are like 'policies' on how locks are taken, and will be discussed in more detail later on. As mentioned earlier, a COMMIT or ROLLBACK statement will release all locks on a row. There are several types of locks in DB2. For simplicity, we only show the main two: An 'S' lock for 'share' is taken when you perform READ operations such as when using a SELECT statement. An 'X' lock for 'exclusive' is taken when you perform operations such as a DELETE, UPDATE, or INSERT.

Problems if there is no concurrency control

- Lost update
- Uncommitted read
- Non-repeatable read
- Phantom read

05/26/2011

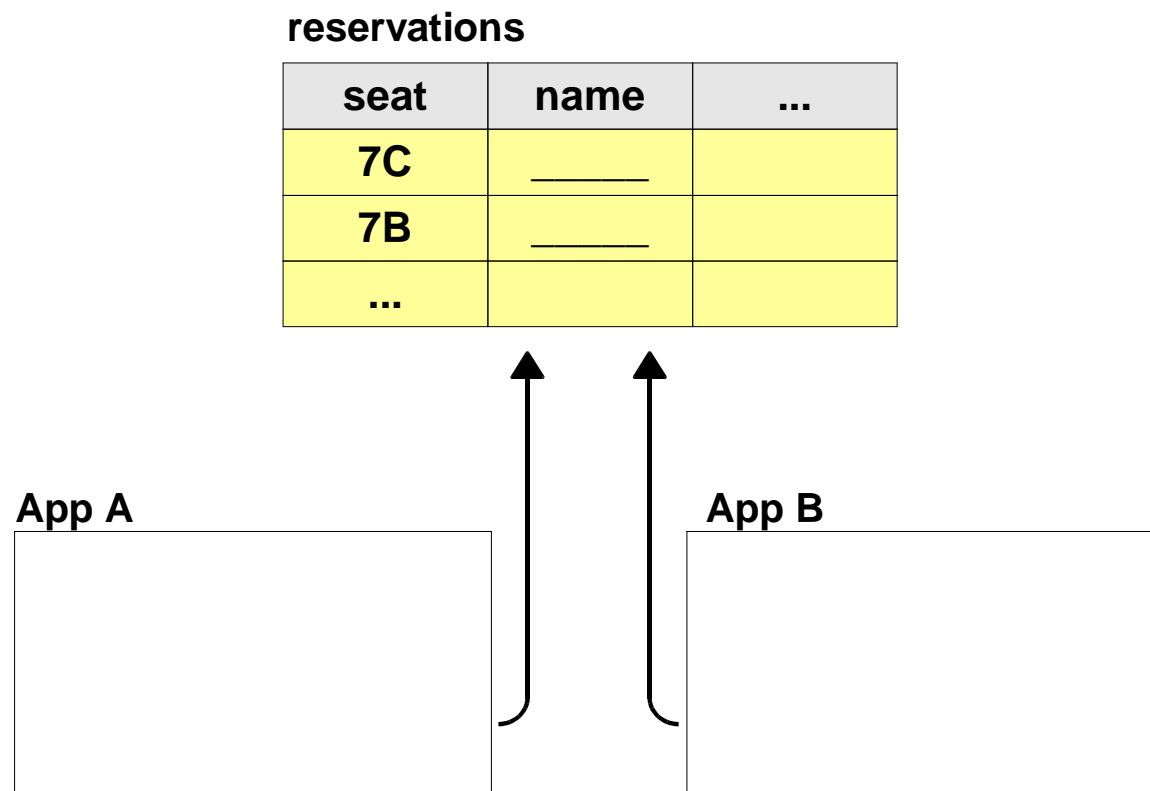
12

Template Documentation

© 2011 IBM Corporation

Concurrency allows users to access a database simultaneously, so that several applications can work on the same database at the same time while the data remains consistent. There are several problems that can happen if there is no concurrency control. The four problems that can happen are: lost update, uncommitted read, non-repeatable read, and phantom read. We explain each of these problems in more detail next.

Lost update



05/26/2011

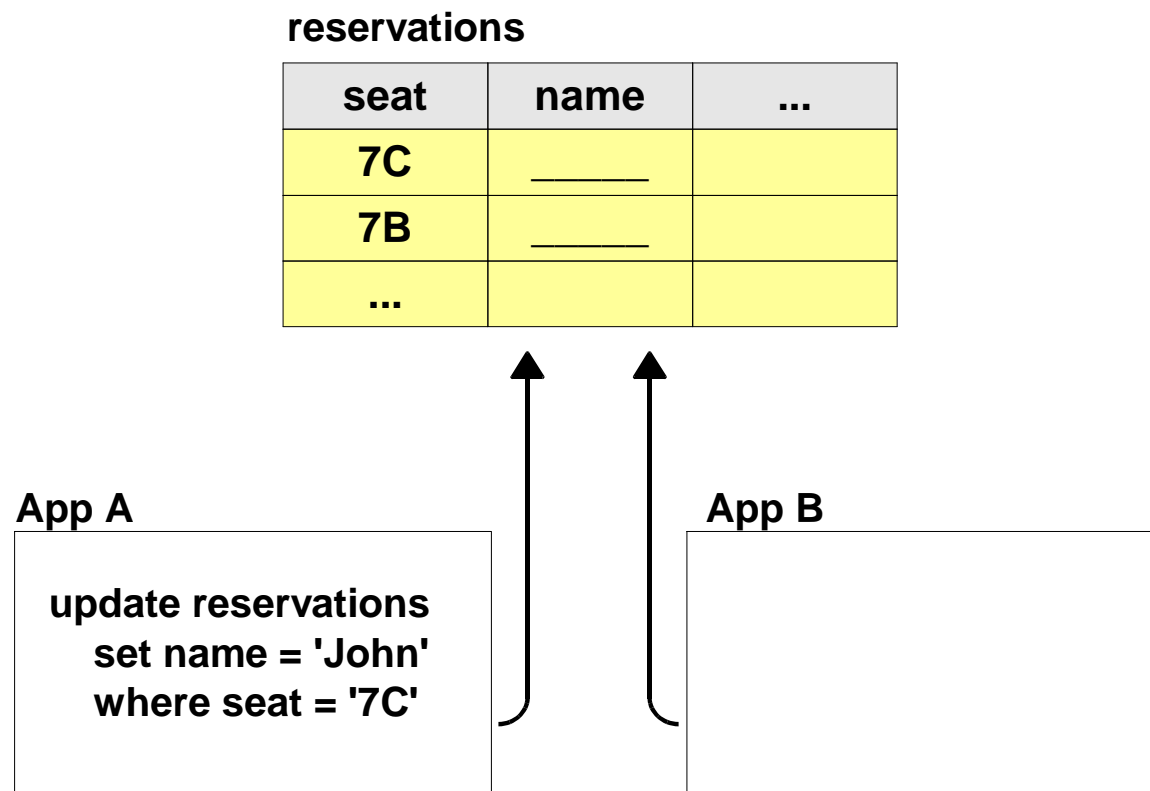
13

Template Documentation

© 2011 IBM Corporation

Let's use the following example to explain the “Lost update” problem. In this example, say you have a table called “reservations “ which has the column “seat”, the column “name”, and so on.

Lost update



05/26/2011

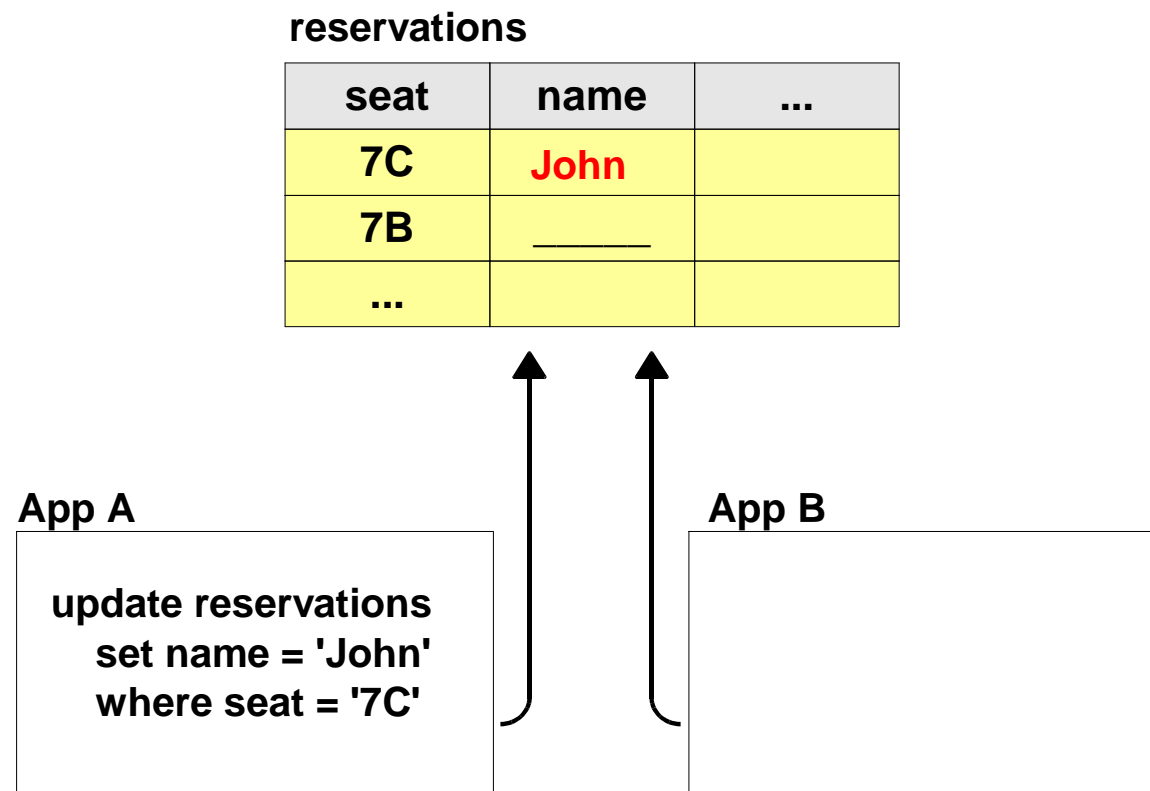
14

Template Documentation

© 2011 IBM Corporation

Now say application “A” updates the table “reservation”, so that the name for seat 7C is

Lost update



05/26/2011

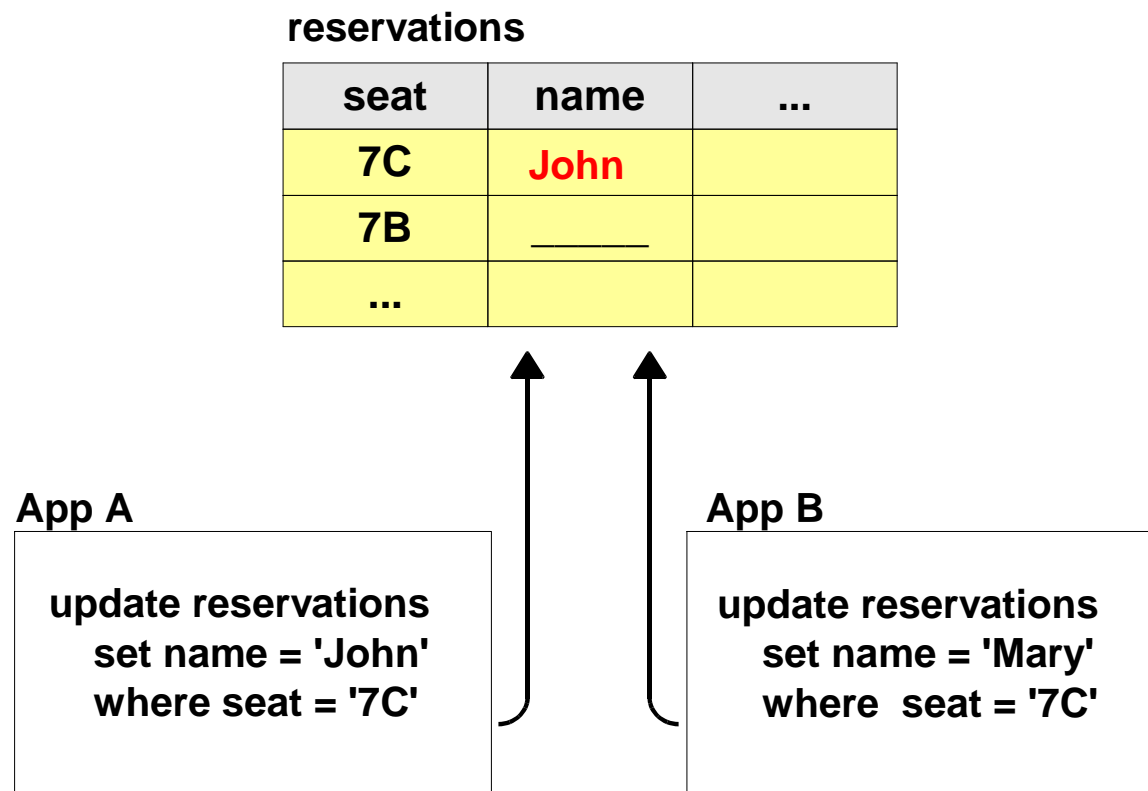
15

Template Documentation

© 2011 IBM Corporation

changed to “John”.

Lost update



05/26/2011

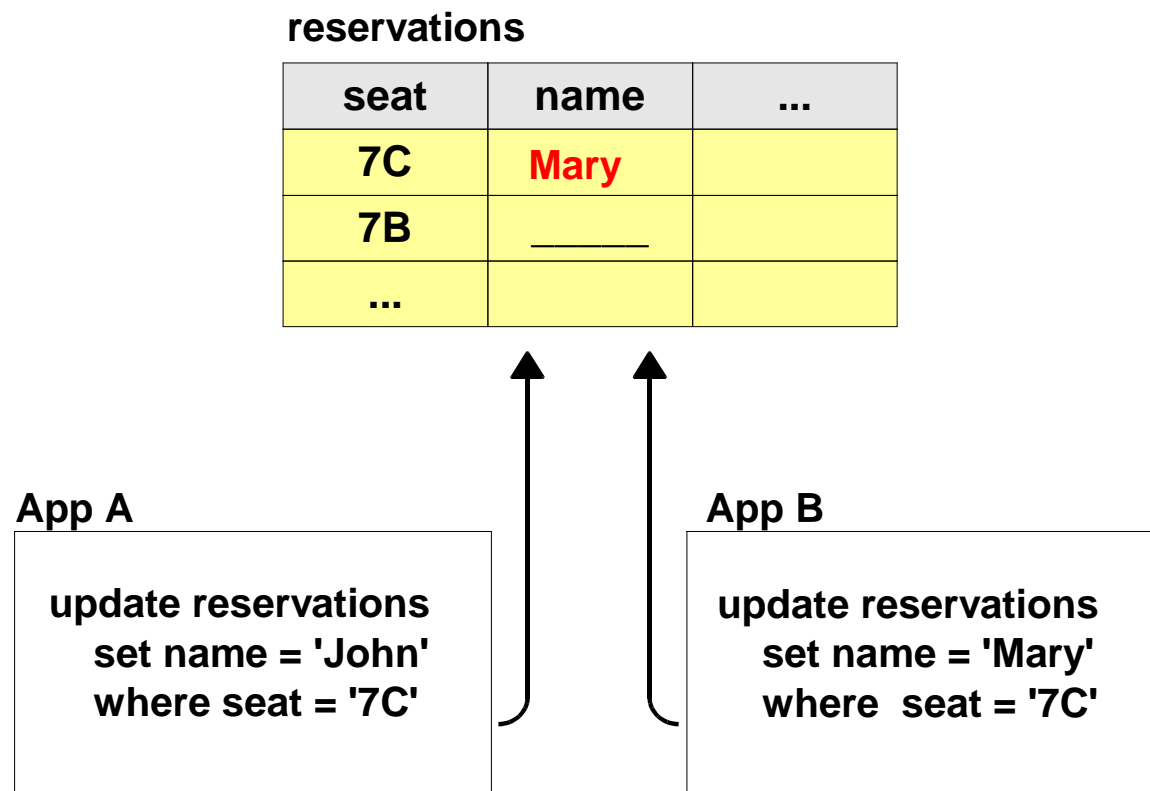
16

Template Documentation

© 2011 IBM Corporation

Now assume there is another application “B” at the same time is trying to do the same but

Lost update



05/26/2011

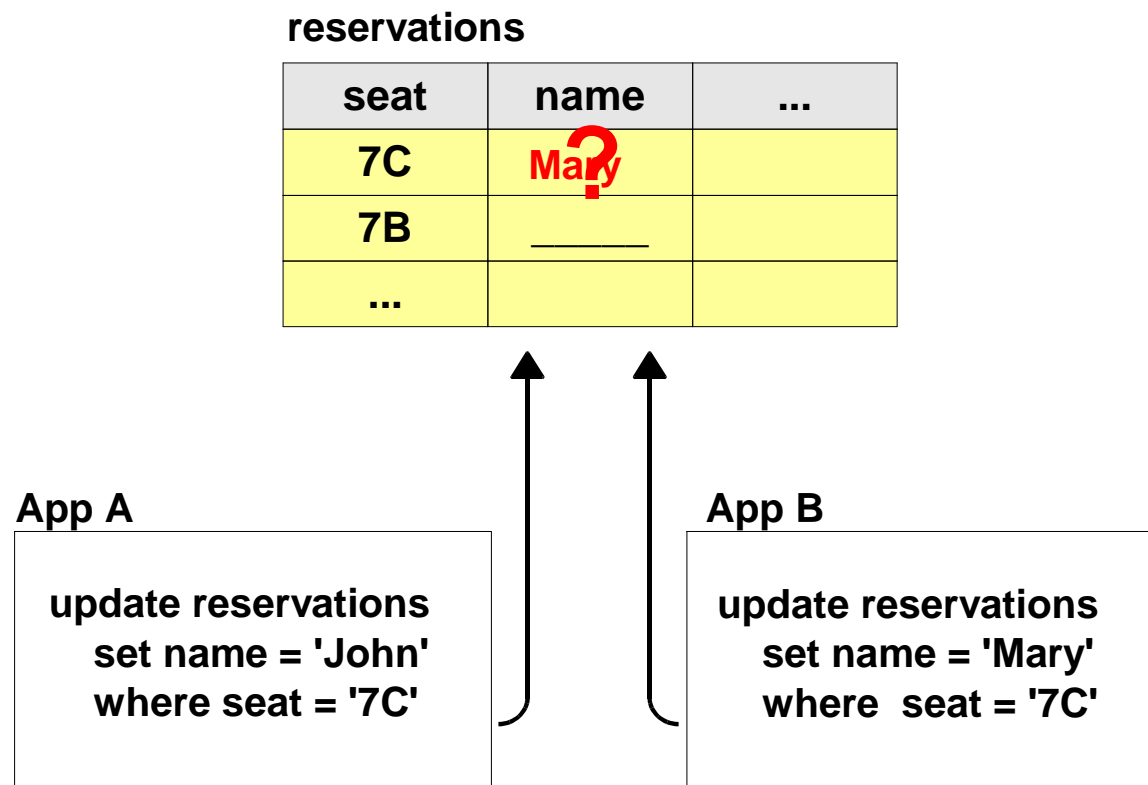
17

Template Documentation

© 2011 IBM Corporation

using the name “Mary”.

Lost update



05/26/2011

18

Template Documentation

© 2011 IBM Corporation

Assuming application B issued the update slightly later than application A, the last value for the name would be “Mary”, which means that the first update from application A was a “lost update”.

If we repeat the same process again, but this time application B performs the update first followed by application A, in that case the last value for the name would be “John”, and the first update from application B would be lost.

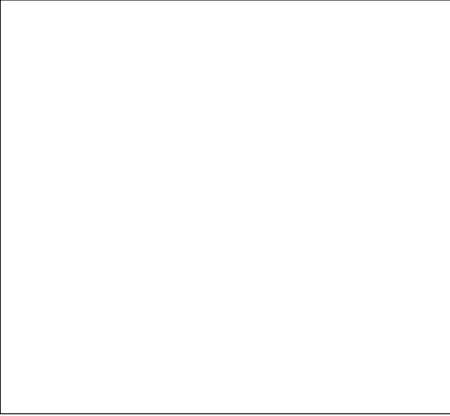
So basically without some type of control, we would lose an update from the first application.

Uncommitted read (also known as “dirty read”)

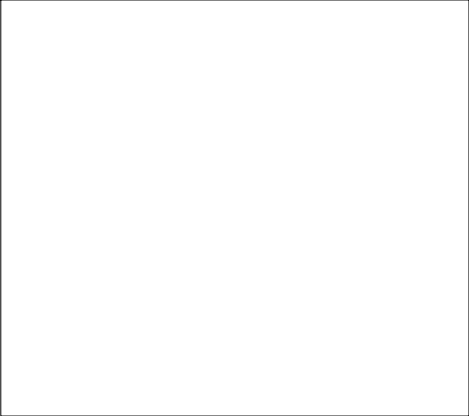
reservations

seat	name	...
7C	_____	
7B	_____	
...		

App A



App B



To explain the second problem called “Uncommitted read”, let's use the same example as before.

Uncommitted read (also known as “dirty read”)

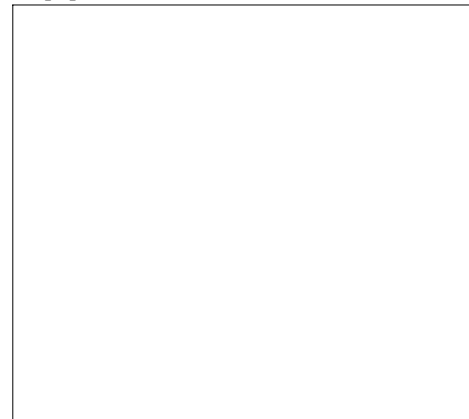
reservations

seat	name	...
7C	_____	
7B	_____	
...		

App A

**update reservations
set name = 'John'
where seat = '7C'**

App B



05/26/2011

20

Template Documentation

© 2011 IBM Corporation

Application A performs an update so that “John” is the name that corresponds to seat '7C'.

Uncommitted read (also known as “dirty read”)

reservations

seat	name	...
7C	John	
7B	_____	
...		

App A

update reservations
set name = 'John'
where seat = '7C'

App B



Uncommitted read (also known as “dirty read”)

reservations

seat	name	...
7C	John	
7B	_____	
...		

App A

**update reservations
set name = 'John'
where seat = '7C'**

App B

**Select name
from reservations
where seat is '7C'**

05/26/2011

22

Template Documentation

© 2011 IBM Corporation

Next, application B issues a SELECT which would retrieve the name of this same record,

Uncommitted read (also known as “dirty read”)

reservations

seat	name	...
7C	John	
7B	_____	
...		

App A

update reservations
set name = 'John'
where seat = '7C'

App B

Select name
from reservations
where seat is '7C'

John

so that output is “John”.

Uncommitted read (also known as “dirty read”)

reservations

seat	name	...
7C	John	
7B	_____	
...		

App A

update reservations
set name = 'John'
where seat = '7C'

Roll back

App B

Select name
from reservations
where seat is '7C'

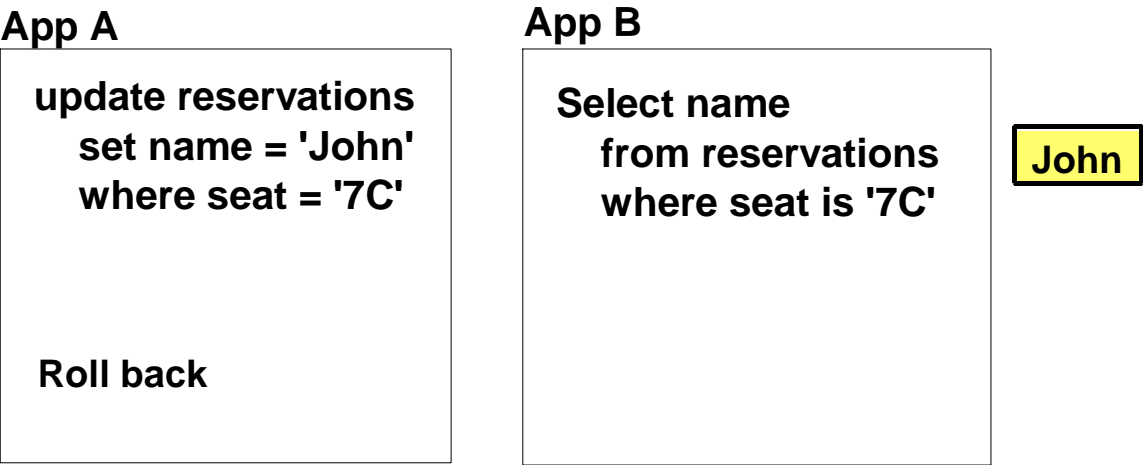
John

Next, application A issues a ROLLBACK

Uncommitted read (also known as “dirty read”)

reservations

seat	name	...
7C	_____	
7B	_____	
...		



which means any changes made would be discarded, so the corresponding name for seat '7C' is back to NULL.

Uncommitted read (also known as “dirty read”)

reservations

seat	name	...
7C	_____	
7B	_____	
...		

App A

update reservations
set name = 'John'
where seat = '7C'

Roll back

App B

Select name
from reservations
where seat is '7C'

John

Further processing in App
B uses incorrect /
uncommitted value of
“John”

05/26/2011

26

Template Documentation

© 2011 IBM Corporation

If application B continues further processing using “John”, it's actually using data that was not committed, and is incorrect.

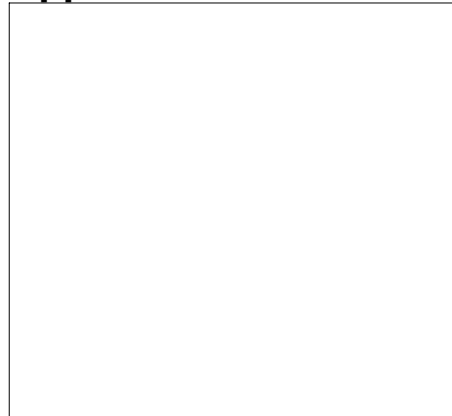
This problem is called “uncommitted read”.

Non-repeatable read

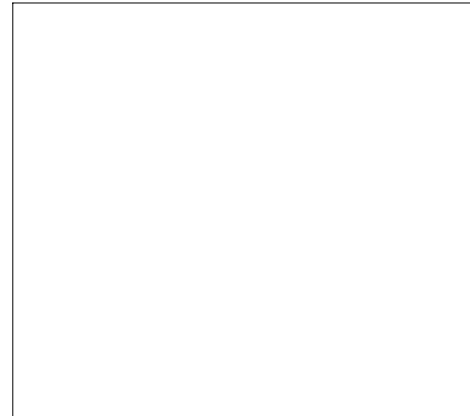
reservations

seat	name	...
7C	_____	
7B	_____	
...		

App A



App B



05/26/2011

27

Template Documentation

© 2011 IBM Corporation

Let's move on to another problem called 'non-repeatable read'.

In this example we have the same reservation table, and the same applications.

Non-repeatable read

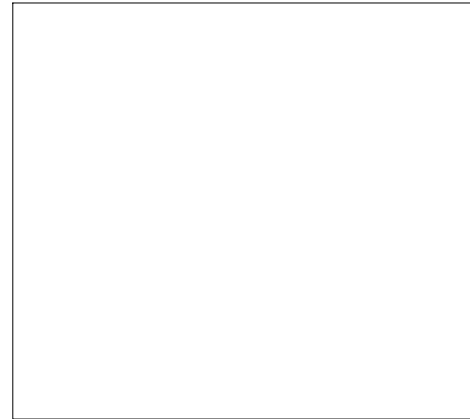
reservations

seat	name	...
7C	_____	
7B	_____	
...		

App A

```
select seat
from reservations
where name is NULL
```

App B



05/26/2011

28

Template Documentation

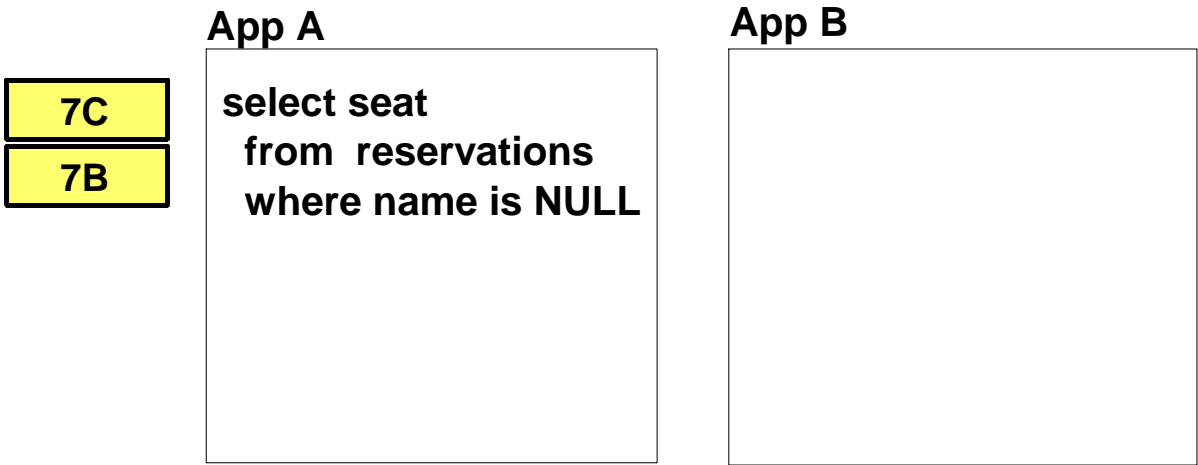
© 2011 IBM Corporation

Application A is this time issuing a SELECT statement

Non-repeatable read

reservations

seat	name	...
7C	_____	
7B	_____	
...		



which retrieves two seats '7C','7B'.

Non-repeatable read

reservations

seat	name	...
7C	_____	
7B	_____	
...		

App A

7C
7B

**select seat
from reservations
where name is NULL**

App B

**update reservations
set name = 'John'
where seat = '7C'**

05/26/2011

30

Template Documentation

© 2011 IBM Corporation

Then application B issues an update,

Non-repeatable read

reservations

seat	name	...
7C	John	
7B	_____	
...		

App A

7C
7B

select seat
from reservations
where name is NULL

App B

update reservations
set name = 'John'
where seat = '7C'

05/26/2011

31

Template Documentation

© 2011 IBM Corporation

so seat '7C' is assigned a passenger with name 'John'.

Non-repeatable read

reservations

seat	name	...
7C	John	
7B	_____	
...		

App A

7C
7B

```
select seat
from reservations
where name is NULL

...

select seat
from reservations
where name is NULL
```

App B

```
update reservations
set name = 'John'
where seat = '7C'
```

05/26/2011

32

Template Documentation

© 2011 IBM Corporation

When application A, within the same transaction issues another SELECT which is exactly the same as the first one it had issued,

Non-repeatable read

reservations

seat	name	...
7C	John	
7B	_____	
...		

App A

7C
7B

```
select seat
from reservations
where name is NULL
```

...

```
select seat
from reservations
where name is NULL
```

7B

App B

```
update reservations
set name = 'John'
where seat = '7C'
```

05/26/2011

33

Template Documentation

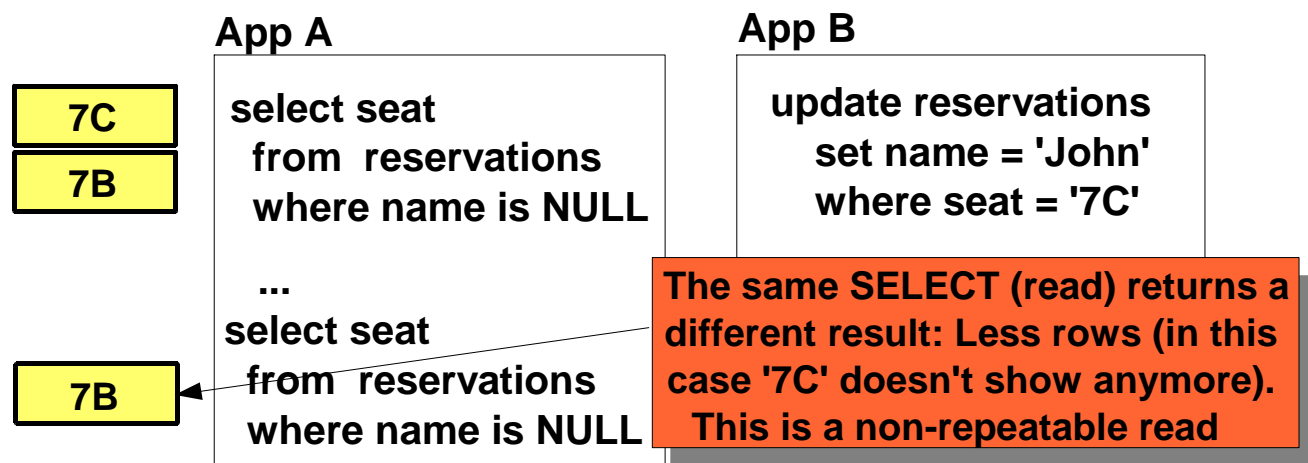
© 2011 IBM Corporation

this application will now get one seat: '7B'.

Non-repeatable read

reservations

seat	name	...
7C	John	
7B	_____	
...		



05/26/2011

34

Template Documentation

© 2011 IBM Corporation

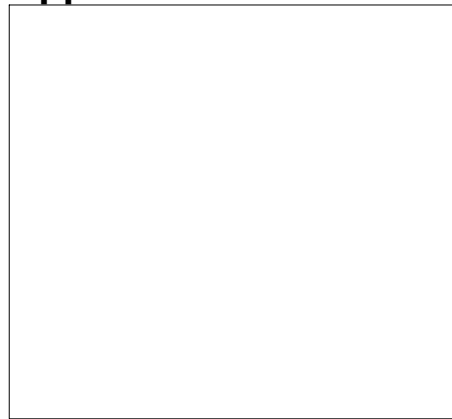
So it's not a repeatable read: Issuing the exact same SELECT statement within a transaction leads to different results.

Phantom read

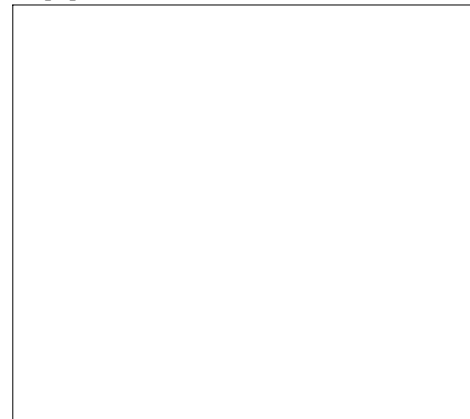
reservations

seat	name	...
7C	Susan	
7B	_____	
...		

App A



App B



05/26/2011

35

Template Documentation

© 2011 IBM Corporation

Now let's talk about the last problem called 'phantom read'.

Assume the reservations table has passenger 'Susan' assigned to seat '7C'.

Phantom read

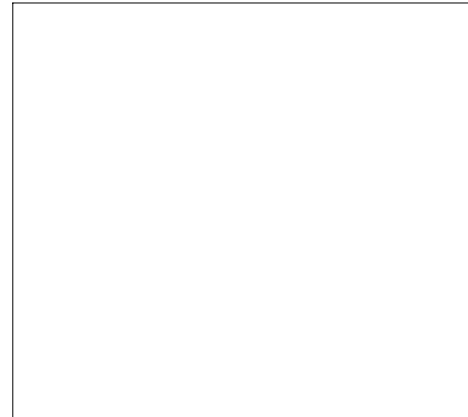
reservations

seat	name	...
7C	Susan	
7B	_____	
...		

App A

```
select seat
from reservations
where name is NULL
```

App B



05/26/2011

36

Template Documentation

© 2011 IBM Corporation

When application A executes a SELECT to retrieve all seats not assigned (where name is NULL),

Phantom read

reservations

seat	name	...
7C	Susan	
7B	_____	
...		

App A

App B

7B

select seat
from reservations
where name is NULL

it will return only one value: '7B'.

Phantom read

reservations

seat	name	...
7C	Susan	
7B	_____	
...		

App A

select seat
from reservations
where name is NULL

7B

App B

update reservations
set name = NULL
where seat = '7C'

05/26/2011

38

Template Documentation

© 2011 IBM Corporation

Next application B updates seat '7C'

Phantom read

reservations

seat	name	...
7C	_____	
7B	_____	
...		

App A

**select seat
from reservations
where name is NULL**

7B

App B

**update reservations
set name = NULL
where seat = '7C'**

05/26/2011

39

Template Documentation

© 2011 IBM Corporation

setting the name to NULL.

Phantom read

reservations

seat	name	...
7C	_____	
7B	_____	
...		

App A

**select seat
from reservations
where name is NULL**

...

**select seat
from reservations
where name is NULL**

App B

**update reservations
set name = NULL
where seat = '7C'**

7B

05/26/2011

40

Template Documentation

© 2011 IBM Corporation

If Application A executes again the exact same SELECT statement within the same transaction,

Phantom read

reservations

seat	name	...
7C	_____	
7B	_____	
...		

App A

select seat
from reservations
where name is NULL

...

select seat
from reservations
where name is NULL

App B

update reservations
set name = NULL
where seat = '7C'

05/26/2011

41

Template Documentation

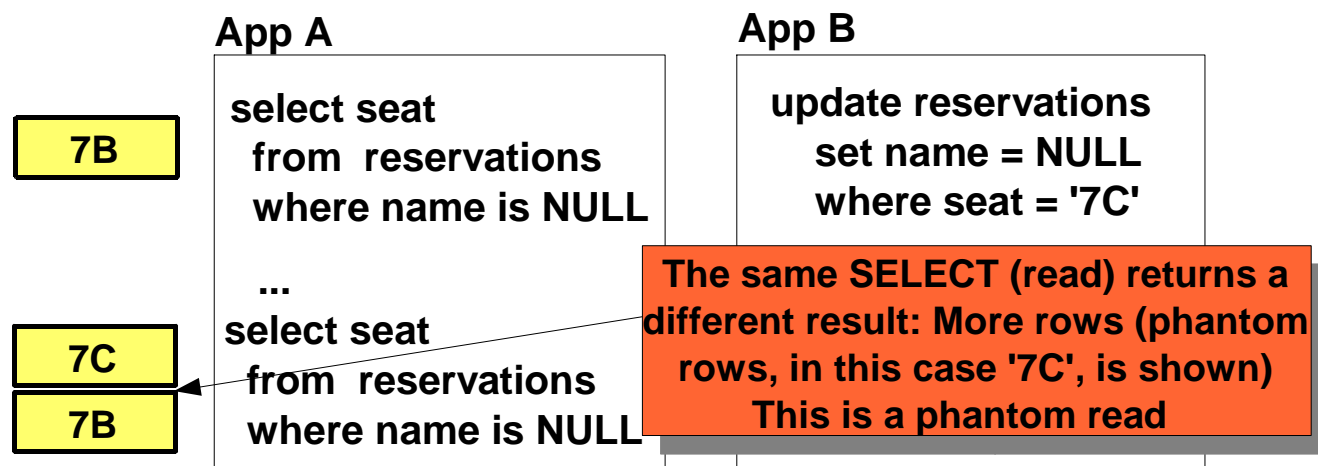
© 2011 IBM Corporation

it will now retrieve two rows: '7C' and '7B'.

Phantom read

reservations

seat	name	...
7C	_____	
7B	_____	
...		



05/26/2011

42

Template Documentation

© 2011 IBM Corporation

So in this case the same SELECT in the same transaction is retrieving one more row, a "phantom" row. This is why this case is called "Phantom Read". It is very similar to a non-repeatable case. In non-repeatable read you get less rows, while for "Phantom Read" you get more rows.

Isolation levels

- **“Policies” to control when locks are taken**
- **DB2 provides different levels of protection to isolate data**
 - Uncommitted Read (UR)
 - Cursor Stability (CS)
 - Currently committed (CC)
 - Read Stability (RS)
 - Repeatable Read (RR)

05/26/2011

43

Template Documentation

© 2011 IBM Corporation

You can think of isolation levels as policies on how you want DB2 to work with locks.

There are different isolation levels and the names are similar to the problems that we discussed earlier. So we have:

- Uncommitted read or UR, also known as dirty read,
- Cursor Stability or CS,
- Read Stability or RS,
- Repeatable Read or RR.

Setting the isolation levels

- **Isolation level can be specified at many levels**
 - Session (application),
 - Connection,
 - Statement
 - For statement level, use the WITH {RR, RS, CS, UR} clause:

```
SELECT COUNT(*) FROM tab1 WITH UR
```

- **For embedded SQL, the level is set at bind time**
- **For dynamic SQL, the level is set at run time**

05/26/2011

44

Template Documentation

© 2011 IBM Corporation

You can set these isolation levels at different levels:

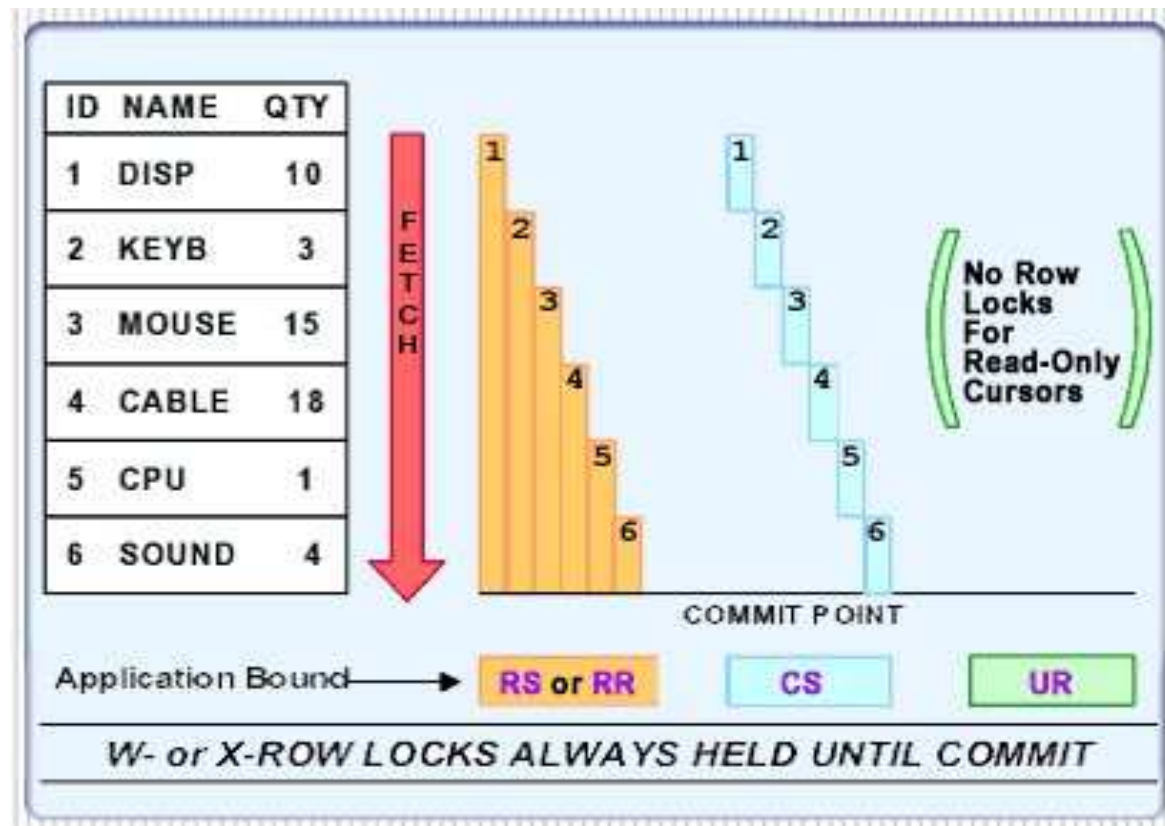
- You can set at the application level or session level, where the default is Cursor Stability.
- You can set at the Connection level;
- you can set at the Statement level.
- If you are using embedded SQL, you can set it at bind time.
- For dynamic SQL, you can set it at run time.

Example Scenario:

Application needs to get a "rough" count of how many rows are in table. Performance is of utmost importance. Cursor Stability isolation level is required with the exception of one SQL statement:

```
SELECT COUNT(*) FROM tab1 WITH UR
```

Comparing isolation levels



05/26/2011

Template Documentation

45

© 2011 IBM Corporation

Let's compare how each of these isolation levels work in terms of holding and releasing locks.

The figure shows a table on the left side with six rows. Let's see what happens when your application fetches rows from this table using different isolation levels.

Let's start with UR. As you can see, in this case no locks are taken as you fetch rows.

Now let's see what happens when using isolation level CS which is the default: You fetch row 1, and a lock is taken. When you move on to fetch row 2, the lock is released from row 1, and a lock is taken on row 2. When you move on to fetch row 3, the lock from row 2 is released, and a lock on row 3 is taken, and so on.

Now let's see what happens when using isolation RS or RR: You fetch row 1, and a lock is taken. When you move on to fetch row 2, the lock in row 1 still is held, and now a lock in row 2 is taken. When you move to fetch row 3, the lock in row 1 and row 2 still are held, and now a lock in row 3 is taken, and so on.

So as you can see, isolation level UR allows for maximum concurrency. While, RR or RS allows for the least concurrency. At the same time, UR provides the least accuracy in terms of results while RS or RR provides the most accuracy.

Cursor stability with currently committed (CC) semantics

- Cursor stability with *currently committed* semantics is the default isolation level
- Use `cur_commit` db cfg parameter to enable/disable
- Avoids timeouts and deadlocks

Cursor stability		Cursor stability with currently committed	
Situation	Result	Situation	Result
Reader blocks Reader	No	Reader blocks Reader	No
Reader blocks Writer	Maybe	Reader blocks Writer	No
Writer blocks Reader	Yes	Writer blocks Reader	No
Writer blocks Writer	Yes	Writer blocks Writer	Yes

05/26/2011

46

Template Documentation

© 2011 IBM Corporation

When using CS, there are two different behaviors (which are described in more detail in the next slide).

The behavior that prevents timeouts/deadlocks more is the one where “currently committed” is enabled.

The main difference is highlighted (row #3 in both tables):

Without currently committed, a writer would block a reader (i.e: An UPDATE would block a SELECT), so basically, the SELECT would have to wait.

With currently committed a writer does NOT block a reader. So even if one application is doing an UPDATE on a row, another application can do a SELECT on the same row.

Cursor stability with currently committed (CC) semantics

Cursor stability without currently committed

App A		reservations	App B	
		seat	name	...
		7C	Susan	
		7B	_____	
		...		

Cursor stability with currently committed (Default behavior)

App A		reservations	App B	
		seat	name	...
		7C	Susan	
		7B	_____	
		...		

05/26/2011

Template Documentation

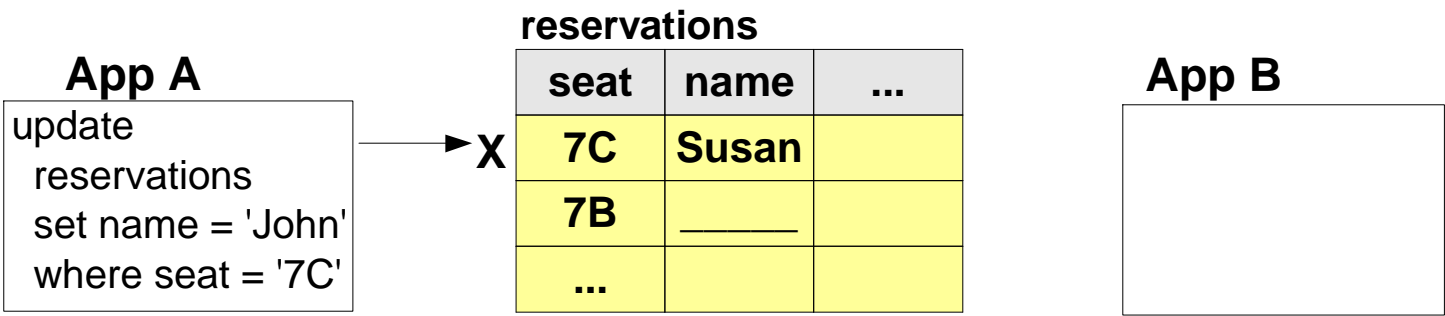
47

© 2011 IBM Corporation

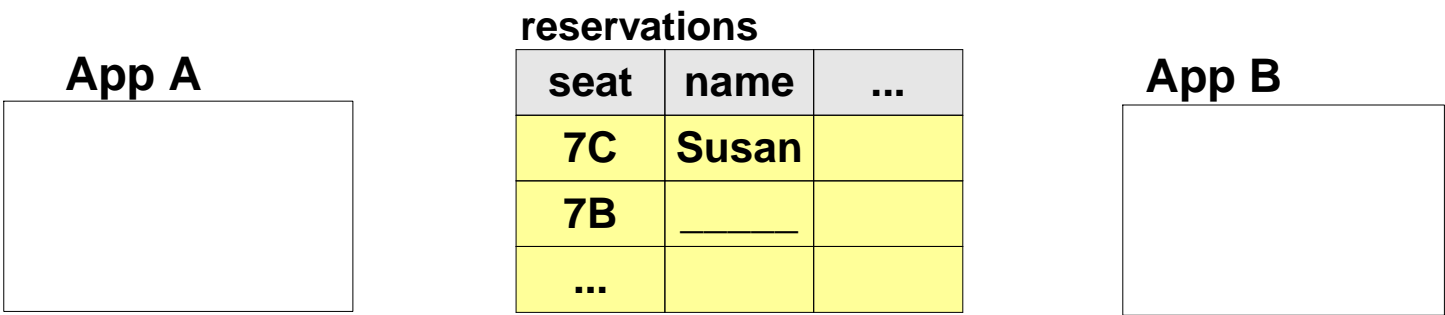
This slide shows the different behaviors when working with CS. At the top we will explain the behavior using CS without currently committed. And at the bottom we will explain the behavior using CS with currently committed.

Cursor stability with currently committed (CC) semantics

Cursor stability without currently committed



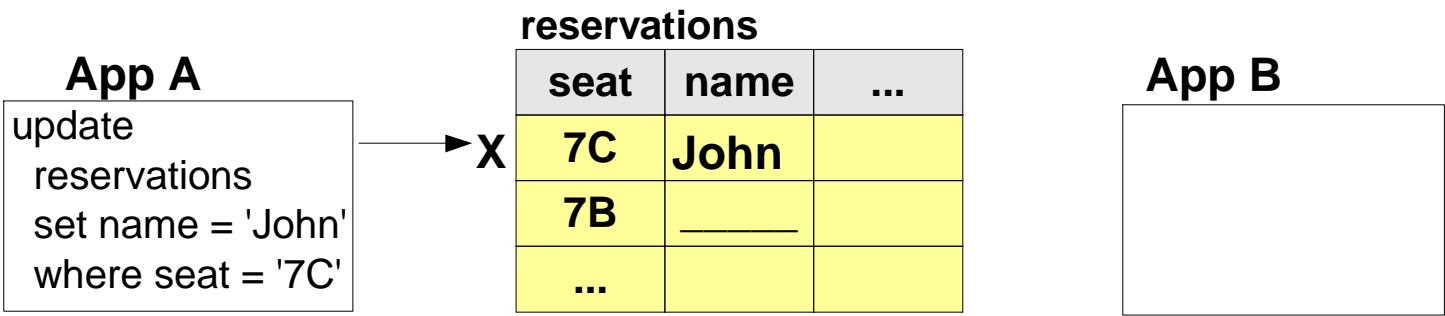
Cursor stability with currently committed (Default behavior)



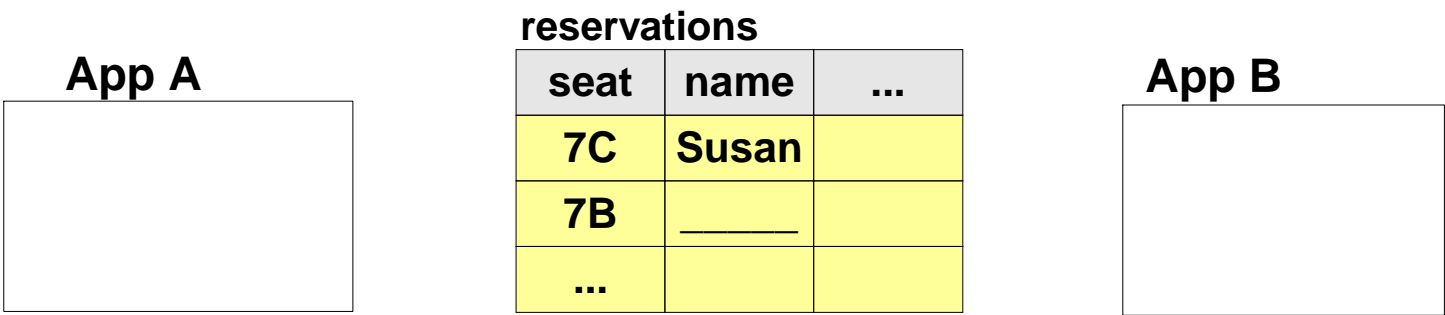
So let's start at the top. Let's say an application A performs an UPDATE which will have to take an X (exclusive) lock on the row.

Cursor stability with currently committed (CC) semantics

Cursor stability without currently committed

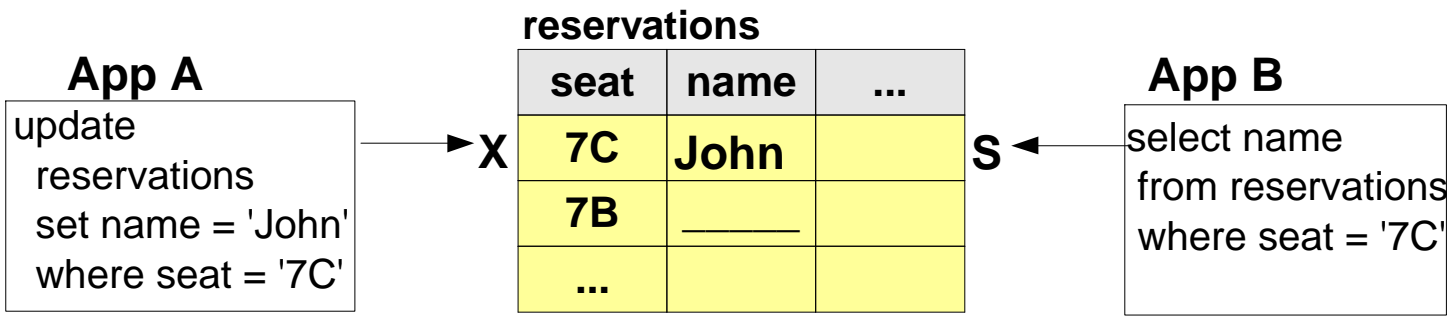


Cursor stability with currently committed (Default behavior)

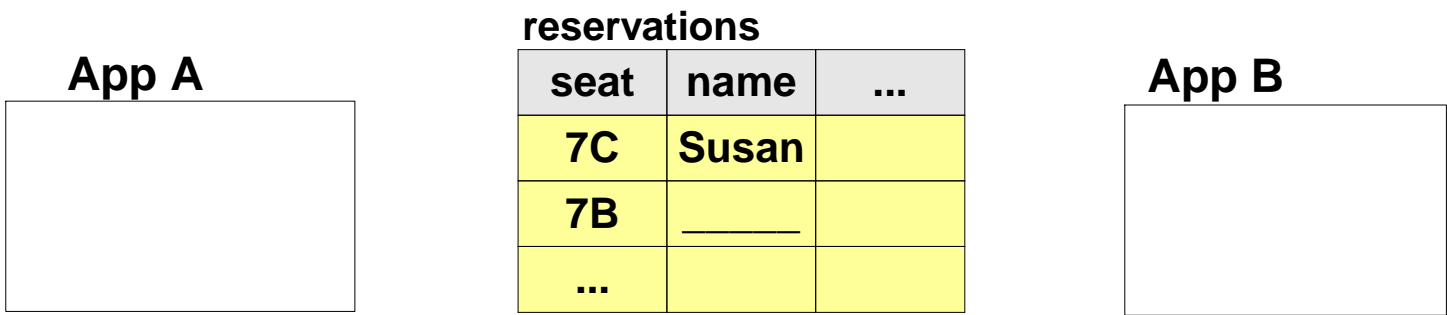


Cursor stability with currently committed (CC) semantics

Cursor stability without currently committed



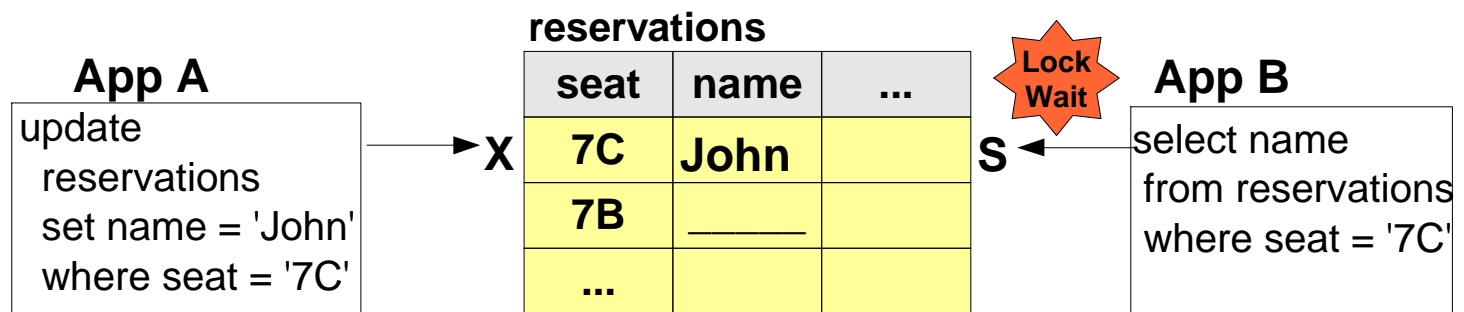
Cursor stability with currently committed (Default behavior)



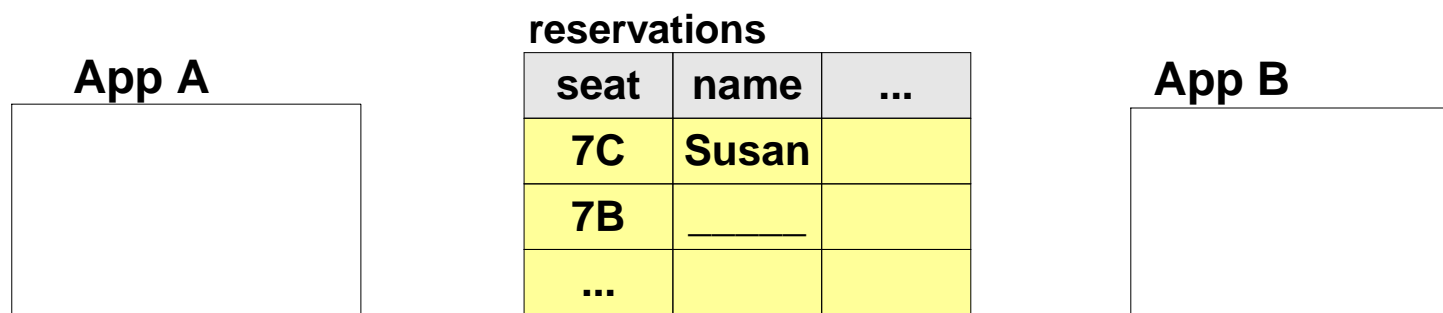
When application B running in CS without CC (currently committed) tries to perform a SELECT on the same row, it will request an 'S' (Shared) lock; which is not compatible with an X lock,

Cursor stability with currently committed (CC) semantics

Cursor stability without currently committed



Cursor stability with currently committed (Default behavior)



05/26/2011

51

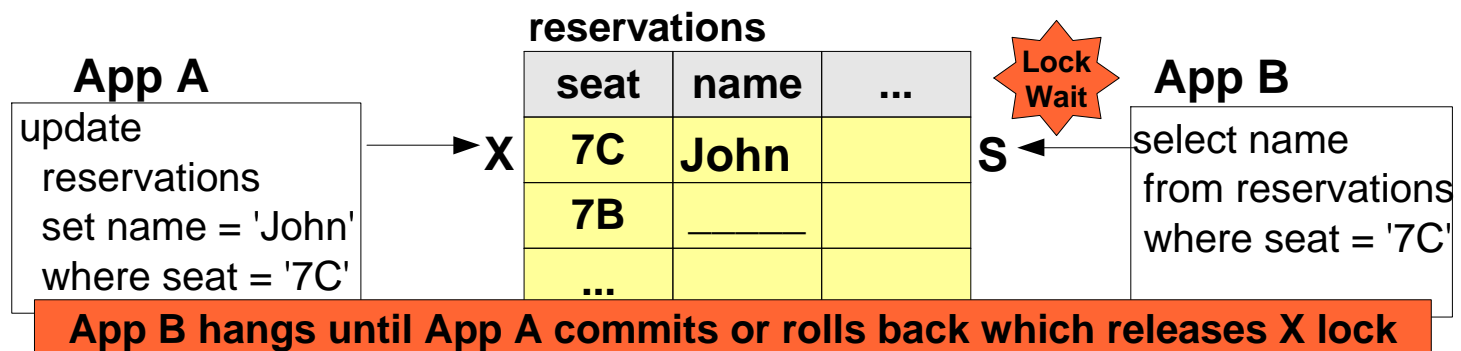
Template Documentation

© 2011 IBM Corporation

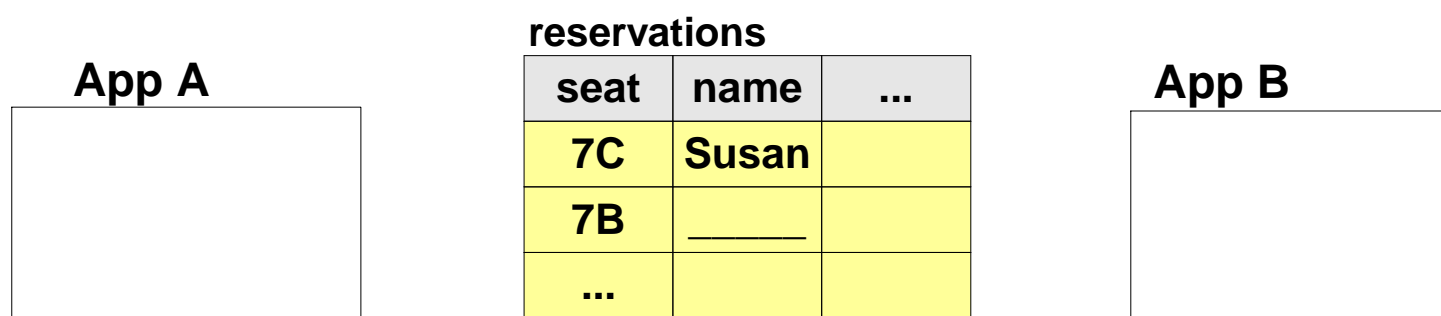
and thus it will have to wait.

Cursor stability with currently committed (CC) semantics

Cursor stability without currently committed



Cursor stability with currently committed (Default behavior)



05/26/2011

Template Documentation

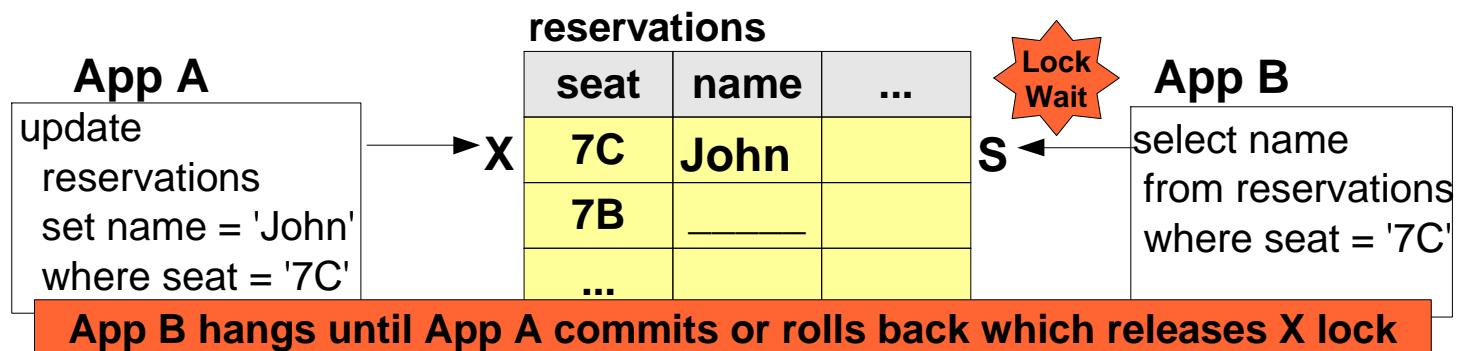
52

© 2011 IBM Corporation

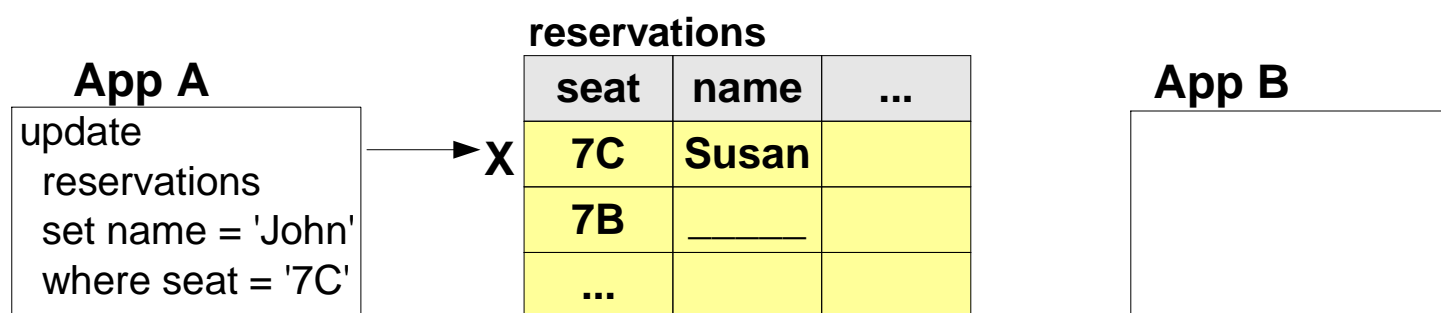
To the user, it will look as if application B is hanging. Only when application A commits, or rolls back will application B be able to proceed with the SELECT.

Cursor stability with currently committed (CC) semantics

Cursor stability without currently committed



Cursor stability with currently committed (Default behavior)



05/26/2011

Template Documentation

53

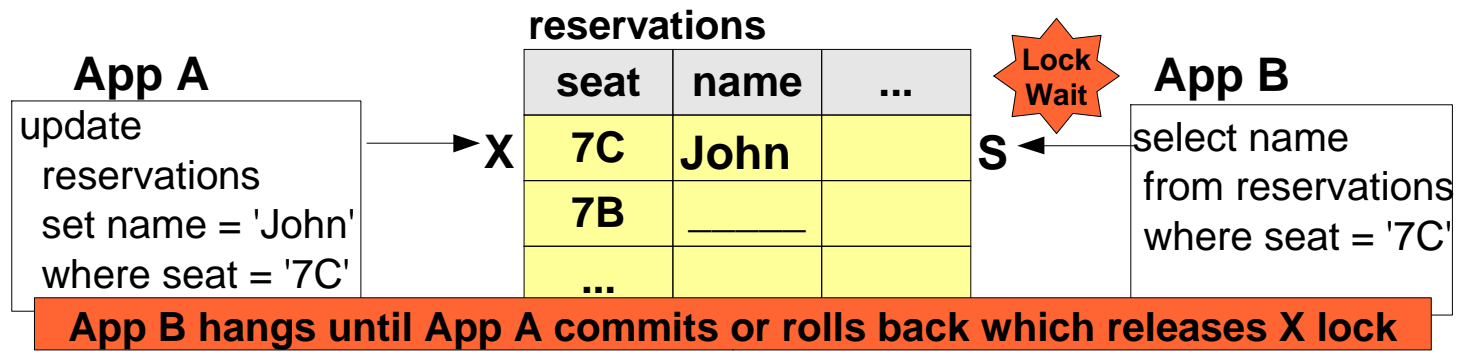
© 2011 IBM Corporation

Now, let's see how's the behavior at the bottom with CS using CC:

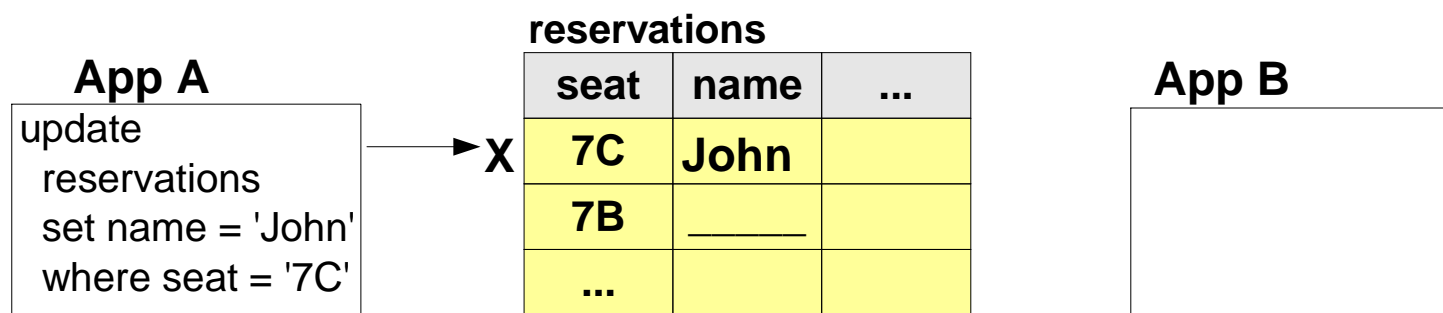
Using the same case, application A performs an update and takes a exclusive lock (X lock).

Cursor stability with currently committed (CC) semantics

Cursor stability without currently committed

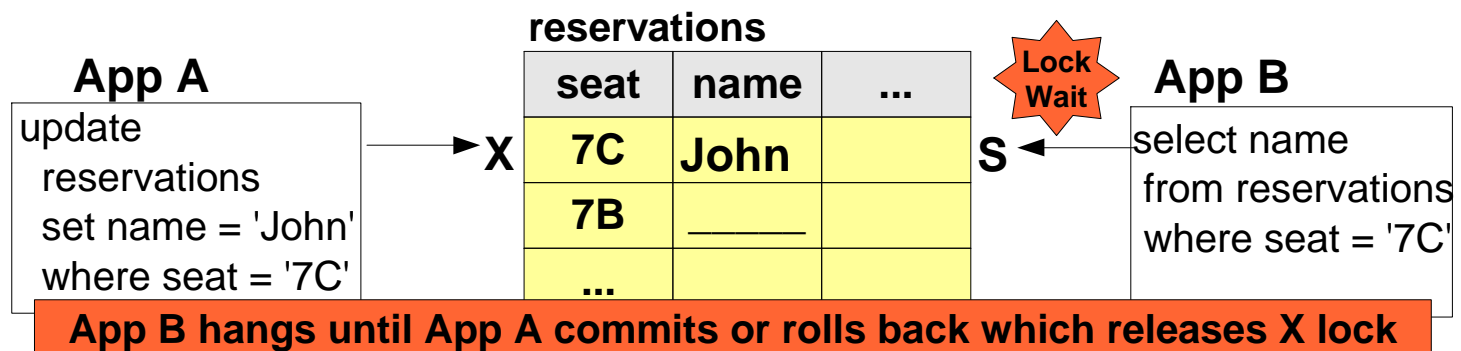


Cursor stability with currently committed (Default behavior)

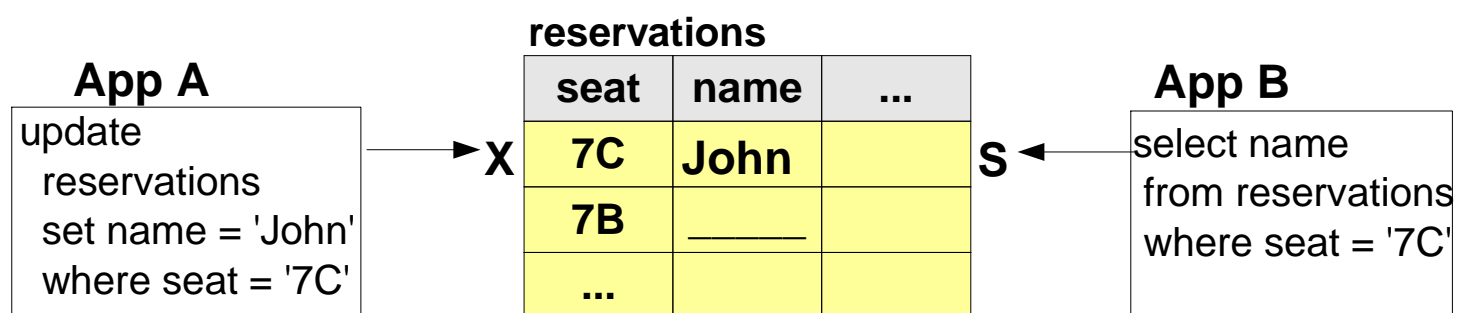


Cursor stability with currently committed (CC) semantics

Cursor stability without currently committed



Cursor stability with currently committed (Default behavior)



05/26/2011

Template Documentation

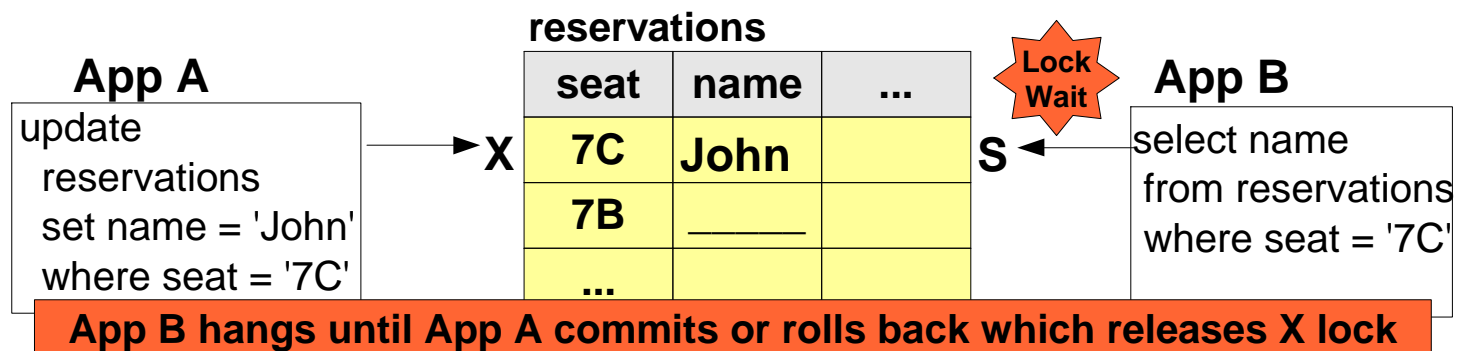
55

© 2011 IBM Corporation

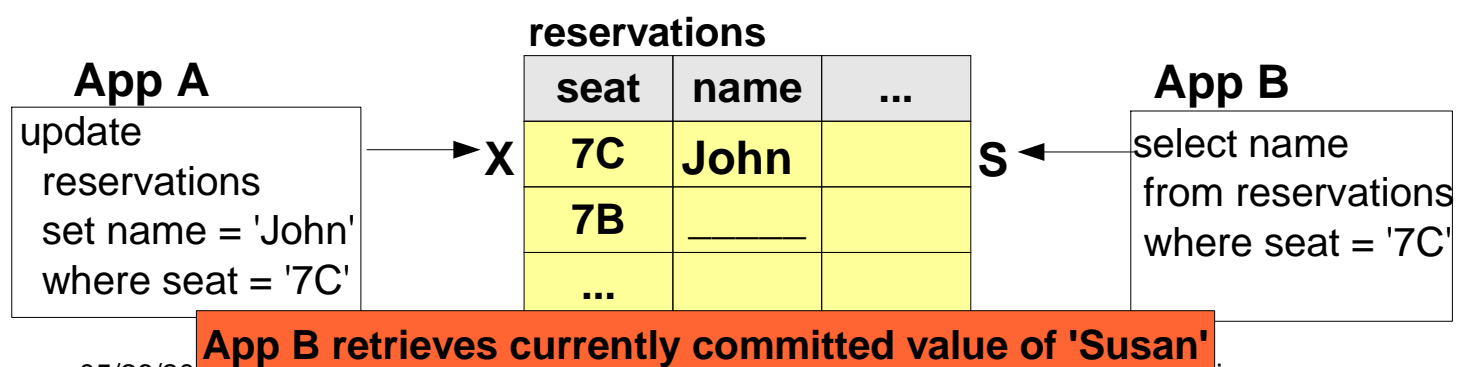
When application B issues a SELECT on the same row, it is allowed to proceed,

Cursor stability with currently committed (CC) semantics

Cursor stability without currently committed



Cursor stability with currently committed (Default behavior)



though it will retrieve the **currently committed** value, which in this example was "Susan". So in this case, the writer (update) does not block the reader (SELECT)

Note: If app B were using UR, the value retrieved would've been 'John' which is the uncommitted value; but under CS, you only can retrieve committed values.

Comparing and choosing an isolation level

Isolation Level	Lost update	Dirty Read	Non-repeatable Read	Phantom Read
Repeatable Read (RR)	-	-	-	-
ReadStability (RS)	-	-	-	Possible
Cursor Stability (CS)	-	-	Possible	Possible
Uncommitted Read (UR)	-	Possible	Possible	Possible

Application Type	High data stability required	High data stability not required
Read-write transactions	RS	CS
Read-only transactions	RS or RR	UR

05/26/2011

Template Documentation

57

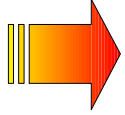
© 2011 IBM Corporation

This chart summarizes which concurrency problems are resolved by which isolation level. The “lost update” problem is resolved by all isolation levels.

The table at the bottom provides you with an idea of when to use which type of isolation level, depending on your application type and “stability” of your data, meaning, how accurate you want it to be.

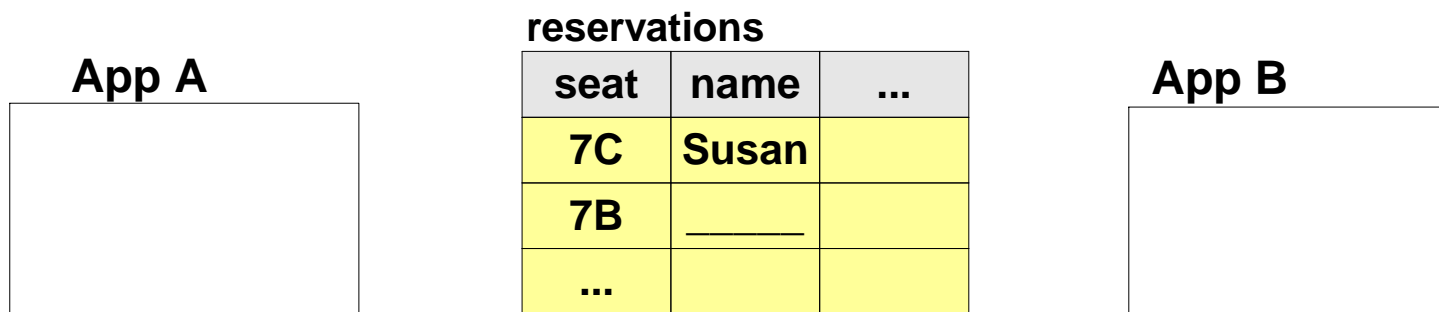
Agenda

- Transactions
- Concurrency & Locking
- Lock Wait
- Deadlocks



Lock wait

- By default, an application waits indefinitely to obtain any needed locks
- **LOCKTIMEOUT** (db cfg):
 - Specifies the number of seconds to wait for a lock
 - Default value is -1 or *infinite* wait
- Example: (Same as when using isolation CS without CC):



05/26/2011

59

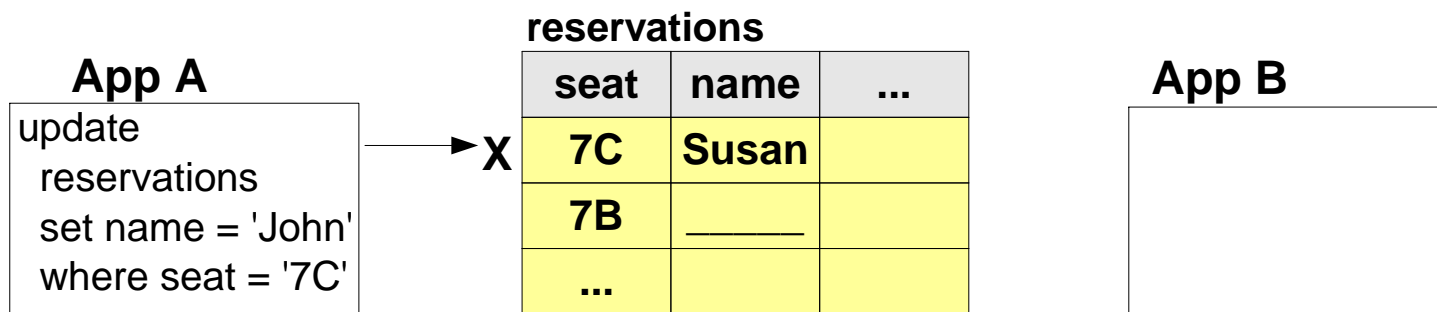
Template Documentation

© 2011 IBM Corporation

You have already seen how LOCK WAIT works when we talked about Cursor stability without Currently Committed. Let's revisit that example:

Lock wait

- By default, an application waits indefinitely to obtain any needed locks
- **LOCKTIMEOUT** (db cfg):
 - Specifies the number of seconds to wait for a lock
 - Default value is -1 or *infinite* wait
- Example: (Same as when using isolation CS without CC):



05/26/2011

60

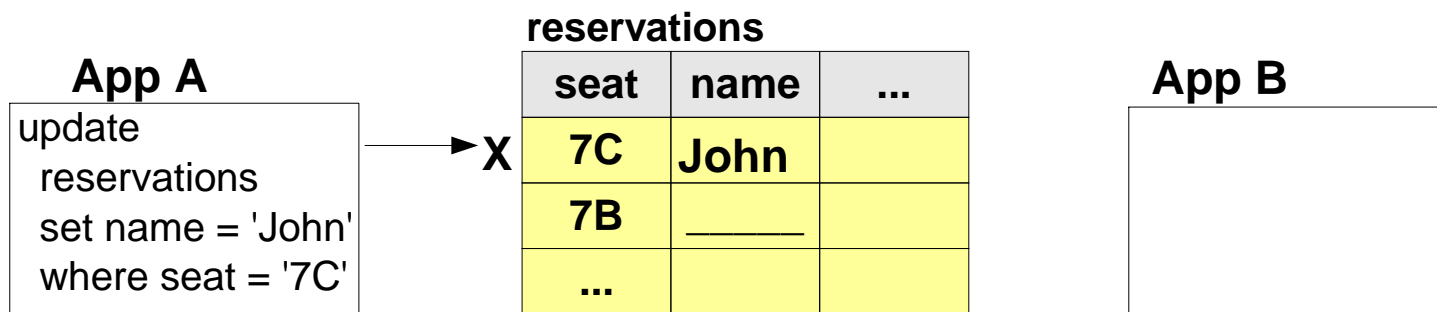
Template Documentation

© 2011 IBM Corporation

Application A performs an update taking an X lock.

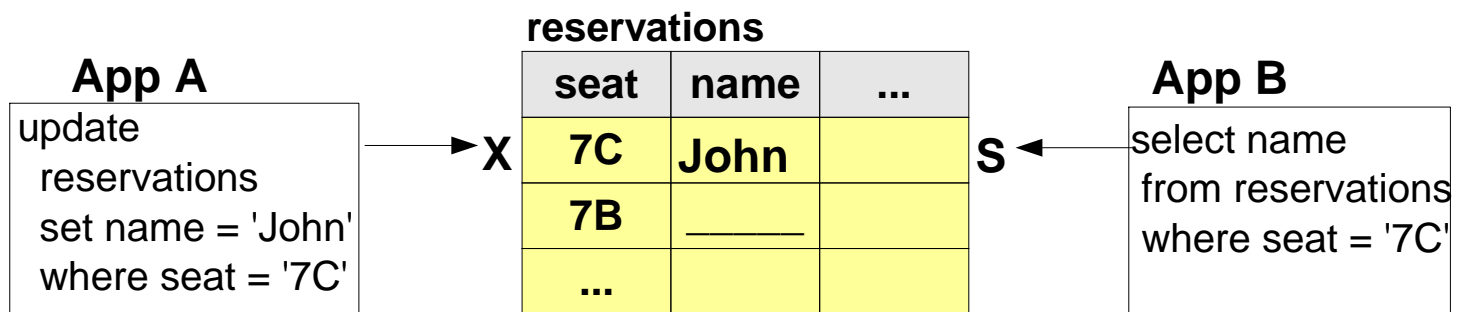
Lock wait

- By default, an application waits indefinitely to obtain any needed locks
- **LOCKTIMEOUT** (db cfg):
 - Specifies the number of seconds to wait for a lock
 - Default value is -1 or *infinite* wait
- Example: (Same as when using isolation CS without CC):



Lock wait

- By default, an application waits indefinitely to obtain any needed locks
- **LOCKTIMEOUT** (db cfg):
 - Specifies the number of seconds to wait for a lock
 - Default value is -1 or *infinite* wait
- Example: (Same as when using isolation CS without CC):



05/26/2011

62

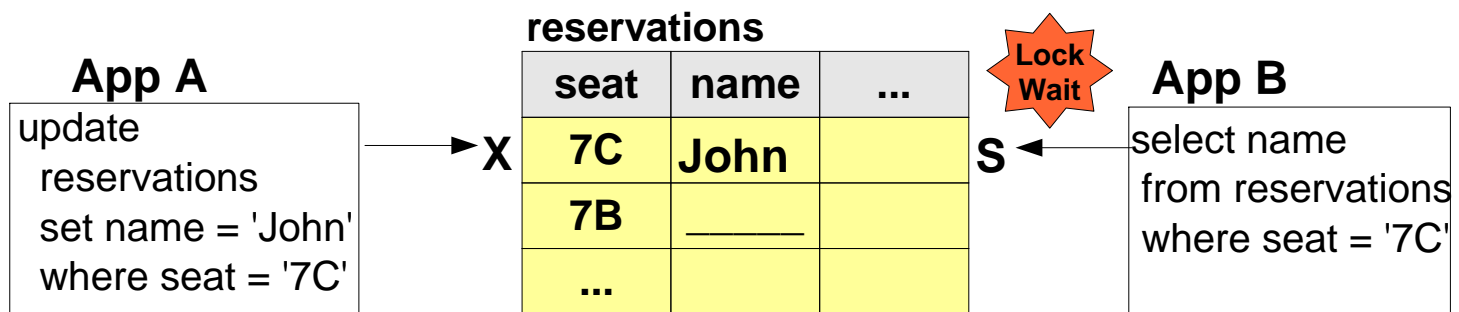
Template Documentation

© 2011 IBM Corporation

Application B wants to read from the same row (which would take an S lock) but it cannot since X and S locks are not compatible in CS without CC.

Lock wait

- By default, an application waits indefinitely to obtain any needed locks
- **LOCKTIMEOUT** (db cfg):
 - Specifies the number of seconds to wait for a lock
 - Default value is -1 or *infinite* wait
- Example: (Same as when using isolation CS without CC):



05/26/2011

63

Template Documentation

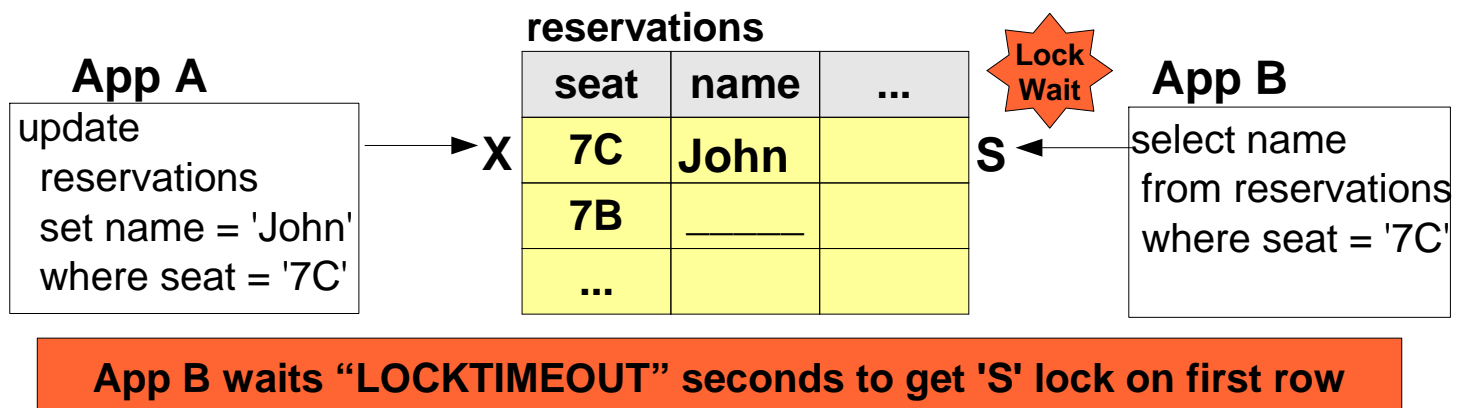
© 2011 IBM Corporation

Therefore, application B must wait.

How long will it wait?

Lock wait

- By default, an application waits indefinitely to obtain any needed locks
- **LOCKTIMEOUT** (db cfg):
 - Specifies the number of seconds to wait for a lock
 - Default value is -1 or *infinite* wait
- Example: (Same as when using isolation CS without CC):



05/26/2011

Template Documentation

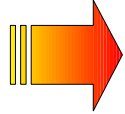
64

© 2011 IBM Corporation

This is set by the parameter LOCKTIMEOUT. By default, this parameter is set to -1, which means infinite wait.

Agenda

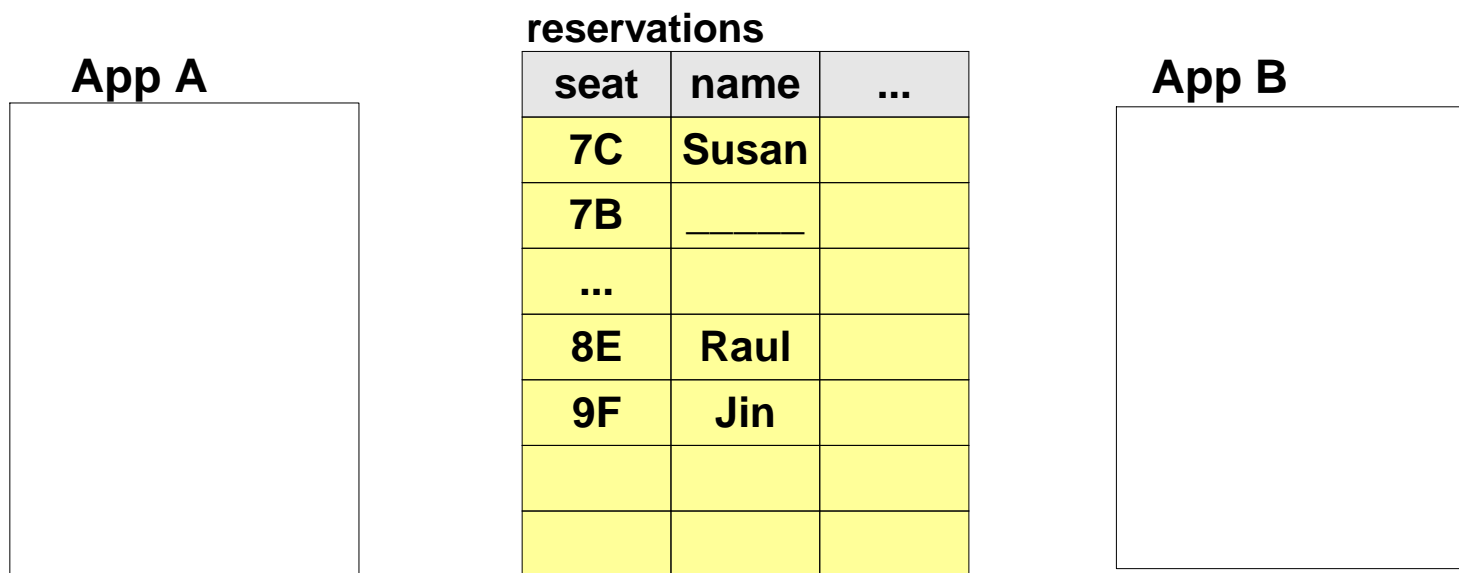
- Transactions
- Concurrency & Locking
- Lock Wait



- Deadlocks

Deadlocks

- Occurs when two or more applications wait indefinitely for a resource
- Each application is holding a resource that the other needs
- Waiting is never resolved
- In the example, assume we are using isolation CS without CC



05/26/2011

66

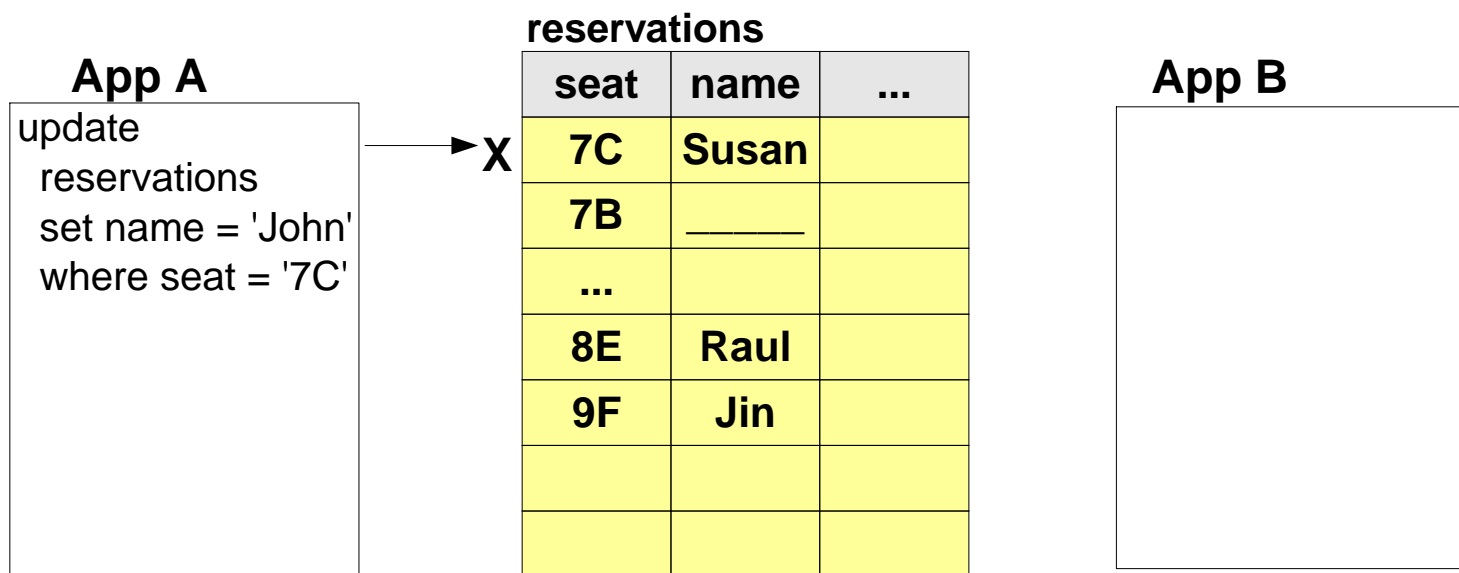
Template Documentation

© 2011 IBM Corporation

The example shows how a deadlock occurs. Note that the operations within App A occur within one transaction. Similarly for App B. This is important to note because, for example, if App A was doing an UPDATE, and then it COMMITTED, it would release all locks and we would have a different scenario altogether.

Deadlocks

- Occurs when two or more applications wait indefinitely for a resource
- Each application is holding a resource that the other needs
- Waiting is never resolved
- In the example, assume we are using isolation CS without CC



05/26/2011

67

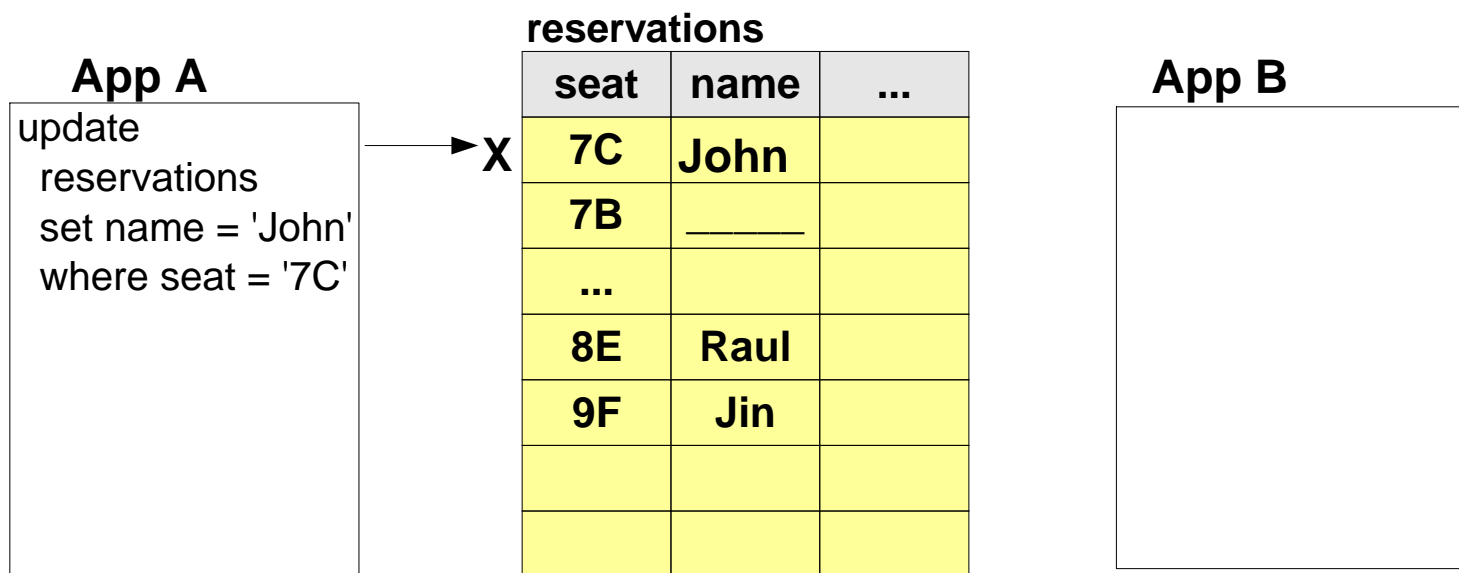
Template Documentation

© 2011 IBM Corporation

App A performs an update, so takes an X lock on row with seat 7C

Deadlocks

- Occurs when two or more applications wait indefinitely for a resource
- Each application is holding a resource that the other needs
- Waiting is never resolved
- In the example, assume we are using isolation CS without CC



05/26/2011

68

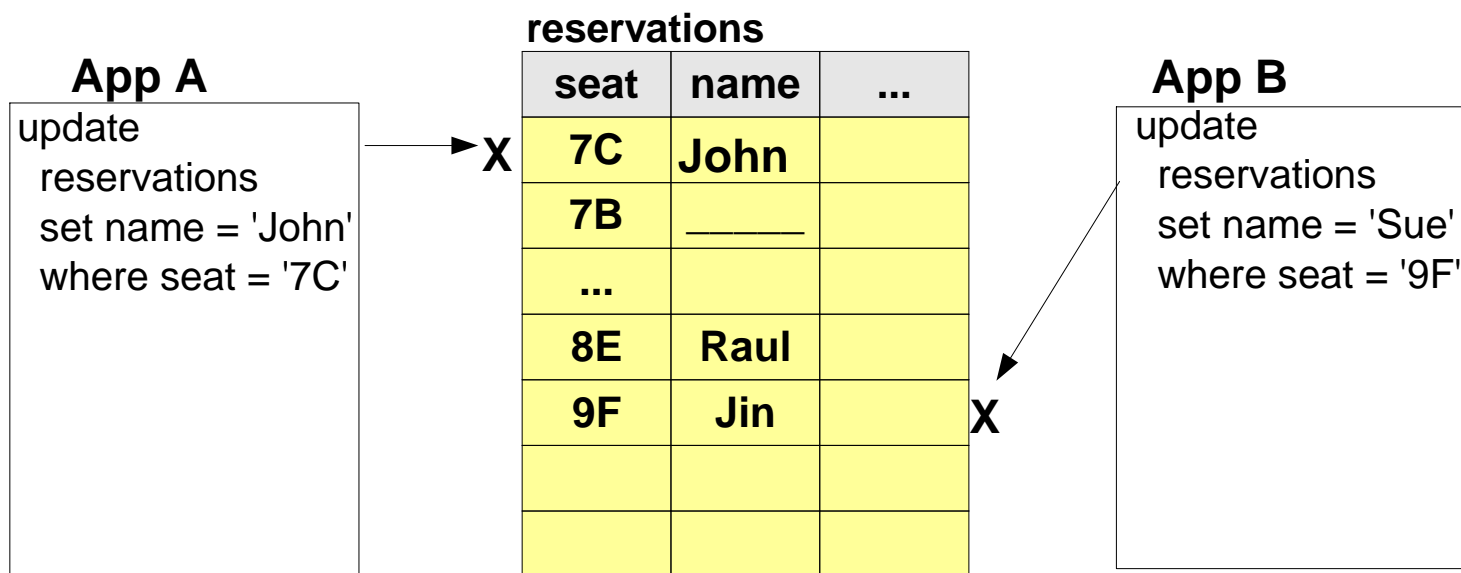
Template Documentation

© 2011 IBM Corporation

The row is updated with “John”, though the change is not yet committed.

Deadlocks

- Occurs when two or more applications wait indefinitely for a resource
- Each application is holding a resource that the other needs
- Waiting is never resolved
- In the example, assume we are using isolation CS without CC



05/26/2011

69

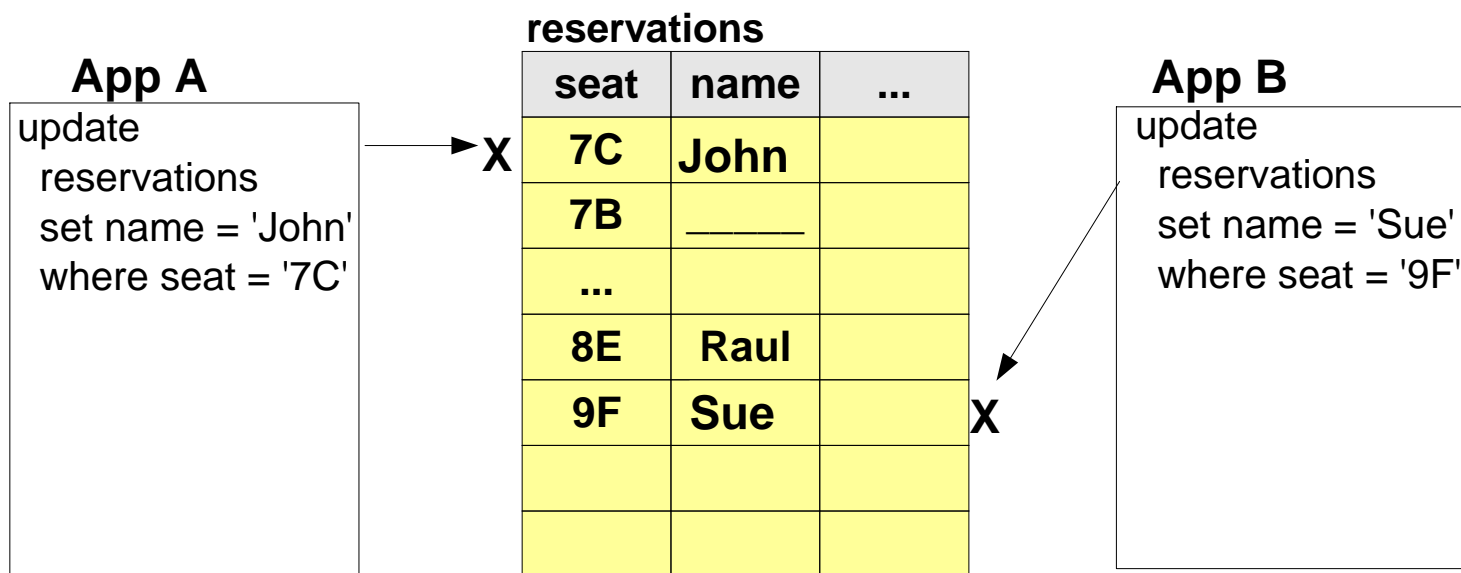
Template Documentation

© 2011 IBM Corporation

App B performs an update on another row, the one with seat '9F'. It also will take an X lock on this row

Deadlocks

- Occurs when two or more applications wait indefinitely for a resource
- Each application is holding a resource that the other needs
- Waiting is never resolved
- In the example, assume we are using isolation CS without CC



05/26/2011

70

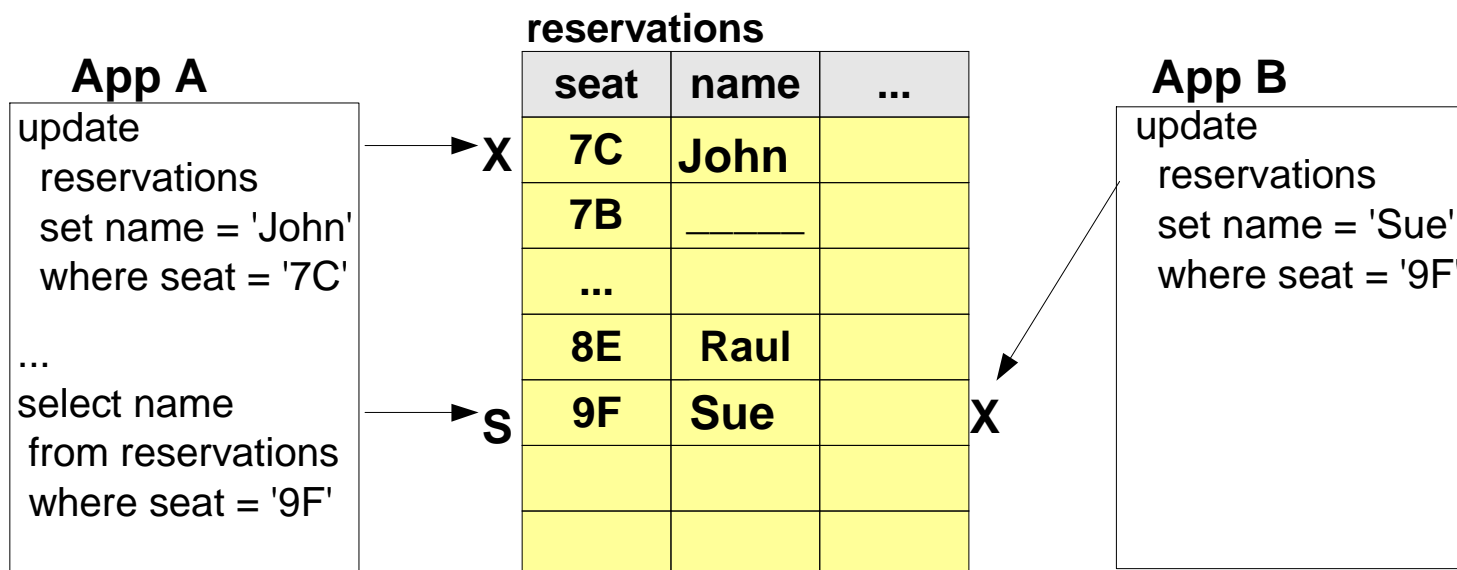
Template Documentation

© 2011 IBM Corporation

The row is updated with “Sue”, though the change is not yet committed.

Deadlocks

- Occurs when two or more applications wait indefinitely for a resource
- Each application is holding a resource that the other needs
- Waiting is never resolved
- In the example, assume we are using isolation CS without CC



05/26/2011

71

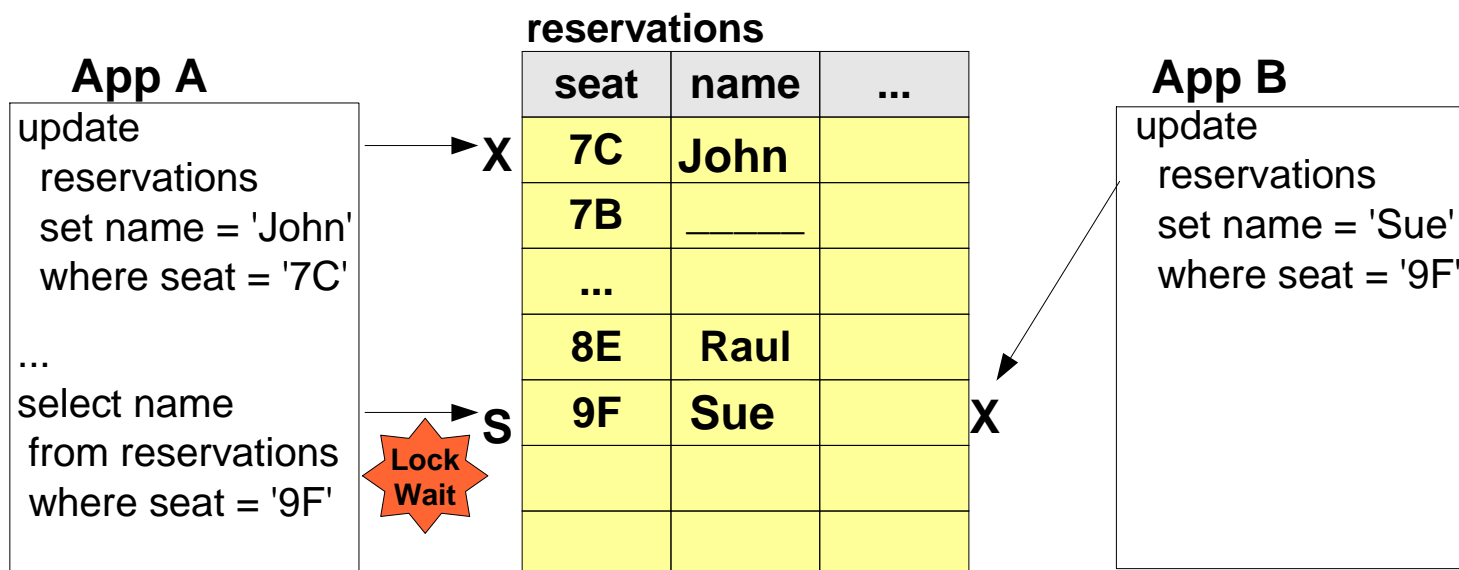
Template Documentation

© 2011 IBM Corporation

App A issues a SELECT on the row with seat 9F, so it wants to take an S lock on the row.

Deadlocks

- Occurs when two or more applications wait indefinitely for a resource
- Each application is holding a resource that the other needs
- Waiting is never resolved
- In the example, assume we are using isolation CS without CC



05/26/2011

72

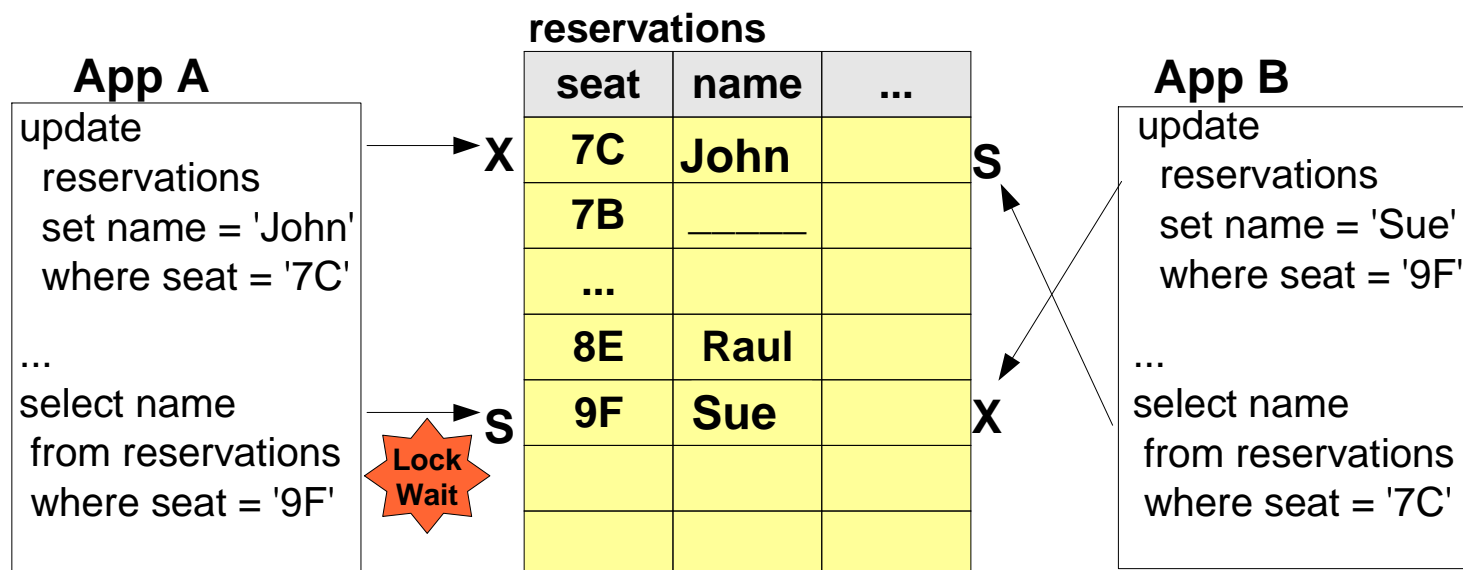
Template Documentation

© 2011 IBM Corporation

App A hangs. It has to wait since App B has an X lock on row with seat 9F

Deadlocks

- Occurs when two or more applications wait indefinitely for a resource
- Each application is holding a resource that the other needs
- Waiting is never resolved
- In the example, assume we are using isolation CS without CC



05/26/2011

73

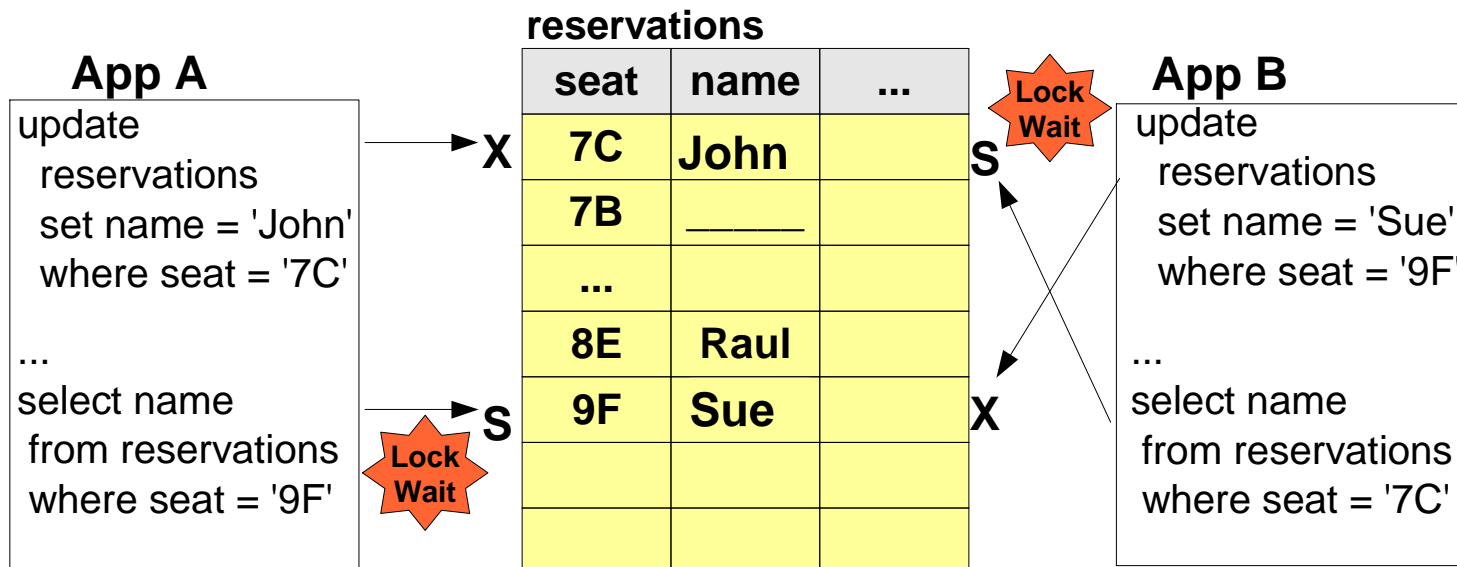
Template Documentation

© 2011 IBM Corporation

App B issues a SELECT on the row with seat 7C, so it wants to take an S lock on the row.

Deadlocks

- Occurs when two or more applications wait indefinitely for a resource
- Each application is holding a resource that the other needs
- Waiting is never resolved
- In the example, assume we are using isolation CS without CC



05/26/2011

Template Documentation

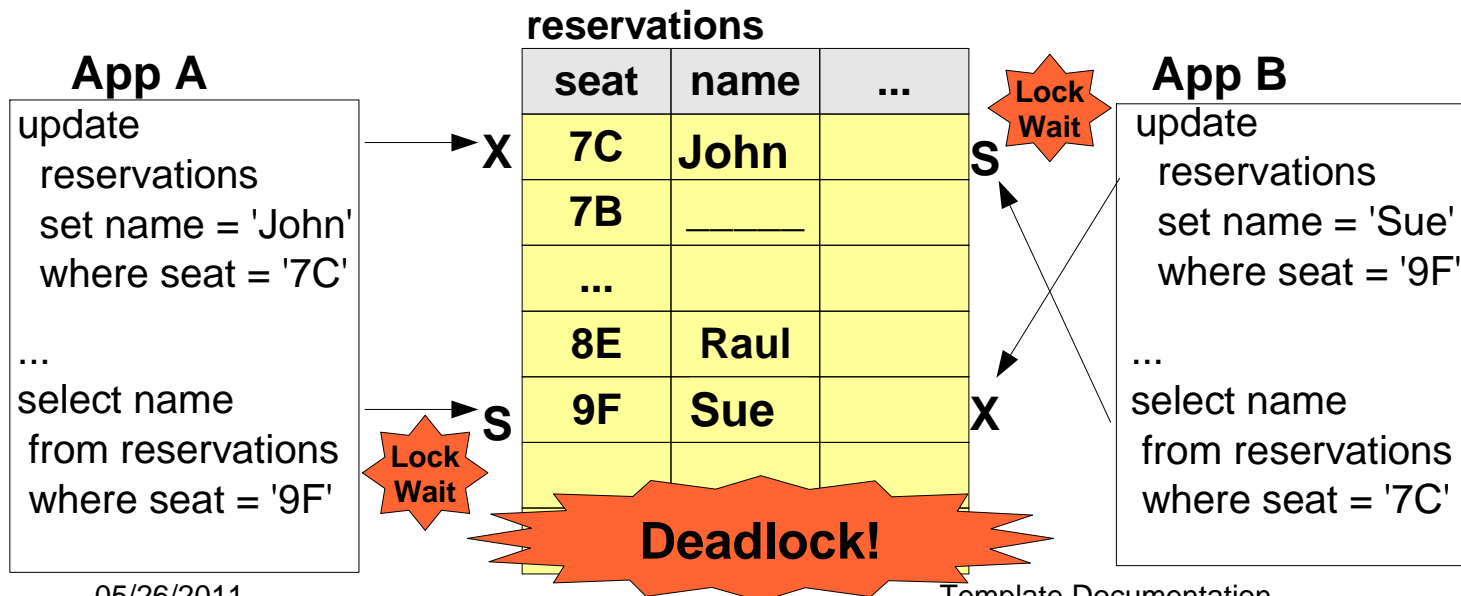
74

© 2011 IBM Corporation

App B hangs. It has to wait since App A has an X lock on row with seat 7C

Deadlocks

- Occurs when two or more applications wait indefinitely for a resource
- Each application is holding a resource that the other needs
- Waiting is never resolved
- In the example, assume we are using isolation CS without CC



05/26/2011

75

Template Documentation

© 2011 IBM Corporation

So we have a deadlock!. App A is waiting for App B to release its locks before it can continue, and App B is waiting for App A to release its locks before it can continue.

Deadlocks

- **Deadlocks are commonly caused by poor application design**
- **DB2 provides a deadlock detector**
 - Use **DLCHKTIME** (db cfg) to set the time interval for checking for deadlocks
 - When a deadlock is detected, DB2 uses an internal algorithm to pick which transaction to roll back, and which one to continue.
 - The transaction that is forced to roll back gets a SQL error. The rollback causes all of its locks to be released

05/26/2011

Template Documentation

76

© 2011 IBM Corporation

Normally deadlocks are situations that happen due to bad application design. DB2 has a deadlock detector that checks every 'x' number of seconds if there is a deadlock. 'x' is set by the deadlock check time (DLCHKTIME) db cfg parameter. When the deadlock detector detects a deadlock, DB2 will use an internal algorithm to pick one of the two applications that is causing the deadlock, and it will roll it back. The exact SQL error the application that is rolled back receives is:

SQLCODE -911 (SQLSTATE 40001), with reason code 2 (Where reason code '2' means "The transaction was rolled back due to a deadlock". The other application will be allowed to continue



Thank you!

Use the forum in the db2university.com course AA001EN if you have technical questions about the materials covered in this course. Fellow students, faculty and IBMers can help you!