4
Concurrency Protocols
Part 1: 2PL

Reading: [Kl], chapter 7; [Ha] chapter 9; [Me] chapter 4

Transactions and Concurreny Control

1. Transactions and their ACID properties
    1. A
    2. C
    3. I
    4. D

2. Concurrency anomalies and isolation levels

3. Serializability

eva.knirsch@kiu.edu.ge

Concurrency Control Protocols

Pessimistic: Lock Based Protocols

Locking Protocols based on 2PL
- introduced about 50 years ago (1976)
- for about 30 years the only CC protocol used in commercial DBMS to enforce serializability.
- Widespread, benchmark for other protocols
- Today, MS SQL Server uses 2PL (default), MySQL / MariaDB partly use 2PL

Optimistic: MVCC (Snapshot)

MVCC (snapshot) protocols
- introduced about 30 years ago
- Today, used by most commercial DBMS
- PostgreSQL, Oracle, MySQL / MariaDB (partly)
- MVCC also uses locks – but differently

Lock-Based CC Protocols

Pesimistic

A transaction acquires locks to lock the objects it wants to access. Other transactions have to wait until the objects are unlocked – depending on the lock and on the intended operation, read or write.

Two lock modes

S (shared, read lock):

X (exclusive, write lock):

Compatibility matrix

	No Lock NL	Shared S	Exclusive X
S	✓	✓	–
X	✓	–	–

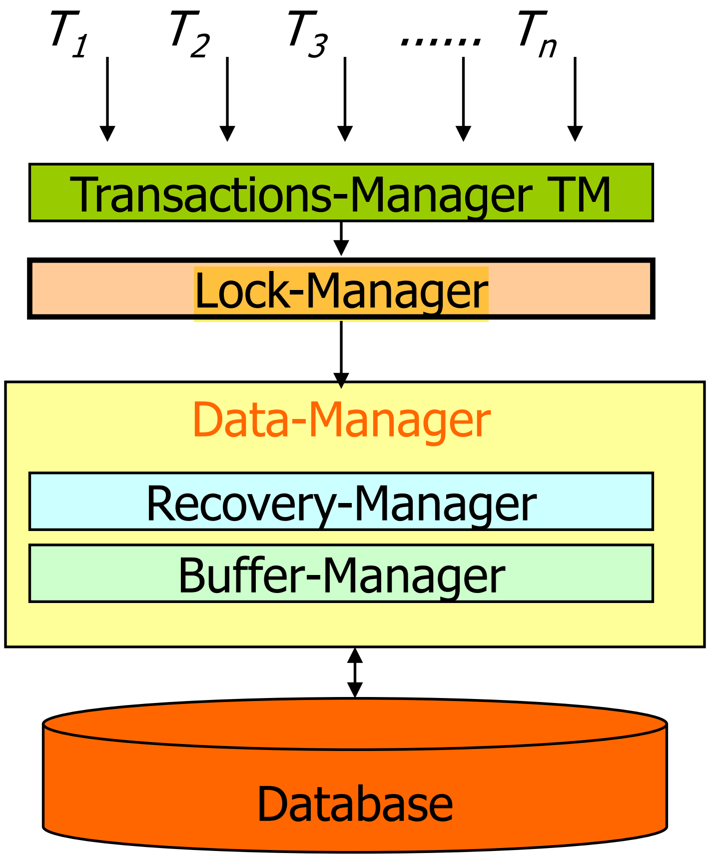
If there is no log then transaction can come and acquire read lock or write lock

if there is a shared lock another transaction can come and acquire another shared lock. So more than 1 transactions can get shared locks because they don't change or write on the data, they only read.

As soon as one transaction holds shared lock and other transaction wants to write, this isn't possible, then the protocol says to wait OR one transaction holds the write lock and other one wants to read, this is not possible either.

If both transactions want to write does not work either, one has a lock and other one has to wait

eva.knirsch@kiu.edu.ge



The lock manager maintains lock tables with
- locked objects,
- transactions that have locked these objects, and
- the details of the lock mode.

The lock manager also keeps waiting lists (intention locks) with the transactions waiting to lock an object.

Adapted from [Ke], Concurrency Control

KUTAISI INTERNATIONAL UNIVERSITY

Lock-based CC Protocols

An airline serves two routes: Kutaisi - Munich and Kutaisi - Berlin.

- Both routes cost the same: 100.00 € each.
- The airline increases its prices by 10 percent (application 1).
- The airline also has to add a CO_2 tax of 10 € to the price (application 2).
- The two routes are to cost the same after the increases. That is an invariant.

Which consistency-preserving results are possible?

both of them should be:
120: increase price, then add tax
OR
121: add tax, then increase price

eva.knirsch@kiu.edu.ge

Lock-based CC Protocols

What result do we get if the codes run in different sessions
– let us assume in autocommit mode - as follows:

A1 read(a)

A1 write(a) a:=a*1.1

A2 read(a)

A2 write(a) a:=a+10

A2 read(b)

A2 write(b) b:=b+10

A1 read(b)

A1 write(b) b:=b*1.1

Outcome: Invariant is violated

both need to cost the same but in this way one route costs 120 and other 121

eva.knirsch@kiu.edu.ge

Operation	T1	T2
1	S-lock (a)	
2	read(a)	
3	X-lock (a)	
4	write(a) a:= a*1.1	
5	Unlock(a)	S-lock (a)
6		Read(a)
7		X-lock (a)
8		Write(a) a: = a+10
9		Unlock(a)
10		s-Lock(b)
11		Read(b)
12		X-lock (b)
13		Write(b) b:=b+10
14		unlock (b)
15	s-Lock(b)	
16	read(B)	
17	X-lock (b)	
18	Write(b) b:=b*1.1	
19	Unlock(b)	

problem is that T1 unlocks a before it locks b

which allows T2 to acquire the lock on a and modify it before T1 finishes updating both a and b consistently

Running the code using locks.

Result?
invariant violated

How to set the locks to prevent this?

2-phase-Lock (2PL)

Even though locks were set, the result violates consistency. We need locks that guarantee serializability.

The two-phase locking protocol (2PL) guarantees serializability.

The 2PL protocol splits each transaction in two phases:

1. lock phase: all locks are set
2. release phase: locks are released.

→ In a first phase, a transaction must acquire all needed locks. During this phase, no lock may be released.

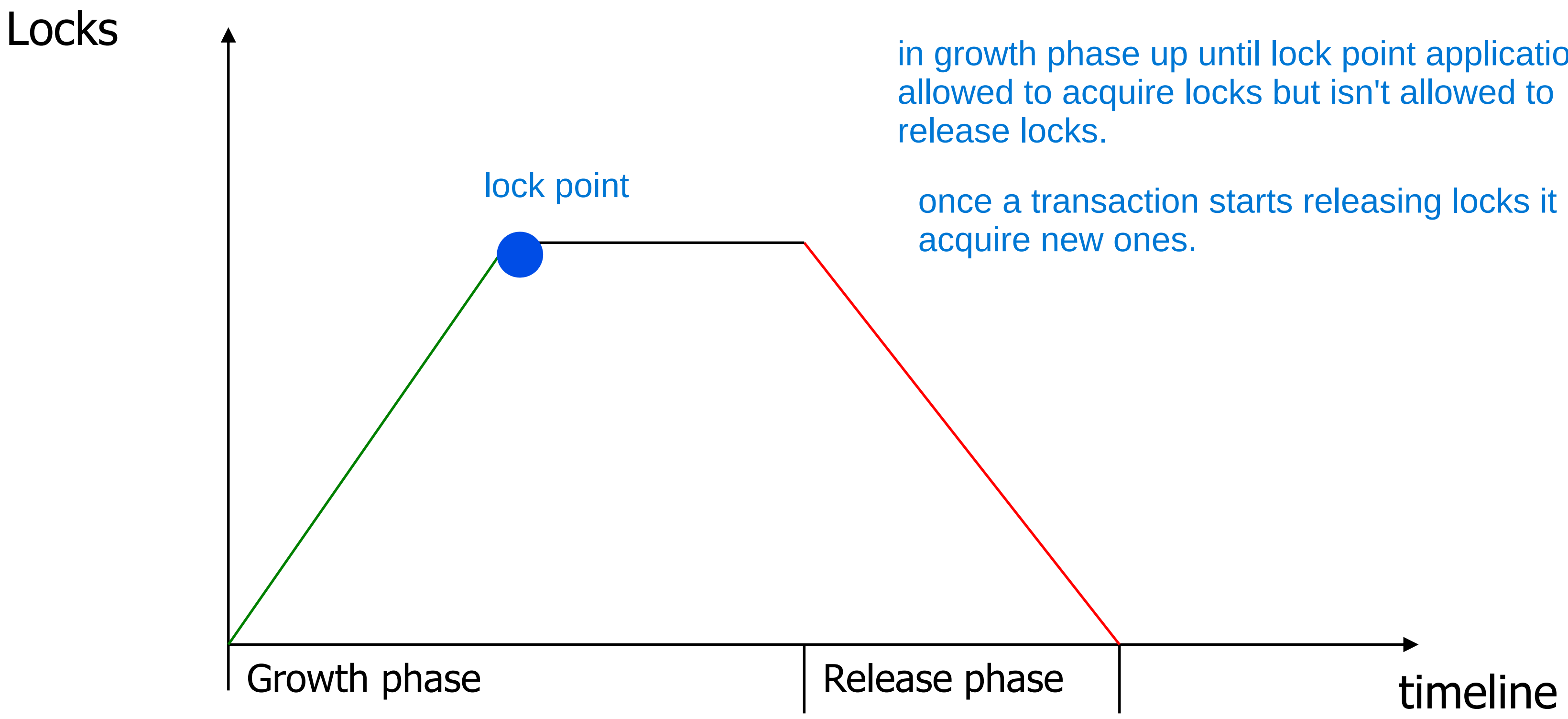
→ After starting the release phase (releasing the first lock) the transaction cannot acquire a new lock.

No LOCK after an UNLOCK.

once you do unlock you can't acquire another lock

eva.knirsch@kiu.edu.ge

2-phase-Lock (2PL)



in growth phase up until lock point application is allowed to acquire locks but isn't allowed to release locks.

once a transaction starts releasing locks it cannot acquire new ones.

2-phase-Lock (2PL)

- At the end of the growth phase, the "lock point", a transaction Ti holds all locks necessary.
- Any conflicting transactions
  - either need to wait
  - or are already in shrinking phase, releasing locks

It can be proved that, if every transaction in a schedule follows the two-phase locking protocol, the schedule is guaranteed to be serializable, obviating the need to test for serializability of schedules. The locking protocol, by enforcing two-phase locking rules, also enforces serializability. [El, page 787]

2PL Schedule

for simplicity S-locks are skipped

b is locked before a is unlocked which means that T2 has to wait

Operation	T1	T2
1	Begin	
2	x-lock (a)	
3	write(a) a:= a*1.1	Begin
4	x-lock(b)	
5	unlock(a)	lock(a)
6	write(b) b=b*1.1	
7		write(a) a=a+10
8	unlock(b)	lock(b)
9		unlock(a)
10	commit	write(b) b=b+10
11		
12		unlock(b)
13		commit
14		
15		

What problem do you still see?

if T1 doesn't commit but rolls back:
T2 already locked a and wrote a. but it wrote incorrect value. this means that T2 has to roll back as well

cascading rollback

to avoid this problem we have to release lock at the end of the transaction with commit.

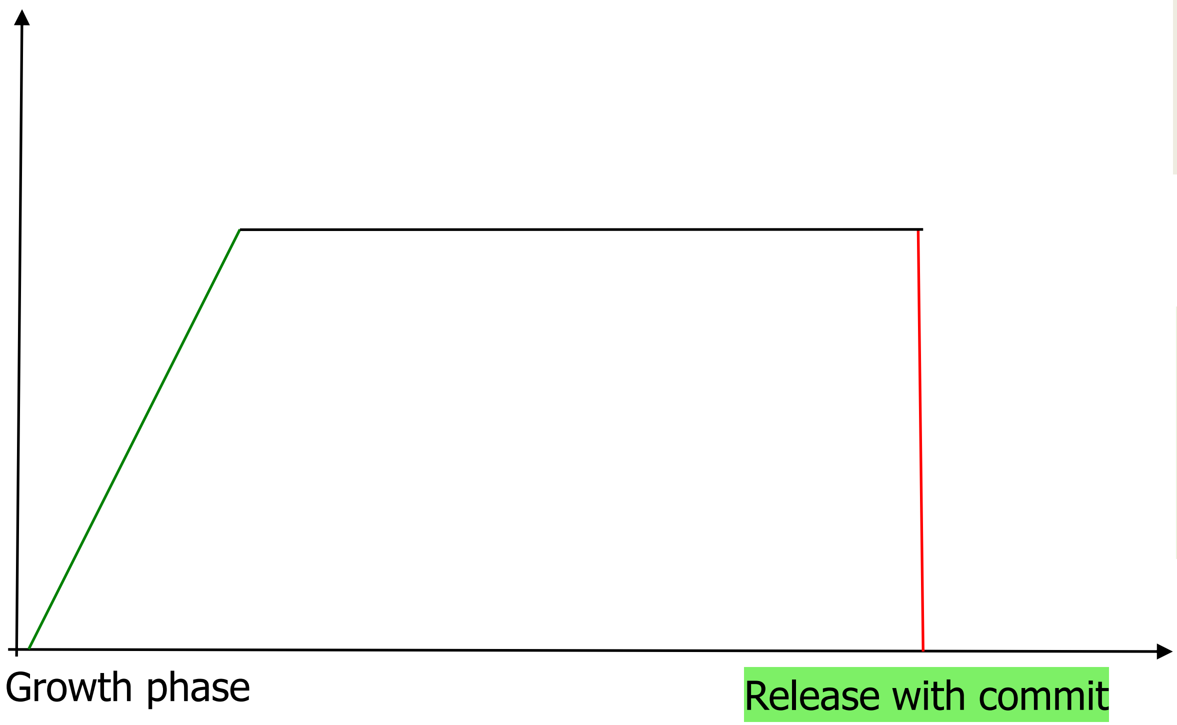
eva.knirsch@kiu.edu.ge

Strict 2PL

Locking Protocol either must handle cascading rollbacks (Commercial RDBMS do NOT do this) or must hold locks until commit.
→ locks must remain set until the end of the transaction.

→ 2PL is implemented as strict 2PL.

With a strict two-phase lock protocol, all locks are held until the end of the transaction and are released only at the end of the transaction (i.e., with the commit). This is also called commit with unlock.



PiostgreSQL does not use 2PL but holds its locks always until transaction end.

Let's take our airline ticket example again. T1 and T2 start directly one after the other. The DBMS applies the strict 2PL.

T1 can't unlock a because first it needs to lock b, similarly T2 can't unlock b because it first needs to lock a.

Operation	T1	T2
1	Begin	
2		Begin
3	x-Lock (a)	
4		x-Lock (b)
5	write(a) a:= a*1.1	
6		Write (b) b:= b+10
7		
8		

Result?

DEADLOCK

T1 waits for lock(b) and T2 waits for lock(a) and they will wait forever.

Frequent Problems

- Null-Values (Three-valued logic)
- Deadlocks
- Date-Time-Conversion with Time Zones
- Duplicate Data in Database, e.g. Customer is stored twice
- Coding, especially German Umlaut (München, M?nchen)

PostgreSQL Detecting Deadlocks

Deadlocks not only happen in 2PL but also in MVCC!

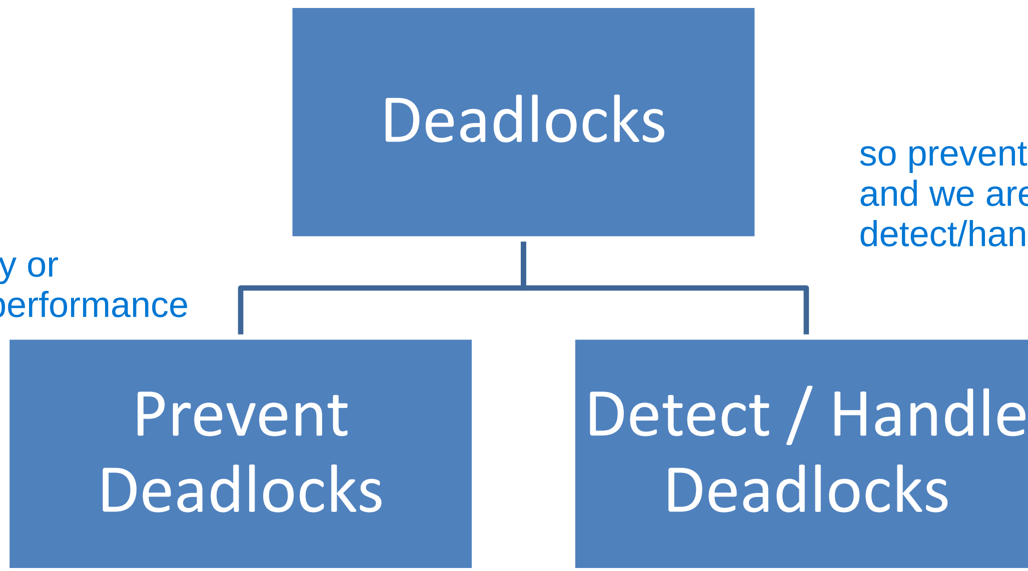
postgres:
ERROR: deadlock detected
DETAIL: Process 1456 waits for ShareLock on transaction 1700; blocked by process 8264.
Process 8264 waits for ShareLock on transaction 1701; blocked by process 10628.
Process 10628 waits for ShareLock on transaction 1702; blocked by process 1456.

Deadlocks

Transactions waiting for resources held by each other. Transactions block each other.

The order of execution causes the problem.

too costly or
limiting performance



so prevent deadlocks does not happen
and we are stuck with option 2: to
detect/handle deadlocks

Commercial databases – today - usually do not prevent deadlocks because the implementation of deadlock prevention algorithms / protocols is costly or limits performance. So, DBMS typiclly have methods to detect and break deadlocks, identifying an involved transaction as "victim" that will be rolled back.

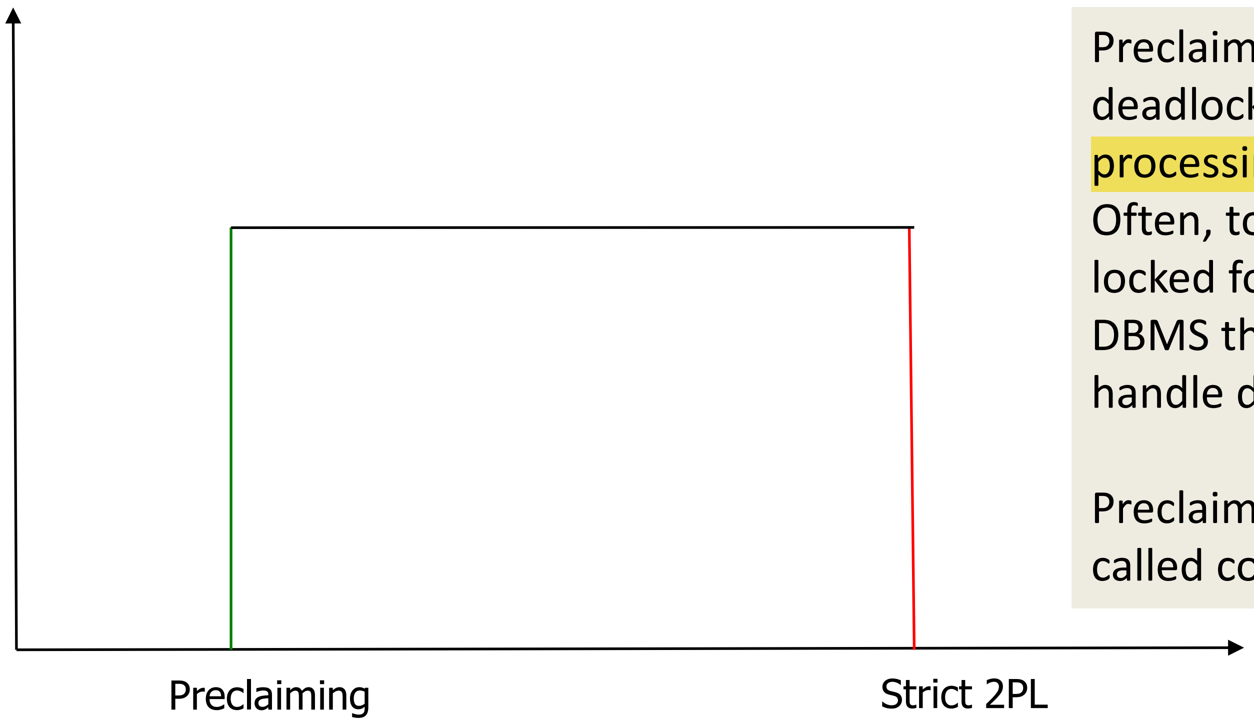
KUTAISI INTERNATIONAL UNIVERSITY

Preventing Deadlocks: Preclaiming

Deadlocks can be avoided by using a Preclaiming 2PL protocol: all locks that a transaction (may) request are set immediately at start of transaction

--> it cannot run into a deadlock.

A combination of preclaiming and strict 2PL would, of course, be possible.



Preclaiming prevents deadlocks, but it limits parallel processing considerably: Often, too many objects are locked for a long time. DBMS therefore prefer to handle deadlocks differently.

Preclaiming is sometimes also called conservative 2PL

Preventing Deadlocks: Timestamp methods

Timestamp methods: **Wait/Die and Wound/Wait methods**

Each transaction receives a timestamp when it is started. The timestamps are only needed in case of a deadlock.

The most recently started (= youngest) transaction has the highest timestamp value
The oldest, still running transaction has the smallest timestamp value.

Timestamp T1 < Timestamp T2

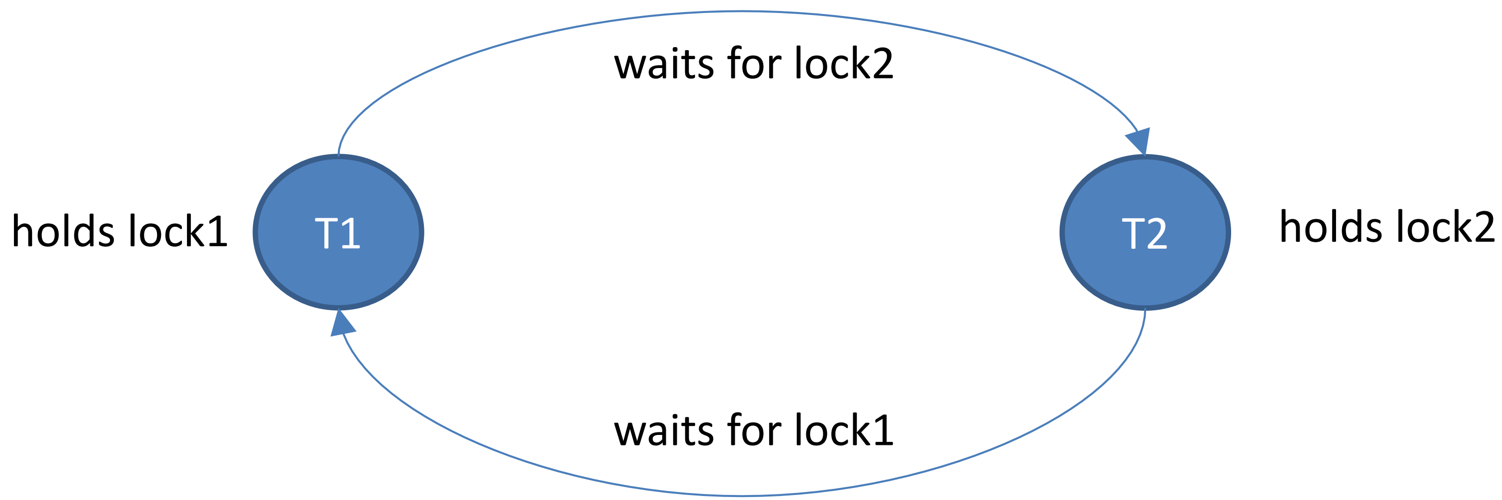
In case of Wait/Die and Wound/Wait methods, deadlocks are prevented by the DBMS comparing timestamp values with each other.

Detecting Deadlocks

Waits-for Graph:

Deadlocks can be detected using a waits-for graph:

- Each active transaction is represented by a node
- When transaction t1 requests an object that is locked by t2, then a directed edge (waits for) is drawn from t1 to t2.
- a deadlock exists if the waits-for graph contains a cycle.
- Many commercial databases use waits-for algorithms to detect deadlocks (PostgreSQL / MySQL) postgres even tells us cycles like on slide 16, while MySQL doesn't



eva.knirsch@kiu.edu.ge

PostgreSQL Detecting / Handling Deadlocks

Deadlock-Timeout (integer) – Waits-for-Graph Detection

- The Waits-for-Graph Detection is is computationally expensive.
- If transactions do not make any progress and locks are not released, PostgreSQL first waits for a certain amount of time (Deadlock-Timeout (integer))
- The Deadlock-Timeout default is one second (1s). If the lock is not released after the Deadlock-Timeout interval, PostgreSQ triggers the deadlock detection algorithm (Waits-for-Graph-Algorithm). then it rolls back after one or two transactions
- PostgreSQL optimistically assumes that deadlocks are not common in production applications and just waits on the lock for a while before checking for a deadlock.

eva.knirsch@kiu.edu.ge

KUTAISI INTERNATIONAL UNIVERSITY

Dealing with Deadlocks from the application side

- If deadlocls occur often, collect more debugging information and optimize – if possible – application.
- Application needs to be prepared to re-issue a transaction if it fails due to deadlock.
- Make sure that locks are released fast

  shorter the transaction more possibility that it runs through

  → Keep transactions small and short in duration

  → Limit use of explicit locking, e.g. select ... for update rather do serializable level
- Commit transactions immediately after making a set of related changes to make them less prone to collision.
- In particular, do not leave an interactive session open for a long time with an uncommitted transaction.

https://dev.mysql.com/doc/refman/5.7/en/innodb-deadlocks-handling.html

eva.knirsch@kiu.edu.ge