Transaction Management

陆伟

College of Software

Database Systems-Design and Application

May 19, 2021

Northwestern Polytechnical University

Outline

- Introduction
- Transactions and Executing Environment
- Potential Problems
- Transaction Model
- Concurrency Control
- Programming with Transactions
- Database Recovery
- Advanced Transaction Models

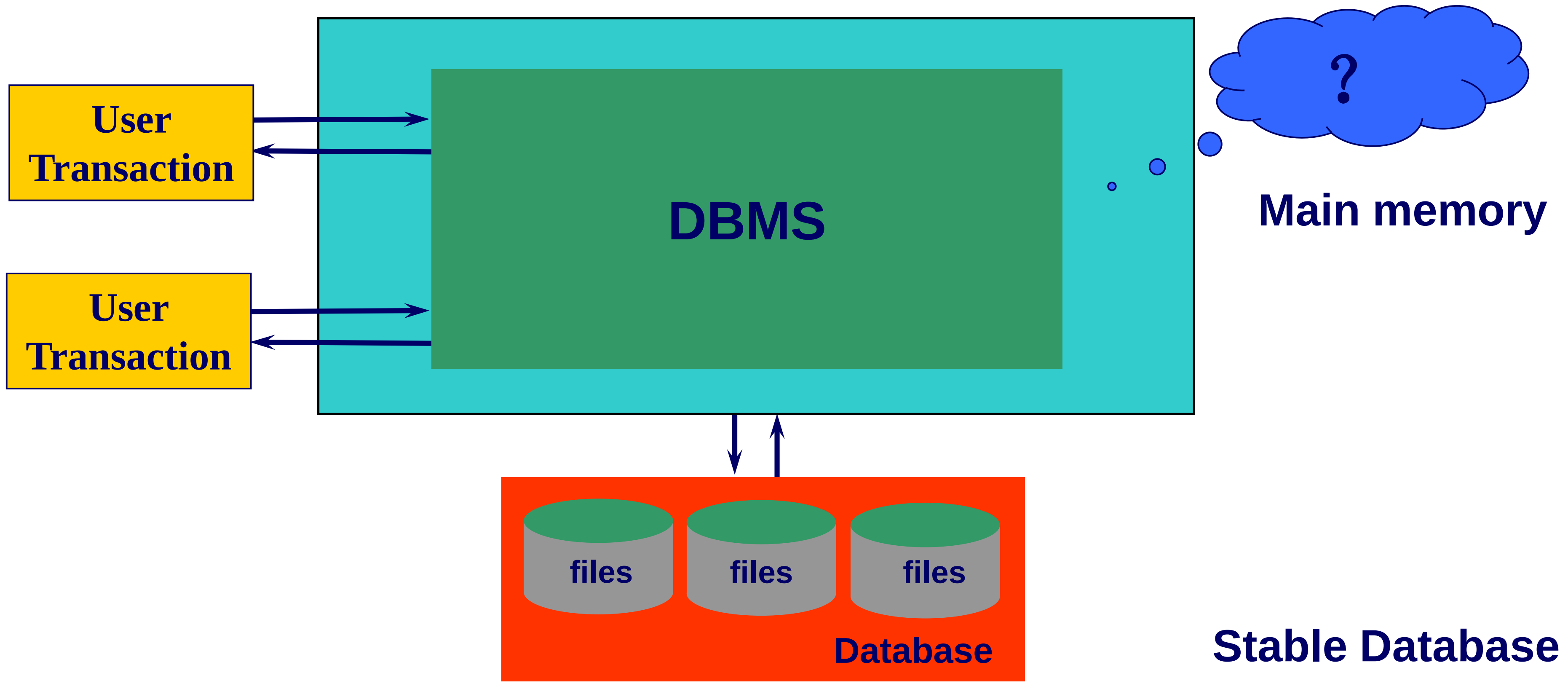
Introduction

- Two core concepts in database system:
  - **Data Model** — data abstraction
  - **Transaction Model** — activity abstraction
- Three characteristics of DBMS we discussed
  - Data abstraction
  - Reliability
  - Efficiency
- Data abstraction is supported by **data model** and the other two characteristics are supported by **transaction model**.

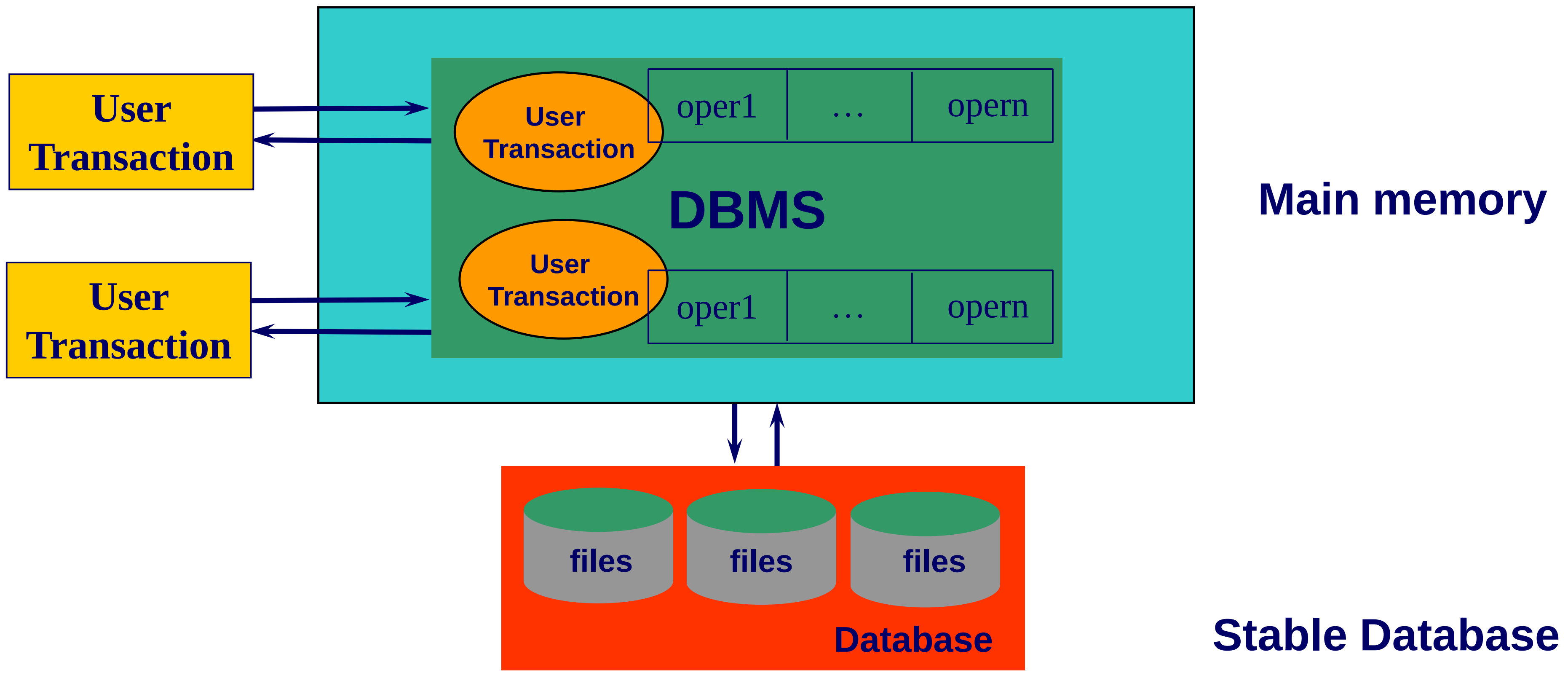
Transactions and Executing Environment

- Definition of transaction
  - A transaction is the execution of a program segment that performs some function or task by accessing a shared database.
  - An action, or series of actions, carried out by a single user or application program, which reads or updates the contents of the database.
  - A transaction is a logical unit of work on the database.
  - Examples:
    - Transfer money
    - Reserve a seat on a flight or train

Transactions and Executing Environment



Potential Problems



Potential Problems

- For single transaction
  - How to do when only partial actions in a transaction were executed?
  - How to do when failures happen before committed transactions finished?
- For multiple transactions
  - How to do when multiple transactions access the same data concurrently?
- How to resolve the problems?

Transaction Model

- DBMS support transaction model to provide application programmers with a high-level execution interface that hides both the effects of concurrency among the different transactions, and the presence of failures.
- In this way, programmers are relieved from dealing with the complexity of concurrent programming and failures. They need only to focus on designing the business and application logic, and developing correct individual transactions.

Transaction Model

- ACID Properties of Transactions
  - DBMS support transaction model which provide QoS (Quality of Service) guarantees of data consistency and database integrity—despite system failures and concurrent data sharing.
  - The above properties and guarantees with regard to transactions are commonly referred to as the ACID properties: Atomicity, Consistency, Isolation, and Durability.

Transaction Model

- Atomicity(原子性)
  - Atomicity requires that either "all or none" of the transaction's operations be performed.
  - All the operations of a transaction are treated as a single, indivisible, atomic unit.
- Consistency(一致性)
  - Consistency requires that a transaction maintain the integrity constraints on the database.
  - Transactions are assumed to be correct and are treated as the unit of consistency

Transaction Model

- Isolation(隔离性)
  - Isolation requires that a transaction execute without any interference from other concurrent transactions.
  - Transactions are assumed to be independent.
- Durability(持久性)
  - Durability requires that all the changes made by a committed transaction become permanent in the database, surviving any subsequent failures.

Transaction Model

- How DBMS supports the ACID properties of transactions?
  - Concurrency control protocols, ensures the isolation property;
  - Recovery protocols, ensures atomicity and durability properties;
  - Triggering mechanisms, enforces the integrity constraints on a database.

Concurrency Control

- The need for concurrency control
- Schedule of transactions
- Two-phase locking(2PL) protocol
- Problems for 2PL
- About dead lock
- Multi-version
- ANSI SQL2 isolation levels
- Advanced Transaction Model

The need for concurrency control

- The lost update problem

Timer	T1	T2	balx
t1		begin_transaction	100
t2	begin_transaction	read(balx)	100
t3	read(balx)	balx=balx+100	100
t4	balx=balx-10	write(balx)	200
t5	write(balx)	commit	90
t6	commit		90

The need for concurrency control

- The uncommitted dependency (or dirty read) problem

Timer	T3	T4	balx
t1		begin_transaction	100
t2		read(balx)	100
t3		balx=balx+100	100
t4	begin_transaction	write(balx)	200
t5	read(balx)	⋮	200
t6	balx=balx-10	rollback	100
t7	write(balx)		190
t8	commit		190

The need for concurrency control

- Nonrepeatable read (fuzzy) problem

Timer	T5	T6	balx	baly	balz
t1		begin_transaction	100	0	0
t2	begin_transaction	read(balx)	100	0	0
t3	read(balx)	balx=balx+100	100	0	0
t4	baly=baly+balx	write(balx)	200	0	0
t5	write(baly)	commit	200	100	0
t6	read(balx)		200	100	0
t7	balz=balz+balx		200	100	0
t8	write(balz)		200	100	200
t6	commit		200	100	200

The need for concurrency control

- The inconsistent analysis problem - a situation of nonrepeatable read

Timer	T7	T8	balx	baly	balz	sum
t1		begin_transaction	100	50	25	
t2	begin_transaction	sum=0	100	50	25	0
t3	read(balx)	read(balx)	100	50	25	0
t4	balx=balx-10	sum=sum+balx	100	50	25	100
t5	write(balx)	read(baly)	90	50	25	100
t6	read(balz)	sum=sum+baly	90	50	25	150
t7	balz=balz+10		90	50	25	150
t8	Write(balz)		90	50	35	150
t9	commit	read(balz)	90	50	35	150
t10		sum=sum+balz	90	50	35	185
t11		commit	90	50	35	185

The need for concurrency control

- Conclusion
  - Every transaction is correct itself.
  - The interleaving of operation may produce problems.
- Discuss and try to resolve the problems
- One obvious solution is to allow only one transaction to execute at a time: one transaction is committed before the next transaction is allowed to begin.
- Is it necessary?

The need for concurrency control

T_0	T_1
Begin transaction;	Begin transaction;
read(a);	
a:=a-10;	
write(a);	
	read(a);
read(b);	a:=a-1;
b:=b+10;	write(a);
write(b);	
commit;	read(b);
	b=:b+1;
	write(b);
	commit;

Schedule of transactions

- Schedule (调度)
  - A sequence of the operations by a set of concurrent transactions that preserves the order of the operations in each of the individual transactions.
- Serial schedule (串行调度)
  - A schedule where the operations of each transaction are executed consecutively (连续) without any interleaved operations from other transactions.
- Nonserial/concurrent schedule
- Examples

Schedule of transactions

- Can all concurrent schedules give the wrong result?
- Example Analysis

Schedule of transactions

Initial value: a=20,b=30

串行调度

T_0	T_1
read(a);	
a:=a-10;	
write(a);	
read(b);	
b:=b+10;	
write(b);	
	read(a);
	tmp:=a/10;
	a:=a-tmp;
	write(a);
	read(b);
	b=:b+tmp;
	write(b);

①

T_0	T_1
	read(a);
	tmp:=a/10;
	a:=a-tmp;
	write(a);
	read(b);
	b=:b+tmp;
	write(b);
read(a);	
a:=a-10;	
write(a);	
read(b);	
b:=b+10;	
write(b);	

②

并发调度

T_0	T_1
read(a);	
a:=a-10;	
write(a);	
	read(a);
	tmp:=a/10;
	a:=a-tmp;
	write(a);
read(b);	
b:=b+10;	
write(b);	
	read(b);
	b=:b+tmp;
	write(b);

③

T_0	T_1
read(a);	
a:=a-10;	
	read(a);
	tmp:=a/10;
	a:=a-tmp;
	write(a);
	read(b);
write(a);	
read(b);	
b:=b+10;	
write(b);	
	b=:b+tmp;
	write(b);

④

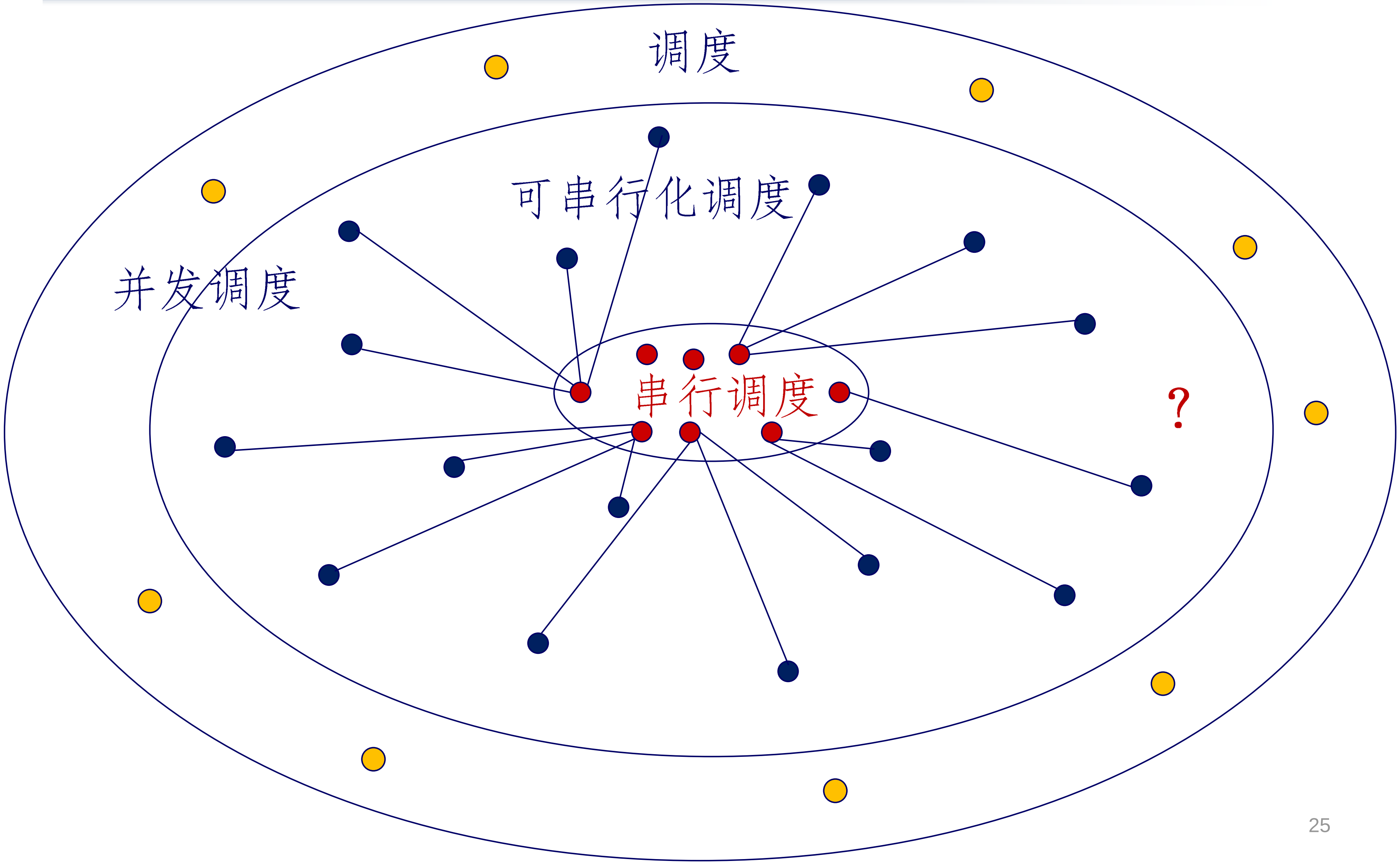
Schedule of transactions

①result	②result	③result	④result
a=9，v=41	a=8，b=42	a=9，b=41	a=10，b=32

Schedule of transactions

- Serial schedule never leaves the database in an inconsistent state, so every serial execution is considered correct, although different results may be produced.
- Serializable schedule
  - If a set of transactions executes concurrently, we say the (nonserial) schedule is serializable (or correct) schedule if it produces the same results as some serial schedule.

Schedule of transactions



Schedule of transactions

- How to test the serializability for a schedule?

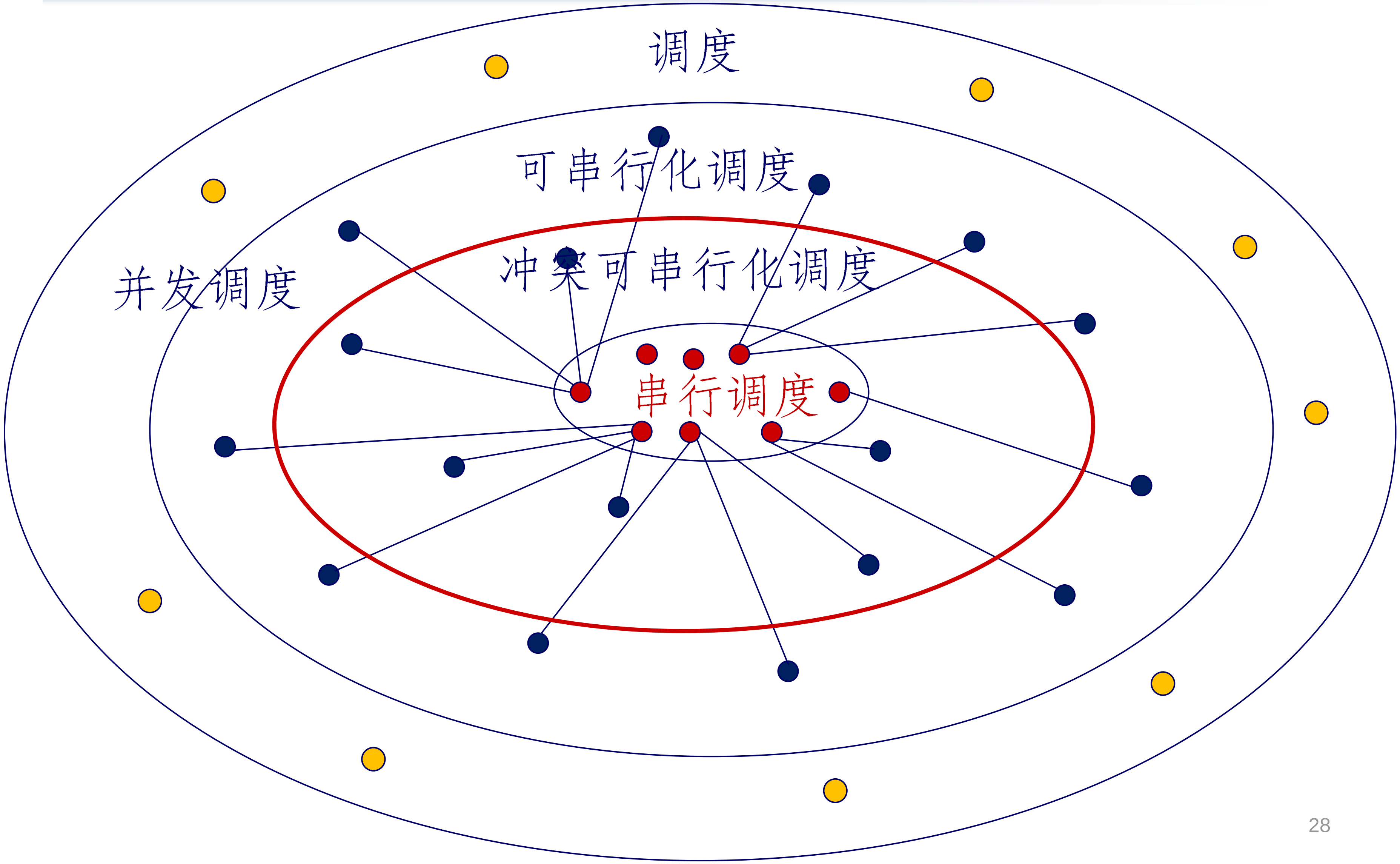
T_0	T_1
read(a);	
a:=a-0;	
	read(a);
	tmp:=a mod10;
	a:=a-tmp;
	write(a);
	read(b);
write(a);	
read(b);	
b:=b+0;	
write(b);	
	b=:b+tmp;
	write(b);

a=10/11？

Schedule of transactions

- Conflict serializable
  - 若调度S可以通过交换某些相继非冲突操作变成某一串行调度，则称角度S为冲突可串行化调度。
- Conflict operations on the same data item:
  - Read-write
  - Write-read
  - Write-write

Schedule of transactions



Schedule of transactions

T	Ti	Tj	Ti	Tj	Ti	Tj
t1	beg_tran		beg_tran		beg_tran	
t2	read(balx)		read(balx)		read(balx)	
t3	write(balx)		write(balx)		write(balx)	
t4		beg_tran		beg_tran	read(baly)	
t5		read(balx)		read(balx)	write(baly)	
t6		write(balx)	read(baly)		commit	
t7	read(baly)			write(balx)		beg_tran
t8	write(baly)		write(baly)			read(balx)
t9	commit		commit			write(balx)
t10		read(baly)		read(baly)		read(baly)
t11		write(baly)		write(baly)		write(baly)
t12		commit		commit		commit

a b c

Schedule of transactions

- Example

	T_0	T_1
1	read(a);	
2	write(a);	
3		read(b);
4		write(b);
5	read(b);	
6	write(b).	
7		read(a);
8		write(a).

非冲突可串行化调度

可串行化调度

Schedule of transactions

- Example

T_0	T_1
read(a);	
a:=a-10;	
write(a);	
	read(a);
	a:=a-1;
	write(a);
read(b);	
b:=b+10;	
write(b);	
	read(b);
	b=:b+1;
	write(b);

冲突可串行化调度

Schedule of transactions

- Test the conflict serializability for a schedule

该例调度是可串行化调度，但不是冲突可串行化调度。其结果和两个串行调度(T_0,T_1)、(T_1,T_0)的结果一样，都是：帐号A把1元钱转到帐号B。
例如，对初值a=20和b=10，调度的处理结果都是a=19,b=11。
可串行性却需要分析所有操作，不存在有效测试算法，
而冲突可串行仅需分析read和write语句，故存在有效测试算法。

T0	T1
read(a);	
a:=a-2;	
write(a);	
	read(b);
	b:=b-1;
	write(b);
read(b);	
b:=b+2;	
write(b);	
	read(a);
	a:=a+1;
	write(a);

Schedule of transactions

- Test for conflict serializability

前趋图(precedence graph)概念G=(V,E)
点集V：每个点表示一个事务；
边集E：每个边(Ti ,Tj)表示访问同一个数据项Q的两个事务Ti ,Tj ，要求满足下列三条件之一(冲突方向)：

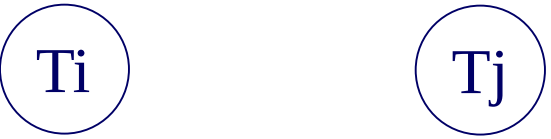
(1) Ti写Q后，Tj读Q；
(2) Ti读Q后，Tj写Q；
(3) Ti写Q后，Tj写Q。

前趋图无回路⇔冲突方向是单一的
⇔调度是冲突可串行化的

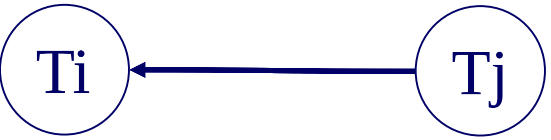
N个事务调度前趋图的回路测试法需要时间为$O(N^2)$

Schedule of transactions

- Examples



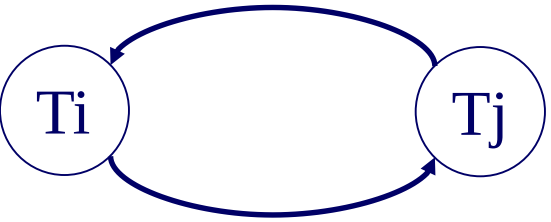
Timer	Ti	Tj
t1	begin_transaction	
t2	read(x)	begin_transaction
t3		read(y)
t4	read(y)	
t5	commit	
t6		commit


$$\begin{tikzcd} Ti & Tj \arrow[l] \end{tikzcd}$$

Timer	Ti	Tj
t1	begin_transaction	
t2	read(x)	begin_transaction
t3		read(y)
t4	write(y)	
t5	commit	
t6		commit

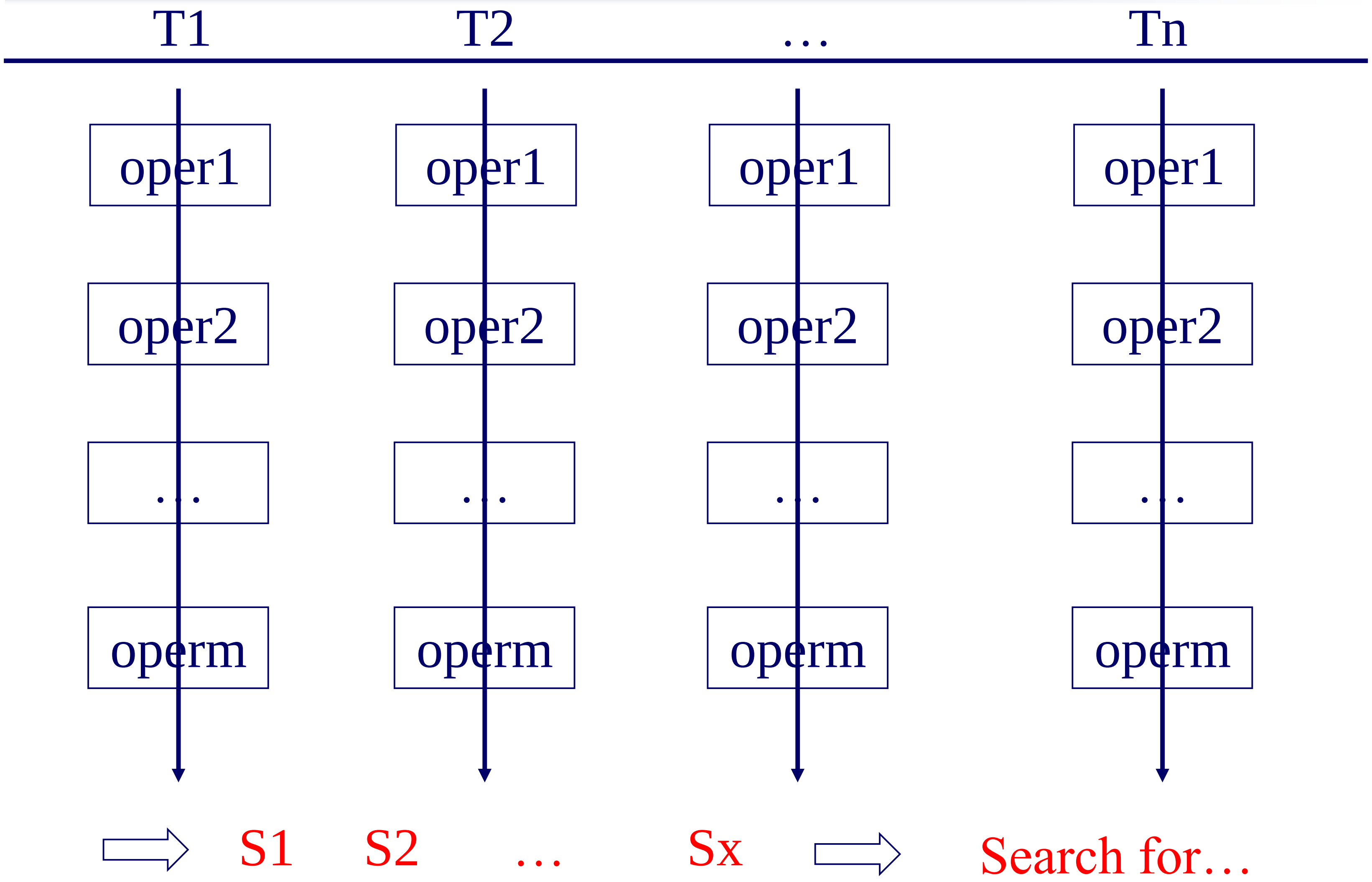
Schedule of transactions

- Examples



Timer	Ti	Tj
t1	begin_transaction	
t2	read(balx)	
t3	balx=balx+100	
t4	write(balx)	begin_transaction
t5		read(balx)
t6		balx=balx*1.1
t7		write(balx)
t8		read(baly)
t9		baly+baly*1.1
t10		write(baly)
t11	read(baly)	commit
t12	baly=baly-100	
t13	write(baly)	
t14	commit	

Two-phase locking(2PL) protocol



Two-phase locking(2PL) protocol

- In practice, a DBMS does not test for the serializability of a schedule. This would be impractical, as the interleaving of operations from concurrent transactions is determined by the operating system.
- Instead, the approach taken is to use protocols that are known to produce serializable schedules. -- Two-phase locking(2PL) protocol

Two-phase locking(2PL) protocol

- Locking
  - A procedure used to control concurrent access to data.
- Shared lock
  - If a transaction T has a shared lock on a data item Q, it can read Q but not update it.
  - Represent it as LOCK_S(Q)
- Exclusive lock
  - If a transaction T has an exclusive lock on a data item Q, it can both read and update Q.
  - Represent it as LOCK_X(Q)

Two-phase locking(2PL) protocol

- Locks are used in the following way:
  - 1)Any transaction that needs to access a data item must first lock the item.
  - 2)If the item is not already locked by another transaction, the lock will be granted.
  - 3)If the item is currently locked and the request lock is compatible with the existing lock, the request will be granted; otherwise, the transaction must wait until the existing lock is released.
  - 4)A transaction contimues to hold a lock until it explicitly releases it. UNLOCK(Q)

Two-phase locking(2PL) protocol

- How to use locks in transactions to guarante serializability of schedule?

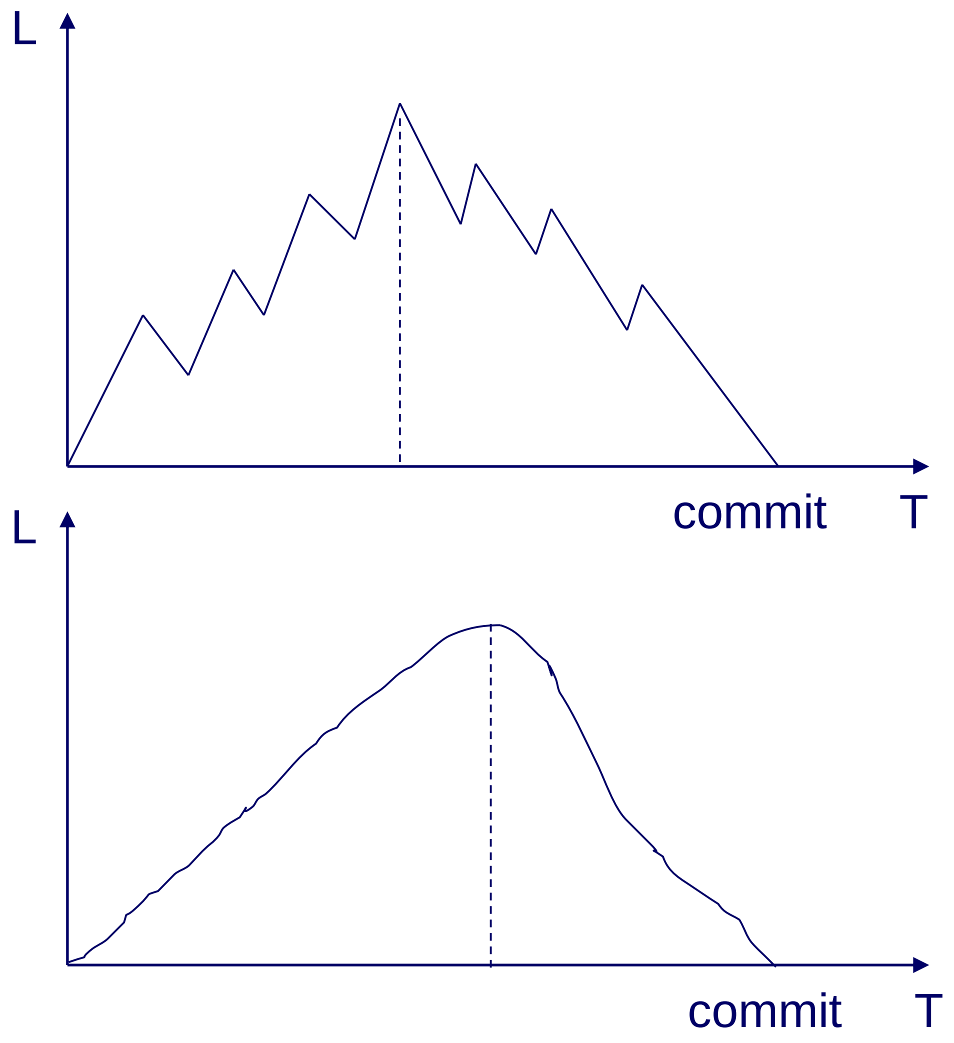
Ti	Tj
Read(B);	
B=B-50;	
Write(B);	
	Read(A);
	Read(B);
	Disp(A+B);
Read(A);	
A=A+50;	
Write(A);	

Ti	Tj
Lock_x(B);	
Read(B);	
B=B-50;	
Write(B);	
Unlock(B);	
	Lock_s(A);
	Read(A);
	Unlock(A);
	Lock_s(B);
	Read(B);
	Unlock(B);
	Disp(A+B);
Lock_x(A);	
Read(A);	
A=A+50;	
Write(A);	
Unlock(A);	

Two-phase locking(2PL) protocol

- Two-phase locking (2PL) protocol
  - A transaction follows the two-phase locking protocol if all locking operations precede the first unlock operation in the transaction.
- Two phases for transaction:
  - Growing phase - acquires all locks but cannot release any locks.
  - Shrinking phase - releases locks but cannot acquire any new locks.

Two-phase locking(2PL) protocol



general

2PL

Two-phase locking(2PL) protocol

- Effect of 2PL on schedule

Timer	T1	T2	balx
t1		begin_transaction	100
t2	begin_transaction	read(balx)	100
t3	read(balx)	balx=balx+100	100
t4	balx=balx-10	write(balx)	200
t5	write(balx)	commit	90
t6	commit		90

Two-phase locking(2PL) protocol

- Effect of 2PL on schedule

Timer	T1	T2	balx
t1		begin_transaction	100
t2	begin_transaction	Lock_x(balx)	100
t3	Lock_x(balx)	read(balx)	100
t4	WAIT	balx=balx+100	100
t5	WAIT	write(balx)	200
t6	WAIT	unlock(balx)	200
t7	read(balx)		200
t8	balx=balx-10	commit	200
t9	write(balx)		190
t10	commit/unlock(balx)		190

Two-phase locking(2PL) protocol

Timer	T7	T8	balx	baly	balz	sum
t1		begin_transaction	100	50	25	
t2	begin_transaction	sum=0	100	50	25	0
t3	Lock_x(balx)		100	50	25	0
t4	read(balx)	Lock_s(balx)	100	50	25	0
t5	balx=balx-10	WAIT	100	50	25	0
t6	write(balx)	WAIT	90	50	25	0
t7	Lock_x(balz)	WAIT	90	50	25	0
t8	read(balz)	WAIT	90	50	25	0
t9	balz=balz+10	WAIT	90	50	25	0
t10	write(balz)	WAIT	90	50	35	0
t11	unlock(balx,balx)	WAIT	90	50	35	0
t12		read(balx)	90	50	35	0
t13		sum=sum+balx	90	50	35	90
t14		Lock_s(baly)	90	50	35	90
t15		read(baly)	90	50	35	90
t16		sum=sum+baly	90	50	35	140
t17		Lock_s(balz)	90	50	35	140
t18	commit	read(balz)	90	50	35	140
t19		sum=sum+balz	90	50	35	175
t20		commit/unlock(balx,baly,balz)	90	50	35	175

Two-phase locking(2PL) protocol

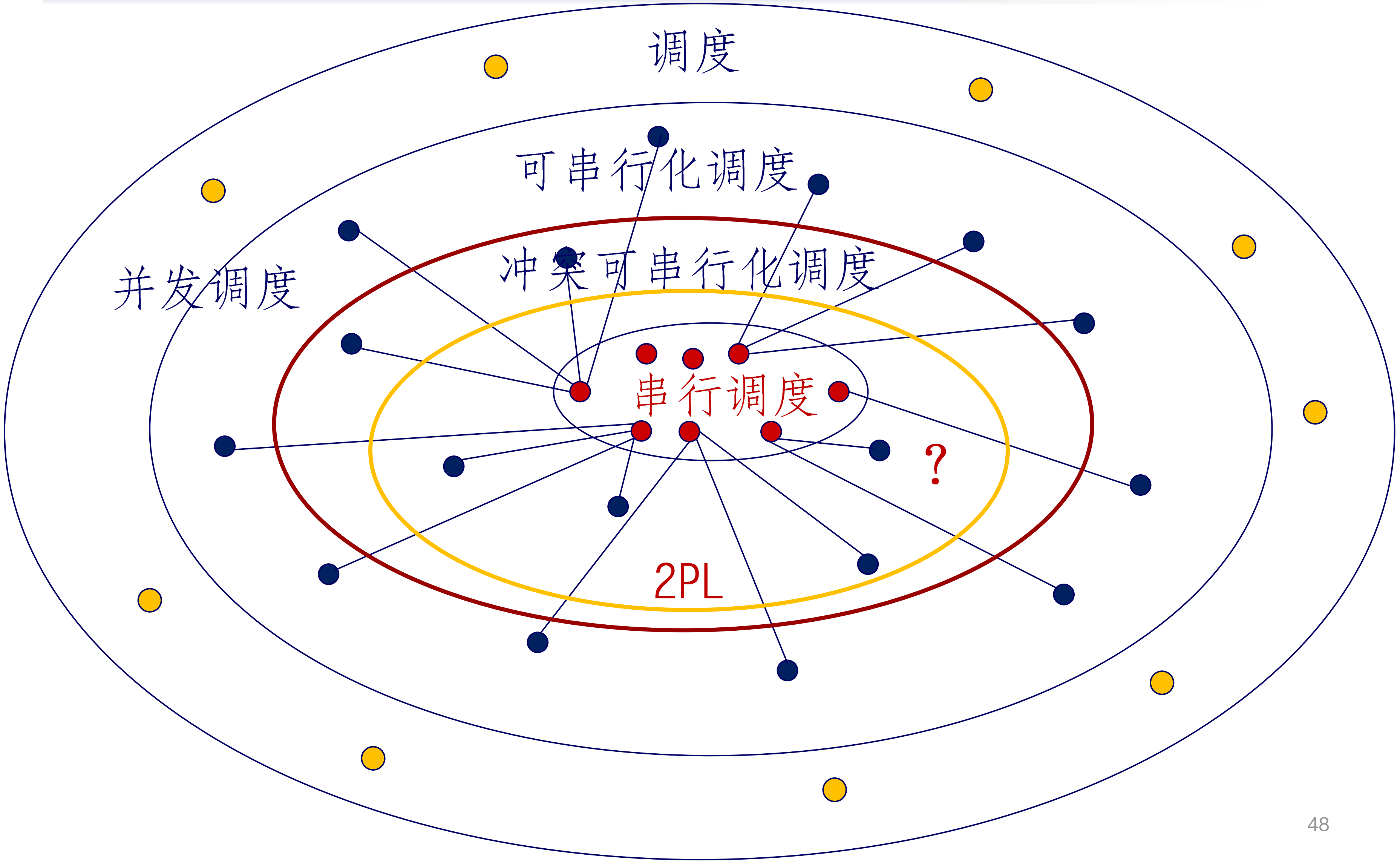
- Can 2PL resolve all problems in all conditions?

Timer	T3	T4	balx
t1		begin_transaction	100
t2		read(balx)	100
t3		balx=balx+100	100
t4	begin_transaction	write(balx)	200
t5	read(balx)	:	200
t6	balx=balx-10	rollback	100
t7	write(balx)		190
t8	commit		190

Two-phase locking(2PL) protocol

- It can be proved that if every transaction which will commit later in a schedule follows the two-phase locking protocol, then the schedule is guaranteed to be conflict serializable.
- Every transaction in a schedule following the two-phase locking protocol is a sufficient but not necessary condition for guaranting the schedule to be conflict serializable.

Schedule of transactions



Two-phase locking(2PL) protocol

- How can we predict the point at which no more locks will be needed?

- How about the schedule when the transactions in it rollback or failed?

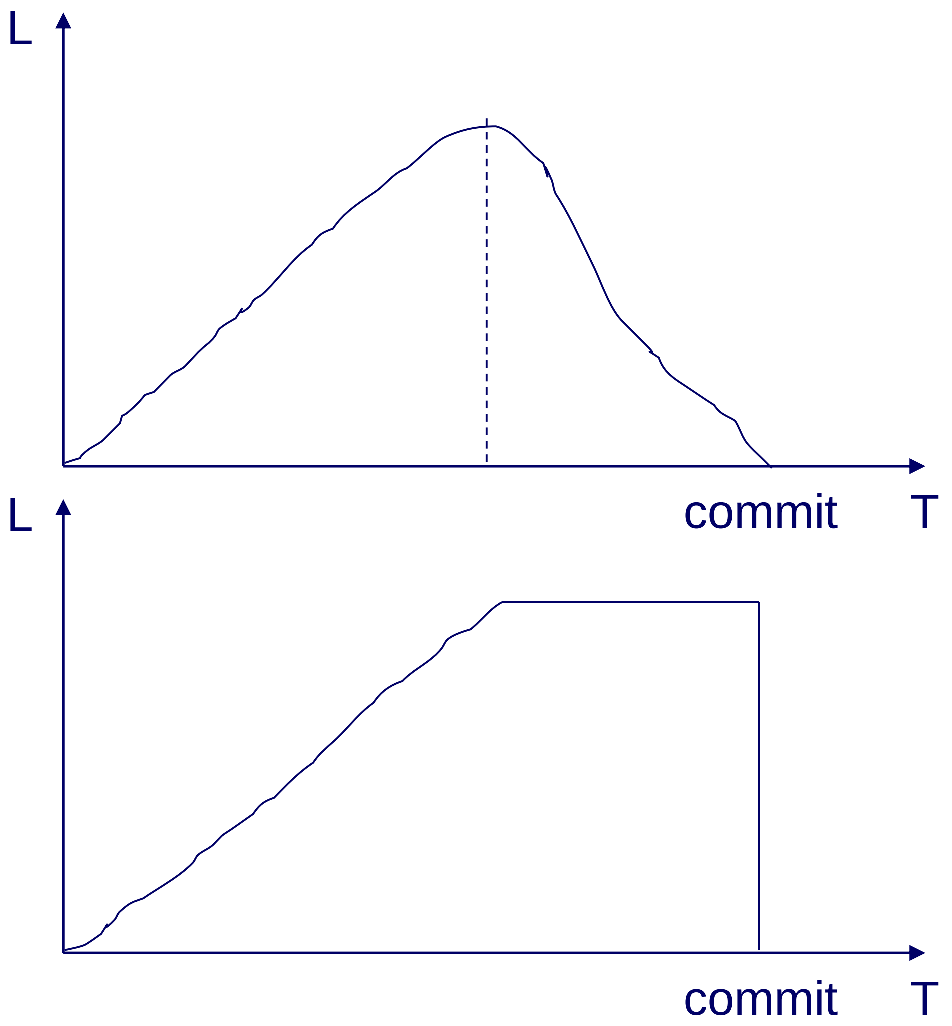
Two-phase locking(2PL) protocol

Timer	T0	T1	T2
t1	begin_transaction		
t2	write_lock(balx)		
t3	read(balx)		
t4	read_lock(baly)		
t5	read(baly)		
t6	balx=baly+balx		
t7	write(balx)		
t8	unlock(balx)	begin_transaction	
t9		write_lock(balx)	
t10		read(balx)	
t11		balx=balx+100	
t12		write(balx)	
t13		unlock(balx)	
t14		Commit/rollback	
t15	rollback		
t16			begin_transaction
t17			Read_lock(balx)
t18			
t19			Commit/rollback

Two-phase locking(2PL) protocol

- Rigorous 2PL protocol
  - The rigorous 2PL is one in which transactions request locks just before they operate on a data item and their growing phase ends just before they are committed.
- With rigorous 2PL, transactions can be serialized in the order in which they commit.
- Rigorous 2PL can guarantee a schedual to be revoverable.

Two-phase locking(2PL) protocol



2PL

R2PL

Two-phase locking(2PL) protocol

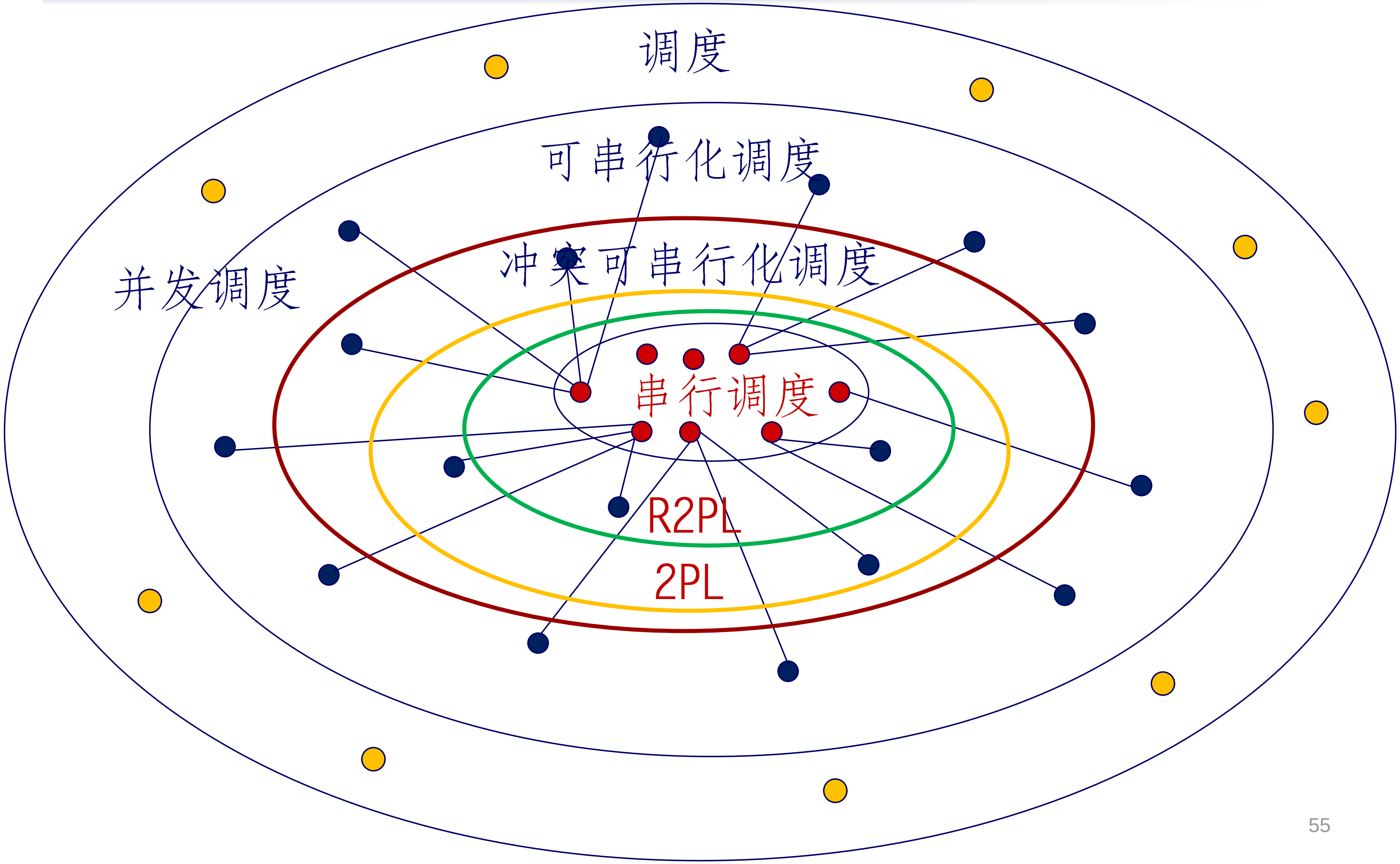
Timer	T0	T1	T2
t1	begin_transaction		
t2	write_lock(balx)		
t3	read(balx)		
t4	read_lock(baly)		
t5	read(baly)		
t6	balx=baly+balx		
t7	write(balx)		
t8		begin_transaction	
t9		write_lock(balx)	
t10		wait	
t11		wait	
t12		wait	
t13	rollback /unlock(balx,baly)	wait	
t14		read(balx)	
t15		balx=balx+100	
t16		write(balx)	begin_transaction
t17		Commit/unlock(balx)	Read_lock(balx)
t18			

Two-phase locking(2PL) protocol

- Strict 2PL
  - The strict 2PL is one in which transactions request locks just before they operate on a data item and they hold exclusive locks until the end of the transaction..

- Rigorous and strict 2PL is easier to implement than other 2PL variants.
- Most database systems implement one of these two variants of 2PL.

Schedule of transactions



Problems for 2PL

- Dead lock may occur
- Concurrency descend

About dead lock

- Deadlock
  - An impasse(僵局) that may result when two (or more) transactions are each waiting for locks to be released that are held by the other.

About dead lock

- Example

Timer	Ta	Tb
t1	begin_transaction	
t2	write_lock(balx)	begin_transaction
t3	read(balx)	write_lock(baly)
t4	balx=balx-10	read(baly)
t5	write(balx)	baly=baly+100
t6	write_lock(baly)	write(baly)
t7	WAIT	write_lock(balx)
t8	WAIT	WAIT
t9	WAIT	WAIT
t10	:	:

About dead lock

- Deadlock detection
  - Wait-for graph (WFG) --等待图

WFG is a directed graph G=(N,E):
1)create a node for each transaction.
2)create a directed edge Ti→Tj, if transaction Ti is waiting to lock an item that is currently locked by Ti.
3)Deadlock exists if and only if the WFG contains a cycle.

About dead lock

- Frequency of deadlock detection
  - Generates and examine the WFG at regular intervals.
  - Dynamic deadlock detection algorithm
- Recovery from deadlock detection
  - Abort one or more of the transactions.
  - Which transaction(s) will be aborted?

About dead lock

- Deadlock prevention
  - Timeouts
  - Wait-Die algorithm
  - Wound-Wait algorithm

Multi-version

- Advanced Database Systems

ANSI SQL2 isolation levels

```
SET TRANSACTION  READ ONLY | READ
WRITE
  [ISOLATION LEVEL  READ UNCOMMITTED|
                   READ COMMIT|
                   REPEATABLE READ|
                   SERIALIZABLE]
```

Programming with Transactions

- An application developer must keep the following factors in mind when programming with transactions:
  - Care must be taken to ensure the consistency of program variables.
  - Transaction boundaries must be set properly to allow maximum concurrency and semantic correctness.
  - The correct isolation level must be set to allow maximum concurrency and semantic correctness.

Programming with Transactions

- Consistency of Program Variables

```
$amount$ = 0
Begin Transaction
For Each Bill
        1)Transfer money from payer to payee (transferring money will
     deduct from the payer's account balance and increment the payee's
     account balance).
        2)Increment amount transferred from the payer to the payee.
Commit if success, roll back otherwise
Display the amount deducted from payer's checking account.
```

Programming with Transactions

- If the transaction to pay the bills fails, the cumulative *amount* variable must be set back to zero so that the correct amount is displayed at the end.
- The amount variable should also be set to zero after the transaction begins because a retry of the transaction will need to have *amount*=0.

Programming with Transactions

- Transaction Boundaries
  - Consider a Web banking system. Suppose two users are sharing a checking account and they simultaneously initiate transactions on the account.

Programming with Transactions

1. User logs into the system.
2. DBMS starts a transaction.
3. The system waits for user's choice of action.
4. User chooses to withdraw money from checking.
5. DBMS locks checking account.
6. The system waits for user to enter withdrawal amount.
7. User enters amount.
8. The system issues command to DBMS to update the account balance.
9. The system waits for user to choose new action or quit.
10. User chooses to quit.
11. DBMS commits transaction and unlocks the checking account.

Programming with Transactions

- Transaction Boundaries
  - In general, you should set transaction boundaries in an application so that only operations on the database are part of the transaction

Programming with Transactions

1. User logs into the system.
2. The system waits for user's choice of action.
3. User chooses to withdraw from checking.
4. The system waits for user to enter withdrawal amount.
5. User enters amount.
6. DBMS starts transaction and locks checking account.
7. The system issues command to DBMS to update the account balance.
8. DBMS ends transaction and unlocks the checking account.
9. ATM waits for user to choose new action or quit.
10. User chooses to quit.

Programming with Transactions

- Isolation Level
  - Setting improper isolation levels for transactions may limit the concurrency of all transactions in the database as in-progress transactions hold unnecessary locks that others need to proceed.
  - Improper isolation levels may also lead to an application retrieving inconsistent or "dirty" data written by updates from other transactions.
- Example of Isolation Level – refer to exercise 9

Programming with Transactions

- Transactions in PostgreSQL
  - PostgreSQL, by default, operates in unchained mode, also known as autocommit mode.
  - The begin transaction statement begins a chained mode transaction in PostgreSQL.
- PostgreSQL supports two transaction isolation levels:
  - read committed (default isolation)
  - serializable.

Programming with Transactions

SELECT * FROM librarian;
INSERT INTO librarian VALUES('Mary');

BEGIN TRANSACTION (*implicit*)
SELECT * FROM librarian;
commit (*implicit*)
BEGIN TRANSACTION (*implicit*)
 INSERT INTO librarian VALUES('Mary');
COMMIT(*implicit*)

Programming with Transactions

```
BEGIN TRANSACTION; (explicit)
  SELECT * FROM librarian;
  INSERT INTO librarian VALUES('Mary');
COMMIT; (explicit)
```

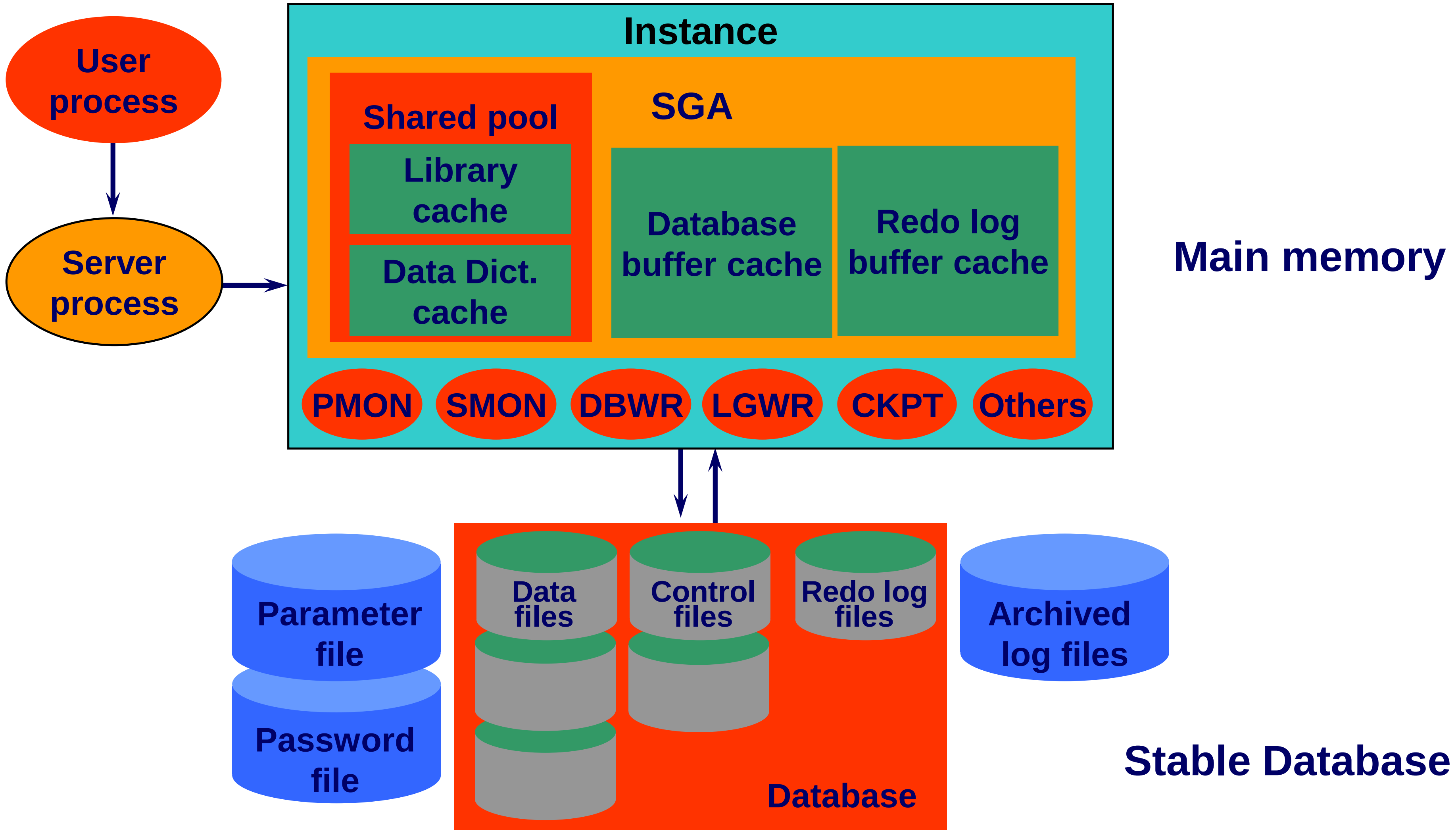
Advanced Transaction Model

- MIT6.824: Distributed Systems

Database Recovery

- The DBMS Executing Environment and Potential Failures
- Warehouse Example
- How to do when Failures Happened
- Log
- The Process of Recovery
- About Checkpoint
- Recovery from non-volatile storage failures

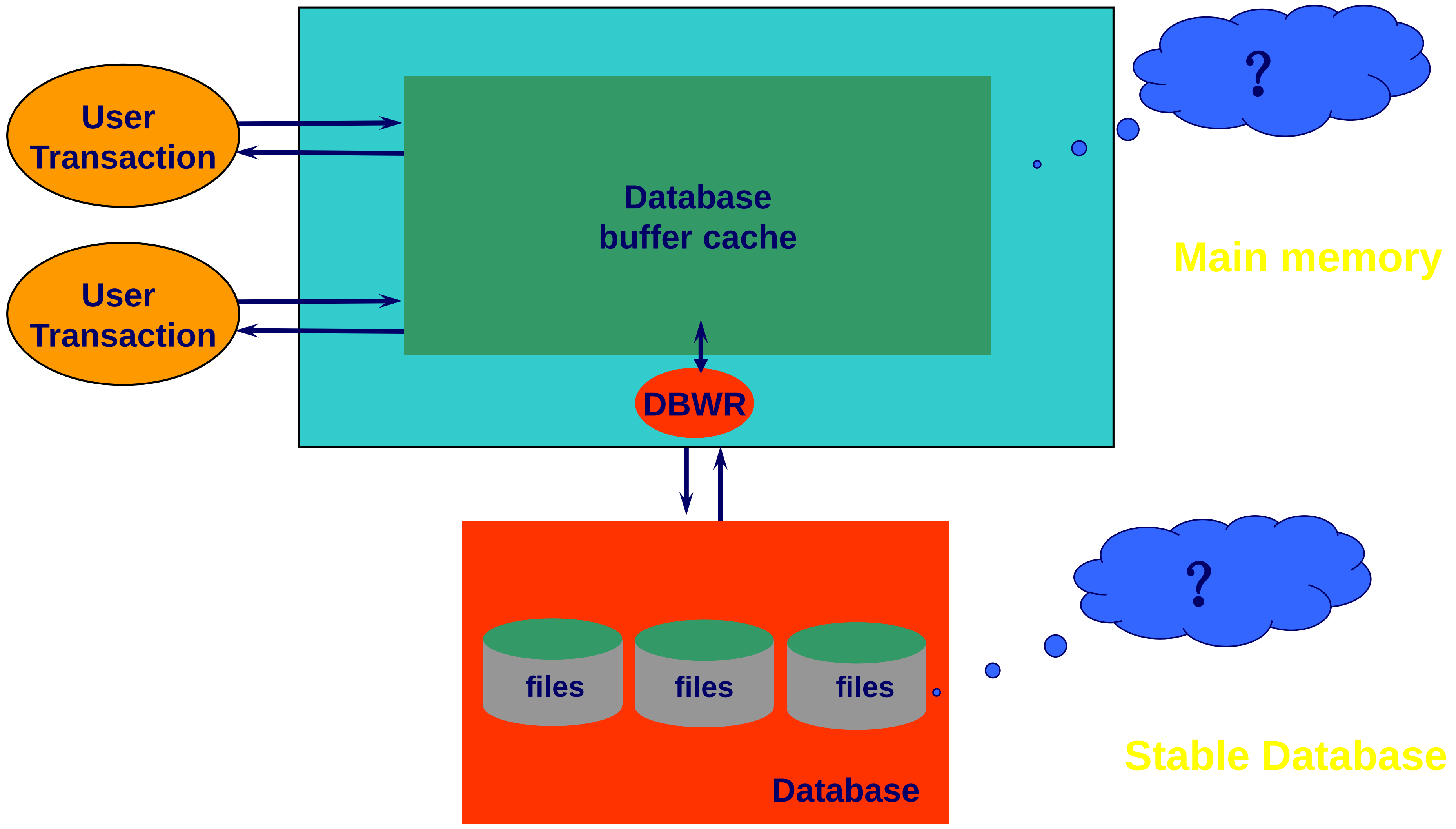
The DBMS Executing Environment



The DBMS Executing Environment

- A DBMS stores a database on a secondary storage. We will refer to the database on secondary storage as the *stable* database.
- Transactions execute within the main memory, the DBMS needs to bring portions of the stable database into a *database buffer* in main memory and flushes new data to the stable database .

Potential Failures



Potential Failures

- The storage of data includes four different types of media:
  - Main memory
  - Magnetic disk
  - Magnetic tape
  - Optical disk

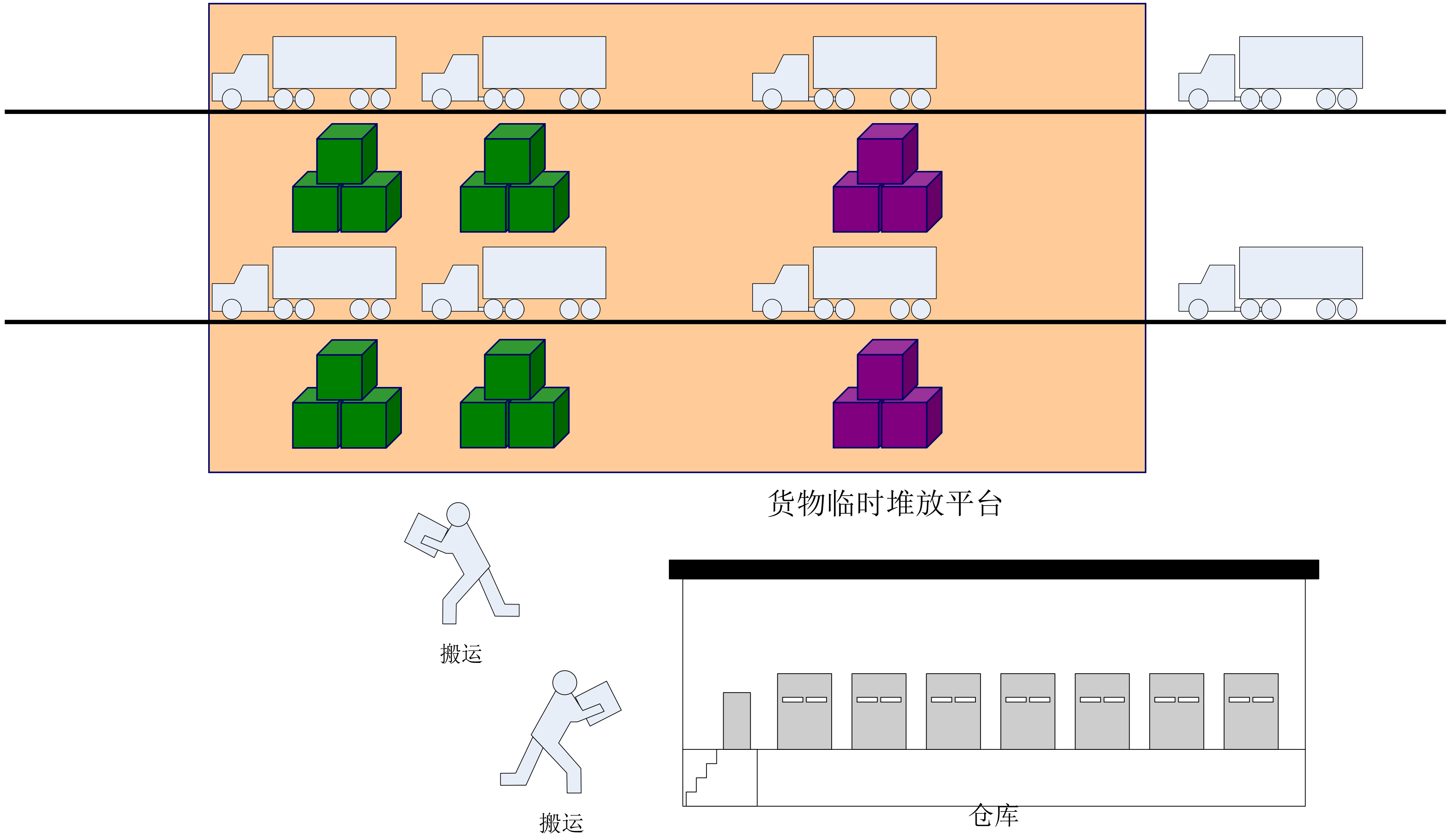
Potential Failures

- Different types of failure
  - System crashes due to hardware or software errors, resulting in loss of main memory.
  - Media failures, resulting in the loss of parts of secondary storage
  - Application software errors, which cause one or more transactions to fail
  - Natural physical disasters
  - Carelessness destruction of data
  - sabotage

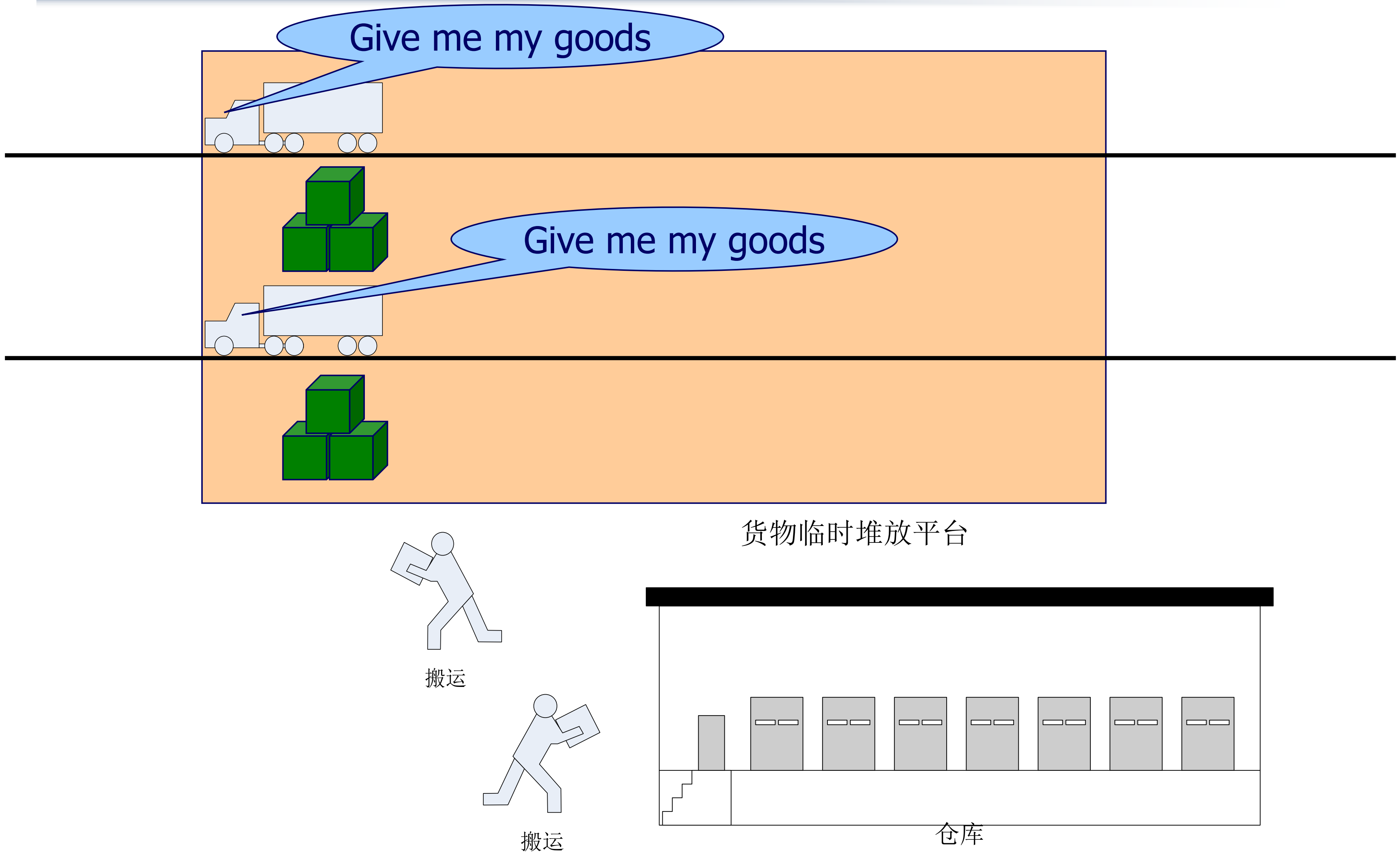
Potential Failures

- Whatever the cause of the failure, there are two principal effects that we need to consider
  - The loss of main memory
  - The loss of the disk copy of the database

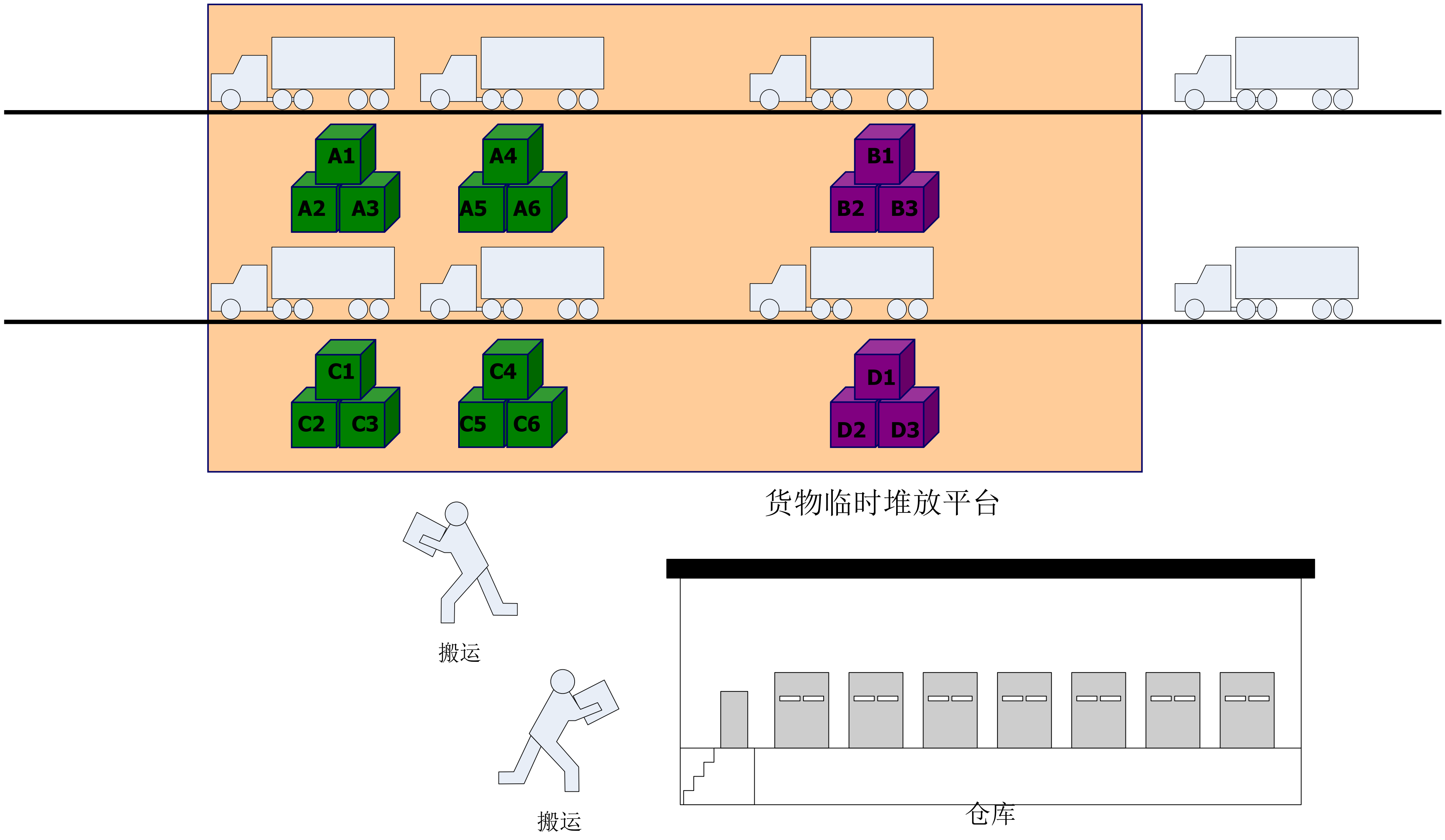
Warehouse Example



Warehouse Example



Warehouse Example



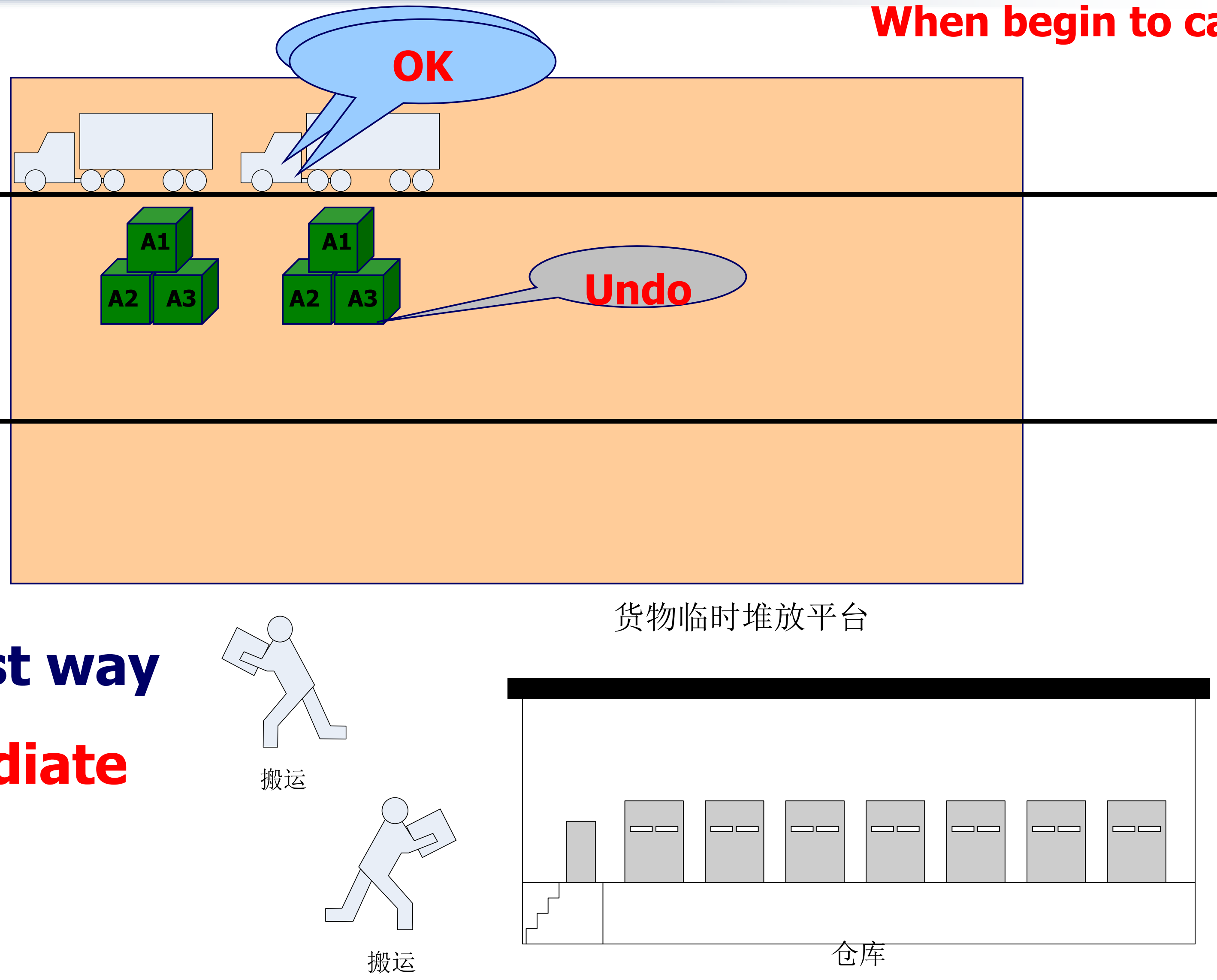
Warehouse Example-Questions

- When begin to carry the goods after they arrived?
  -

- When the driver can leave?
  -

Warehouse Example

When begin to carry?

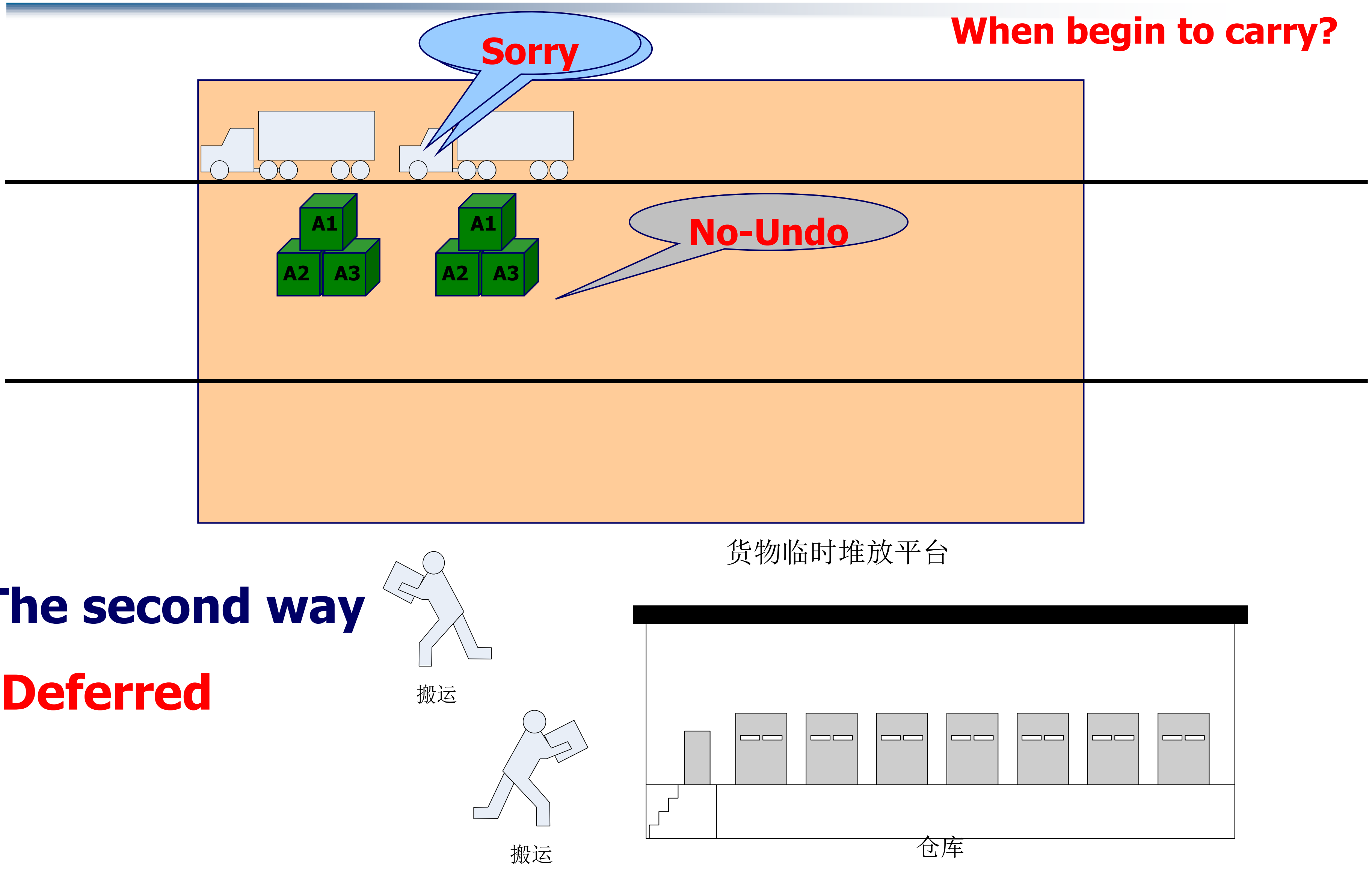


The first way

-Immediate

Warehouse Example

When begin to carry?

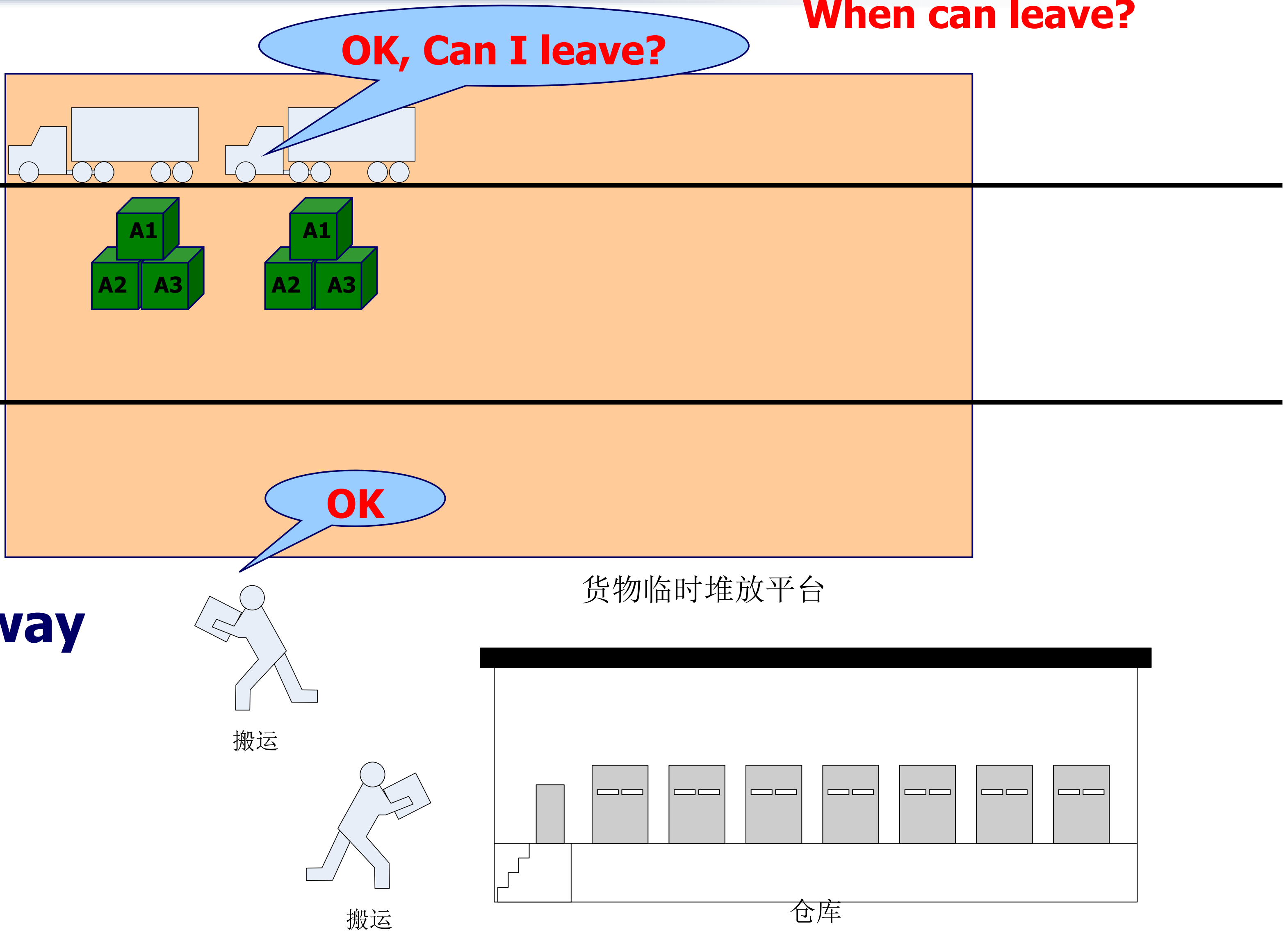


The second way

-Deferred

Warehouse Example

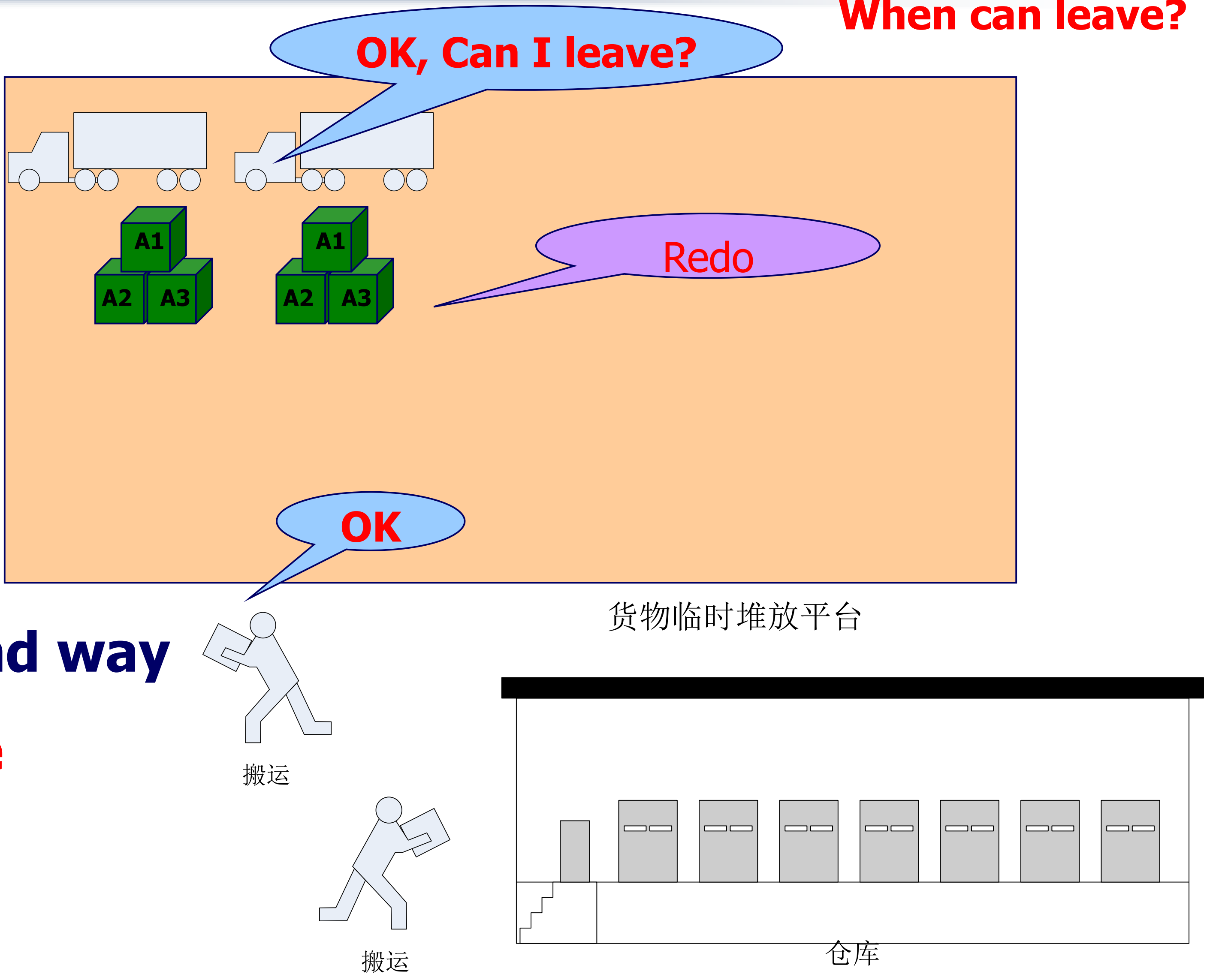
When can leave?



The first way

-Force

Warehouse Example



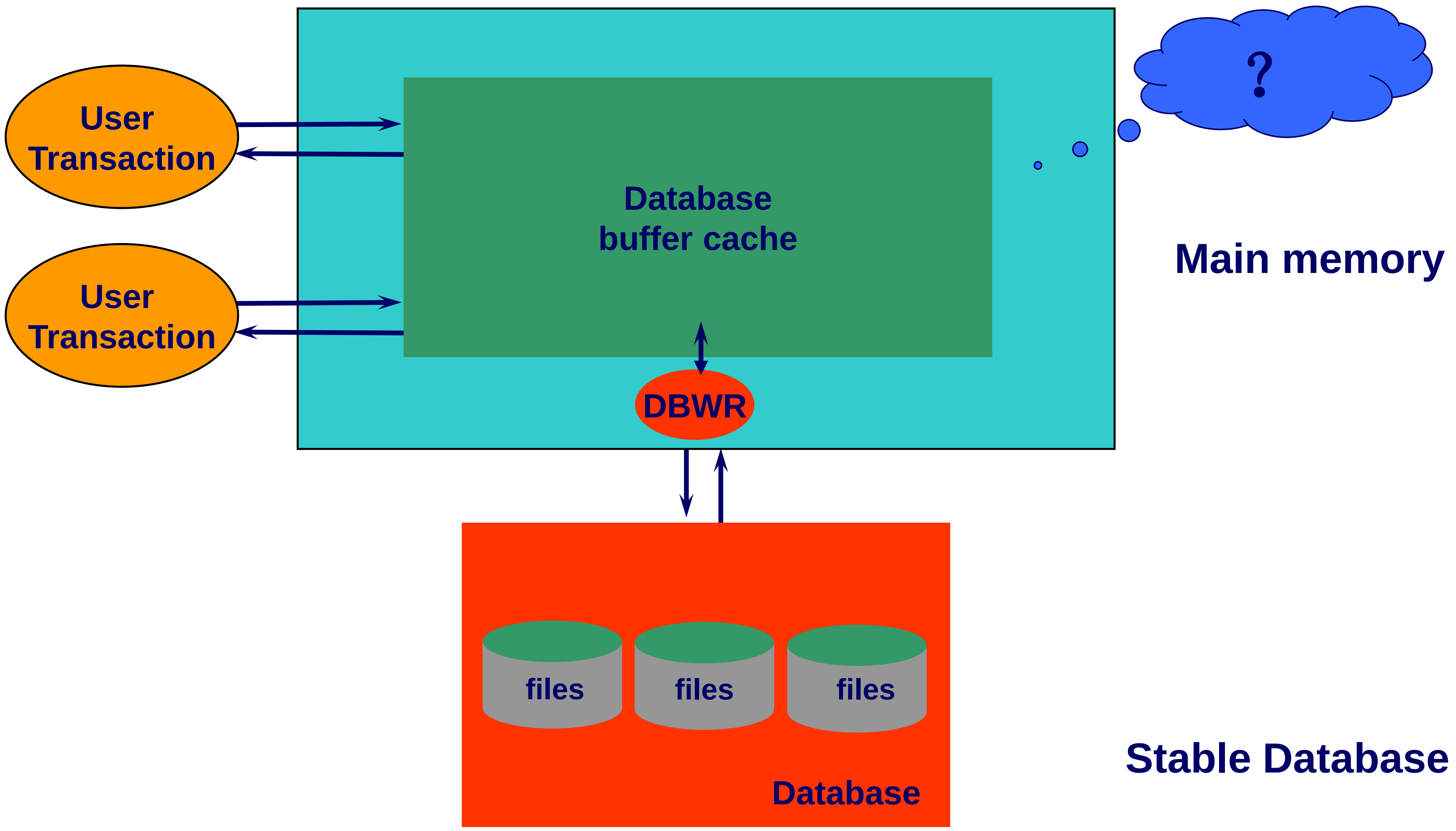
The second way

-No-Force

Warehouse Example-Conclusion

- Update policy when data come
  - Immediate - Undo
  - Deferred – No-Undo
- Propagation policy at commit point
  - Force – No-Redo
  - No-Force - Redo

How to do when Failures Happened



How to do when Failures Happened

Failures Happen

Immediate/No-Force	⇨	**Undo/Redo**
Immediate/Force	⇨	**Undo/No-Redo**
Deferred/No-Force	⇨	**No-Undo/Redo**
Deferred/Force	⇨	**No-Undo/No-Redo**

How to do when Failures Happened

- Recovery protocols usually implement two basic actions that are performed on the state of the stable database, namely, the undo action and the redo action.
  - undo action
    - This is required for atomicity. It undoes the effects that aborted transactions have had on the stable database.
  - redo action
    - This is required for durability. It redoes the effects that committed transactions have had on the stable database.

Log

- The information needed for the undo and redo actions is kept in a log.
- A log records:
  - identifiers of the transactions that are committed or aborted.
    - For example, [Ti, COMMIT]   [Tj, ABORT]
  - all modifications to the database.
    - physical logging
      - [Ti, D, b, a].
    - logical logging
      - [Ti, OP, OP Parameters, INV, INV parameters]

Log

- In a sense, the log represents the history of the transaction execution with respect to its updates. The order in which updates appear is the same as the order in which they actually occurred.

Log

- The DBMS maintains the log in a *log buffer* in main memory.
- In order to ensure that a log contains the needed information for recovery, it saves the log on secondary storage that survives systems failures.

Log

- The log is made stable, i.e., written on secondary storage, following two rules.
  - Undo Rule or WAL (Write-Ahead Logging) principle
    - The updates of a transaction are not applied in the stable database until after the log records that correspond to the updates show up in stable storage.
  - Redo Rule
    - A transaction is not committed until after the part of the log pertaining to the transaction is in stable storage.

The Process of Recovery

- The complexity of the crash recovery procedure—and the need for undo and redo actions—depend on the database update and update propagation policies employed by the DBMS.

The Process of Recovery

- The need for undo actions depends on which one of the two database update policies is used:
  - Immediate Updates
    - Pages in the database buffer can be flushed to the stable database before a transaction reaches its commit point.
    - Undo actions are needed in the event of a system failure.
  - Deferred Updates
    - Pages in the database buffer cannot be propagated to the stable database before the transaction has reached its commit point.
    - There is no need for undo actions in the event of a system failure.

The Process of Recovery

- The need for redo action depends on the page propagation strategy at commit time:
  - Force Propagation
    - A transaction is not committed until all its modified pages are written back to the stable database
    - There is no need for redo actions in the event of a system failure.
  - No-Force Propagation
    - Pages modified by committed transaction might not have been propagated to the stable database.
    - There is a need for redo actions in the event of a system failure.

The Process of Recovery

- In general, crash recovery techniques are classified into four types:
  - Undo/Redo protocol
    - immediate updates and no-force propagation policies
  - Undo/No-Redo protocol
    - immediate updates and force propagation policies
  - No-Undo/Redo protocol
    - deferred updates and non-force propagation policies
  - No-Undo/No-Redo protocol
    - deferred updates and force propagation policies

The Process of Recovery

- UNDO/REDO protocol must be the most practical of all, especially given the fact that nowadays systems failures are rather rare.
- In fact, this is the recovery protocol implemented by almost all DBMSs.

About Checkpoint

- After a system crash, recovery is performed as part of the system restart.
- Assuming strict executions and physical logging, the undo/redo recovery proceeds in three sequential phases:
  - analysis phase
  - undo phase
  - redo phases

About Checkpoint

- During the analysis phase, the log is scanned backwards from the end of the log in order to determine the following:
  - Which transactions have committed
  - Which transactions have aborted
  - Which transactions were active at the time of the crash

About Checkpoint

- During the analysis phase, the log is scanned backwards from the end of the log in order to determine the following:
  - Which transactions have committed
  - Which transactions have aborted
  - Which transactions were active at the time of the crash

About Checkpoint

- The analysis phase terminates by producing two transaction lists
  - redo list
    - containing all the committed transactions
  - undo list
    - containing both all the abort transactions and active transactions at the time of the crash

About Checkpoint

- During the undo phase, the log is again scanned backwards starting from the end.
  - For each update log record associated with a transaction on the undo list, the before image in the record is written to the data item specified in the record and then remove the transaction from the undo list.
  - The undo phase is only complete when all these have been removed and the undo list is empty.

About Checkpoint

- During the redo phase, the log is scanned forward, starting from the beginning.
  - For each update log record associated with a transaction in the redo list, the after image in the record is written to the data item specified in the record and then remove the transaction from the redo list.
  - The recovery is complete when the redo list is empty.

About Checkpoint

```
                                                         Crash
          T1 |------------------------------>                :
                    T2 |-------------------------------------  :
                              T3 |--------------<              :
                T4 |----------->                               :
                      T5 |------------------<                  :
                              T6 |--------------------->       :
                                                                :
        time t0………………………………………………………t9
```

About Checkpoint

- The basic UNDO/REDO recovery assumes that all the effects of the aborted and active transactions—and none of the effects of the committed transactions—were propagated to the stable database. As a result, it needs to scan the entire log!
- As the log grows longer, restart and the recovery procedure become prohibitively slow. Furthermore, the stable log may become very long and may not fit on disk.

About Checkpoint

- With checkpointing,
  - The effects of all transactions that are committed or aborted by the time of a certain checkpoint are propagated to the stable database, thereby eliminating the need to redo or undo these transactions after a system failure. Hence, their associated log records are not needed and can be discarded from the stable log.
  - Transactions that started and committed or aborted after the checkpoint need to be redone or undone completely.

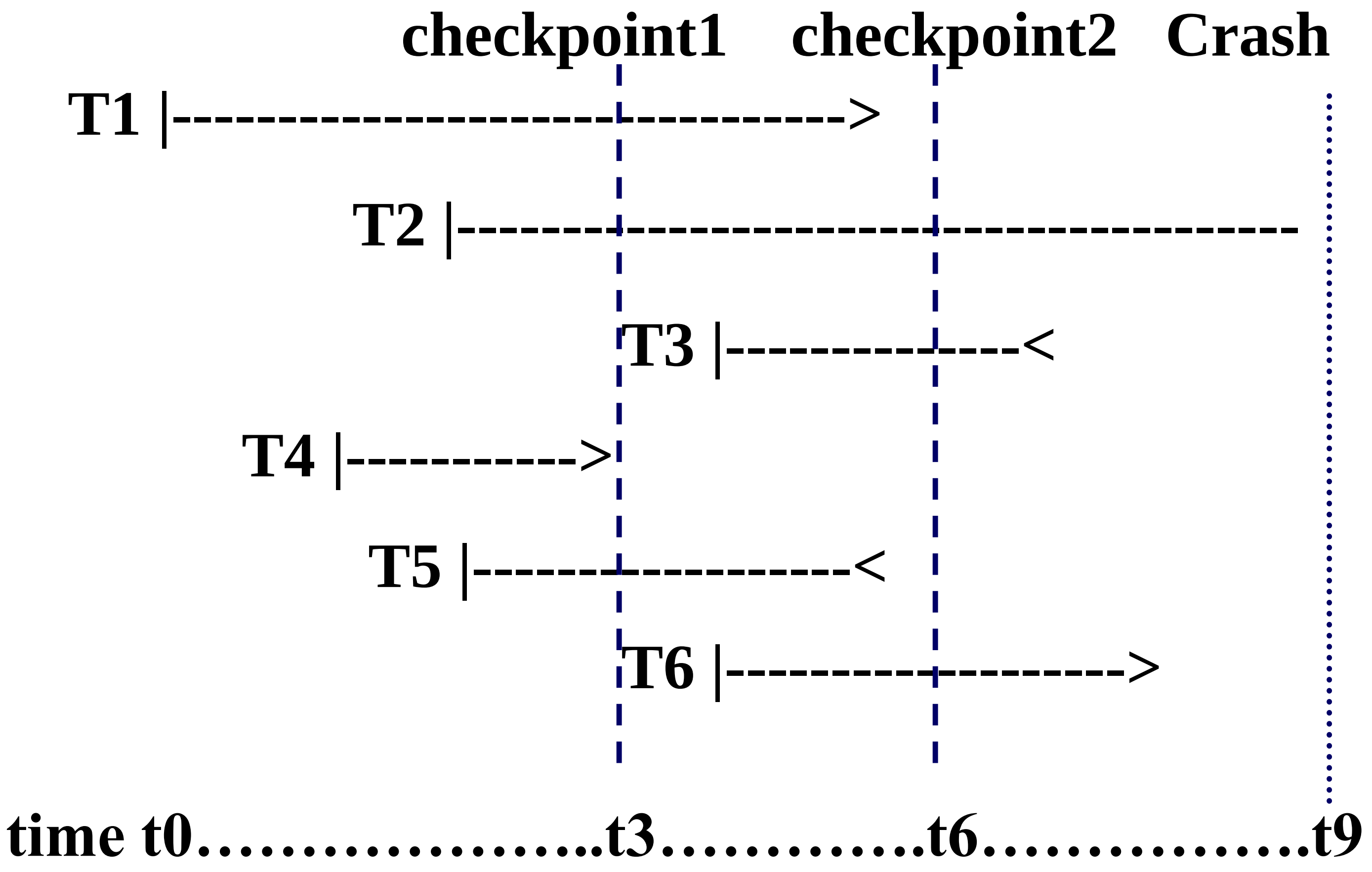
About Checkpoint

- When a checkpoint is taken, no transaction processing is performed.
- Whether the effects of transactions in progress are also propagated to the stable database during checkpointing depends on the particular checkpointing method.

About Checkpoint

- The most commonly used checkpoints are action-consistent checkpointing:
  - The action-consistent checkpoint method waits for all the operations in progress to reach completion, and then it forces all the updates in the database buffer onto the stable database, making them permanent.
  - As a result, the effects of active transaction during checkpointing are propagated to the stable database, along with those of the committed and aborted transactions.

About Checkpoint


checkpoint1
checkpoint2
Crash
T1 |--------------------------------->
T2 |--
T3 |--------------<
T4 |----------->
T5 |------------------<
T6 |------------------->
time t0………………...t3…………..t6……………..t9

Recovery from non-volatile storage failures

- All the techniques for recovery we discussed above is base on the assuming that non-volatile storage is not failure.
- Approaches for recoverying the database from non-volatile storage failure:
  - Static backup
  - Dynamic backup
  - Archive log
  - Redo

Summary

- In this chapter you should have learned:
  - The concept of transaction
  - Concurrency control protocols
  - Programming with Transactions
  - Recovery protocols