数据库系统原理

Database System Principle

邵蓥侠

Email：shaoyx@bupt.edu.cn

北京邮电大学计算机学院

计算机应用技术中心

PART 7

TRANSACTION MANAGEMENT

C, Pascal programs

词法/语法/语义分析

中间代码生成

(中间)代码优化

目标代码生成

目标程序代码

程序编译/**编译器**

process / thread

程序执行/**OS**

进程管理

并发控制&进程调度

死锁处理

Chapter. 15, 16

Chapter. 17, 18, 19

Query processing / DBMS

事务处理 / DBMS

query

扫描和语法/语义分析

关系代数表达式&查询树

查询优化 (§16)

(优化后)查询执行计划

查询代码生成 (§15)

查询计划执行的代码

transaction

事务管理 (§17)

并发控制&事务调度 (§18)

死锁处理

恢复技术 (§19)

Fig. 15.0.1

Introduction to Part 7

- DBS中，从DBMS的角度，用户对DB的访问体现为DBS中1个或多个**事务**的执行
- 事务是DBS中应用程序/数据库应用系统的基本逻辑单位，也是DBMS管理DBS运行的基本单位, § 17.1/17.2
- 多用户DBS中，多个用户对DB的并发访问体现为DBS中多个**数据库事务**的并发执行
- 事务是动态的，具有一定生命周期
  - 事务具有多种状态，事务的执行体现为各状态间的转换过程, § 17.4, Fig.17.1

Introduction to Part 7 (cont.)

- 在DBS中，事务执行时，特别是当多个事务对共享数据进行并发访问时，为维护DBS系统中数据的正确性(integrities)，事务必须满足一定的约束条件，表现为事务的4个基本特征(**ACID**) , § 17.1/17.2/17.4/17/5
  - 原子性(atomicity)，一致性(consistency)，独立性/隔离性(isolation)，永久性/ 持续性/操作结果永久保持性 (durability)

- DBMS中的**recovery-management component**负责保障事务的原子性(§ 19)
- **事务设计者/DBS应用程序编程者**负责保障事务的一致性

Introduction to Part 7 (cont.)

- DBMS中的concurrency-control component负责保障事务的独立性(§18)
- DBMS中的recovery-management component负责保障事务的永久性(§19)

- 当DBS中存在多个并发事务时，DBMS通过**事务调度**对各事务进行并发控制 (§18.4)，以保证并发事务的运行结果正确性
  - DBMS中的concurrency-control component依据事务调度可串行性（Serializability)的基本原理，对事务进行并发控制和调度, §18.6

北京邮电大学

Introduction to Part 5 (cont.)

- 具体实现技术包括：

  基于锁的并发控制技术(§ 18.1)， 多粒度控制技术(§ 18.3);

  基于时间戳的并发控制技术(§ 18.4) 、基于验证的并发控制技术(§ 18.5)、多版本控制(§ 18.6)

- 此外， DBMS的concurrency-control component还需要对多个并发执行的事务进行死锁处理(§ 18.2)

Chapter 17

TRANSACTIONS

§ 17.1 Transaction Concept

- Definition and composition
- ***Read*** and ***write***
- ***ACID*** properties

Transaction Concept (cont.)

- The transaction is
  - *a unit of* DBS application *program executing* that accesses and possibly updates various data items in DB

- A transaction is ***initiated*** by a user application program written in
  - data manipulating language such as SQL, for example *select-from-where*, or
  - programming languages, e.g. C++ or Java, with embedded DB accesses in JDBC or ODBC

Transaction Concept (cont.)

- A transaction consists of
  - *a collection of operations* that form a single logic unit of application works, delimited by the statements of the form ***begin transaction*** and ***end transaction***
  - operations: ***read*** or ***write*** on DB, or other operations
    - begin-transaction

      op_1;

      op_2;

      .

      op_n;

    end-transaction (e.g. commit or abort/rollback)

Transaction Concept (cont.)

- Conceptually, a transaction can be defined by ***read*** and ***write*** operations, which are independent of DBS query languages (e.g. SQL) and DBMS (e.g. SQL Server)
- E.g.1. *Account-transfer* T1 :

  transfer $50 from account A to account B

  T1 : **begin-transaction**

  read (account_A);

  account_A := account_A -50;

  write (account_A);

  read (account_ B);

  account_ B := account_ B + 50;

  write (account_ B);

  end-transaction

```
begin-transaction
read ( amount_A );
amount_A:= amount_A -50;
if  amount_A < 0  then
then  begin
          print(‘insufficient funds’)
          rollback T2
         end
else  begin
          write (account_A );
          read (amount_ B );
          amount_ B := amount_ B + 50;
          write (amount_ B );
          commit T2
        end
```

- abort/ ***rollback*** T2: 撤销之前对loan_A 的修改, 即恢复loan_A 的原值，并立即结束T2，不再执行其后的各操作

Fig.17.0.1 *Account_transfer* T2 with ***rollback***

```
DECLARE   @transfer_name   varchar(10)    /*事务变量定义*/
SET @transfer_name = ‘I-transfer-from-A-to-B’   /*事务命名*/
BEGIN TRANSACTION @transfer_name          /*事务开始*/
USE  ACCOUNT         /*打开数据库ACCOUNT*/
GO                   /*将上述批SQL语句提交SQL Server*/
UPDATE   account_A                      /* 修改A帐户*/
   SET  balance =  balance – 50
   WHERE  branch_name= ‘Brooklyn’
UPDATE   account_B                      /* 修改B帐户*/
   SET      balance = balance + 50
   WHERE  branch_name= ‘Brooklyn’
 GO
 COMMIT TRANSACTION @transfer_name    /*事务提交*/
 GO
```

Fig.17.0.2 *Account_transfer* T3 in T-SQL

ACID Properties (ref. to 17.2)

To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.

- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.

ACID Properties (cont.)

- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished——好像两者串行执行
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

17.2 A Simple Transaction Model (*Read* and *Write*)

- Each data access in transactions, such as *select*, *Update* in SQL, or *API* in ODBC, is translated/decomposed by DBMS into one or more ***read*** and ***write*** operations
- Refer to Fig. 17.0.3
- DBMS executes these ***read*** and ***write*** operations to fulfill data access in transactions
- For each transaction, DBMS allocates a ***local buffer*** in main memory as the working area for this transaction
- The database permanently resides on disk, but some portion of it is temporarily residing in the ***disk buffer*** in main memory

Read and ***Write*** Operations (cont.)

- ***read(X)***
  - transfer the data item ***X*** from DB files on disk or the *disk/system buffer* to a variable, also called X, in the *local buffer* belonging to the transaction that executes the ***read*** operation
- ***write(X)***
  - conceptually, transfer the value in the variable ***X*** in the *local buffer* of the transaction that execute the ***write*** operation back to the data item ***X*** in the DB files on disk
- In real DBS, the ***write*** operation issued by transactions does not necessarily result in the immediate update of the data on the disks, this operation may be temporarily stored in the *system buffers* and executed on the disk later
  - refer to Fig. 17.0.3

Transaction

data accesses on data item **x** and **y** issued by T_i,

e.g. ***select, insert, delete, update***, …

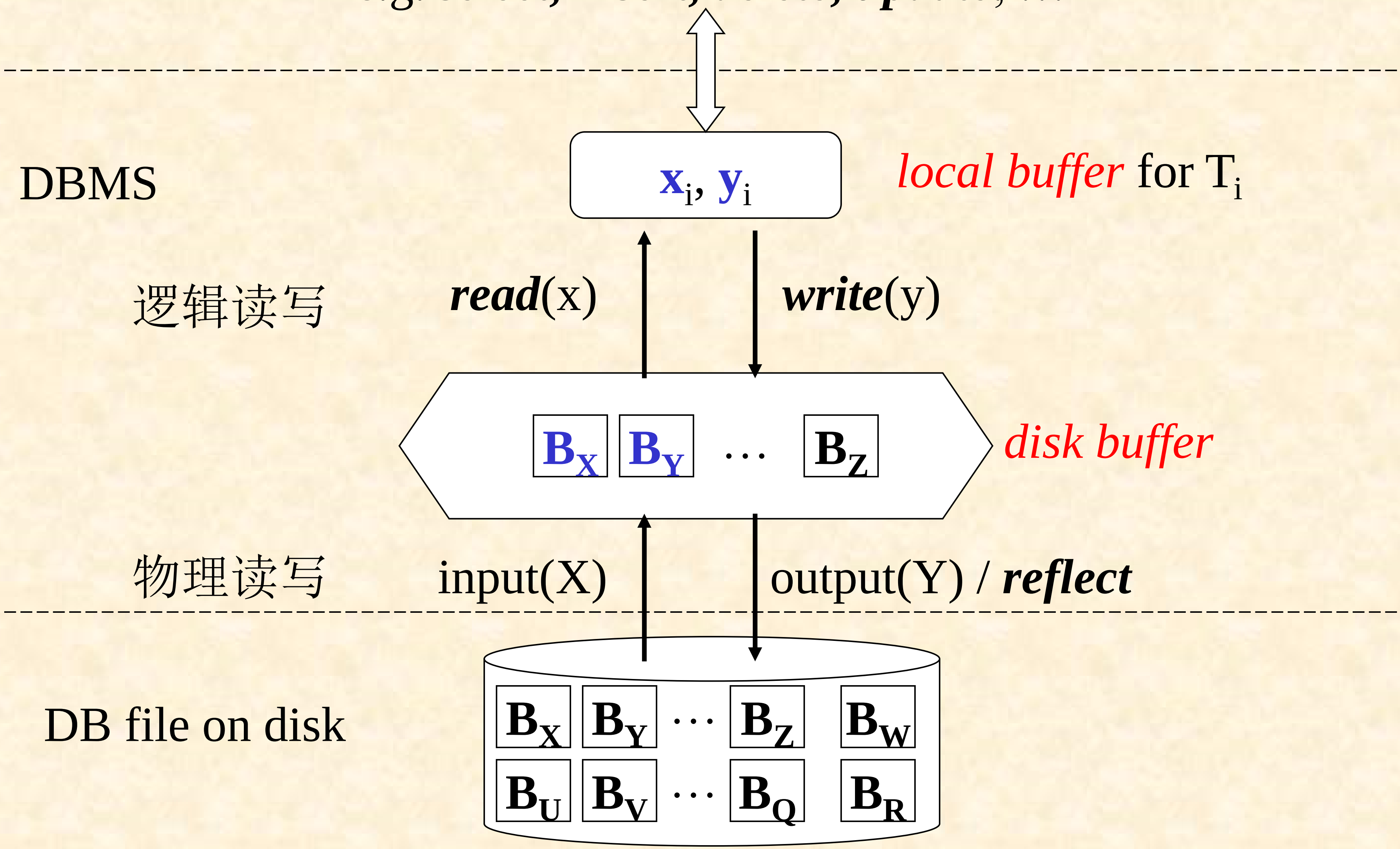


Fig.17.0.3 ***read*** and ***write*** operations

Transaction Properties

- To guarantee integrities of DB, *ACID Transaction Properties* should be maintained
  - Atomicity, Consistency, Isolation, Durability

- Consistency
  - execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the *consistency* of the DB, i.e. before and after the transaction execution, DB is in a correct DB state

— /*1个事务的正确执行使得DB从一个正确状态转移到另一个正确状态

— DB state: contents of tables in DB, or DB instance

- consistency/correctness

integrity constraints, specified by the application logic, are not violated

- during transaction execution the database may be temporarily inconsistent.

/*事务执行过程中，完整性约束有可能被破坏

- ensuring consistency of an individual transaction is the responsibility of the application programmers who codes the transaction
- E.g. The sum of A and B is unchanged by the execution of the transaction

A=1000, B=2000

1. ***read*** (A);
2. A := A -50;
3. ***write*** (aA);
4. ***read*** (B)
5. B := 50;
6. ***write*** (accB);

A=950, B=2050

北京邮电大学

Atomicity

- Either all operations of the transaction are ***reflected*** ( ) properly in the database, or none are
  - /*事务中所有操作或者全部成功完成，并且这些操作结果 被写入到DB; 或者事务中的所有操作一个都不作, 该事务对数据库和其他事务没有任何影响
  - /*将事务所有操作作为1个不可分割的整体

A=1000, B=2000

- E.g. If the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state
  - failure could be

    due to software or hardware

1. ***read*** (A);
2. A := A -50;
3. ***write*** (aA);

A=950, B=2000

4. ***read*** (B)
5. B := 50;
6. ***write*** (B);

Atomicity (cont.)

- The log-based recovery component in DBMS is responsible for ensuring atomicity

Isolation

- Even though multiple transactions may execute concurrently, the system guarantee that all transactions seem to execute serially, so the consistent states of DB can be preserved
  - each transaction is unaware of other transactions executing concurrently in the system
  - /*对多个并发执行的事务，1个事务的执行不能被其他事务干扰，即1个事物的内部操作和数据对其它事务是隔离的，并发执行的事务间不能相互干扰，从执行结果来看，相当于事务串行执行
- **Isolation** guarantees the correctness of concurrent execution of more than one transactions
- The concurrency control component in DBMS ensures the isolation properties

- **E.g.** if between steps 3 and 6 (of the fund transfer transaction) , another transaction **T2** is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

A=1000,	B=2000

T1	**T2**
1. **read**(A)	
2. $A := A – 50$	
3. **write**(A)	
	A=950, B=2000
	read(A), read(B), print(A+B)
4. **read**(B)	
5. $B := B + 50$	
6. **write**(B)	

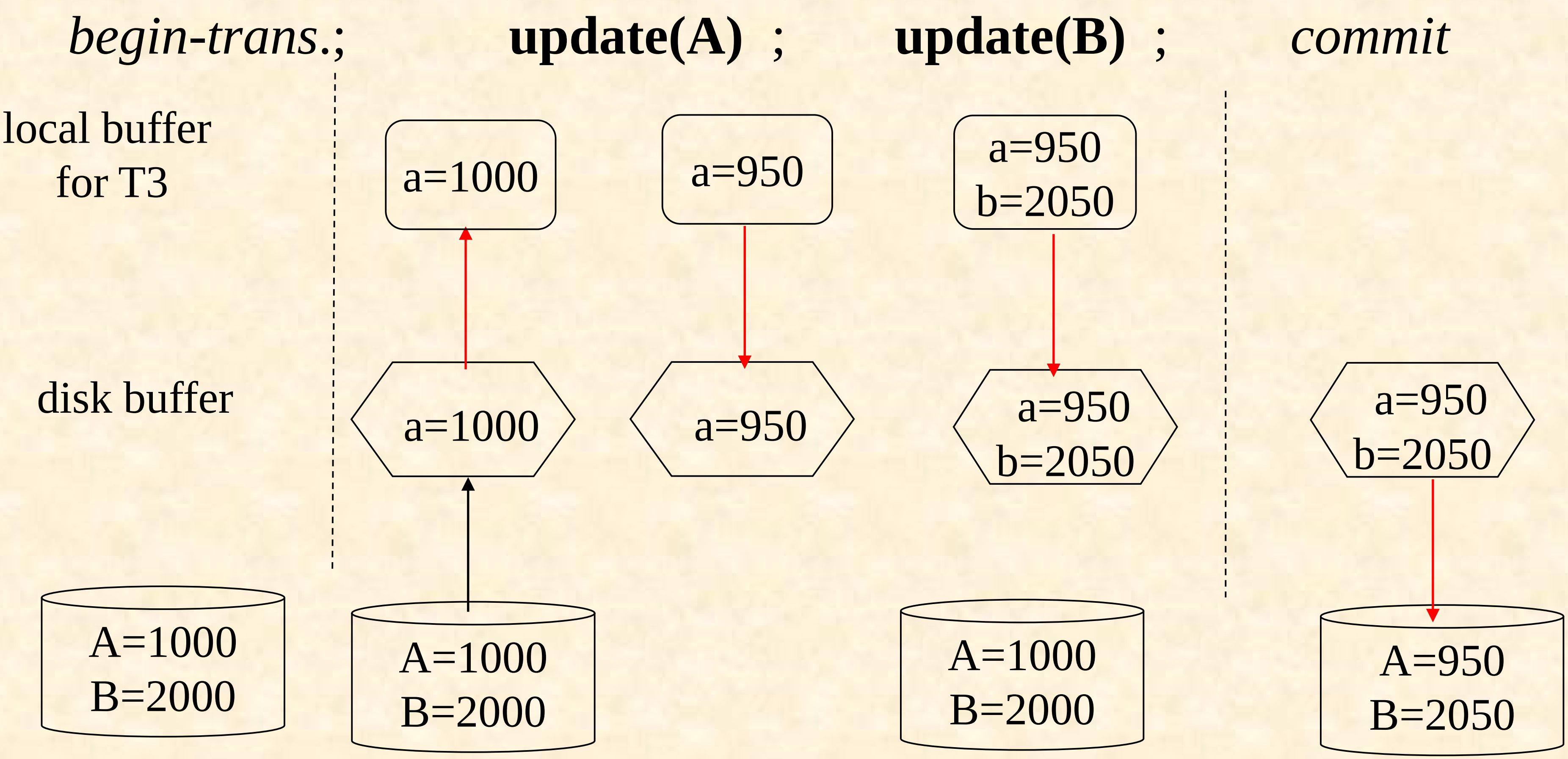
- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other.

Durability

- **A**fter a transaction completes successfully (i.e. the transaction is ***committed***), the changes it has made to the DB persist, even if there are system failure
  - /*事务一旦提交（成功完成），它对数据库中数据的改变就应该是永久性的，这些改变不随其后的数据库系统错误而丢失
  - ensured by recovery management component
- **E.g.1** the transfer of the $50 has taken place, the updates to the database by the transaction must persist even if there are software or hardware failures.
- **E.g.2** the modifications on ***Y*** in Fig.17.0.3
  - the modified results in *local buffer* should be ***reflected*** to the *DB file* on disk

Transaction Properties (cont.)

- e.g.2. for T3 in Fig.17.0.2



Transaction Properties (cont.)

- ACID mechanisms in DBS
  - A : transaction management / recovery management component
  - C: programmer/transaction designer, integrity constraints testing mechanism in DBMS
  - I: concurrency control component
  - D: recovery management component

§ 17.4 Transaction Atomicity and Durability - States

- Committed (提交) transaction
  - successfully complete all its operations

- Aborted (撤销/夭折) transaction
  - not finish all its operations successfully
  - *rollback*

- During its execution, a transaction can stay in several states, as shown in Fig.17.1
  - taking Fig.17.0.5, Fig.17.0.6 as examples

Transaction States (cont.)

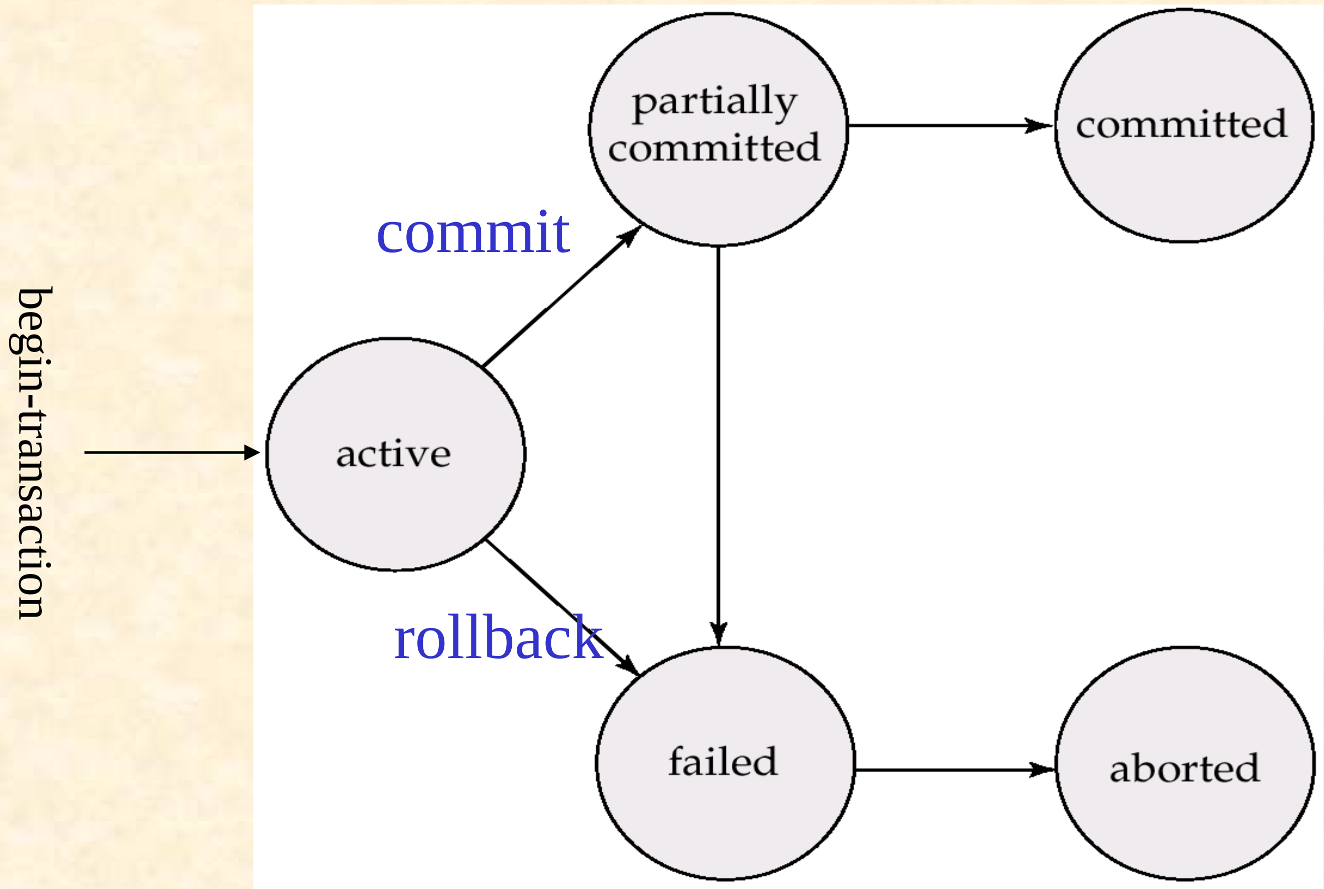


Fig.17.1. State diagram of a transaction

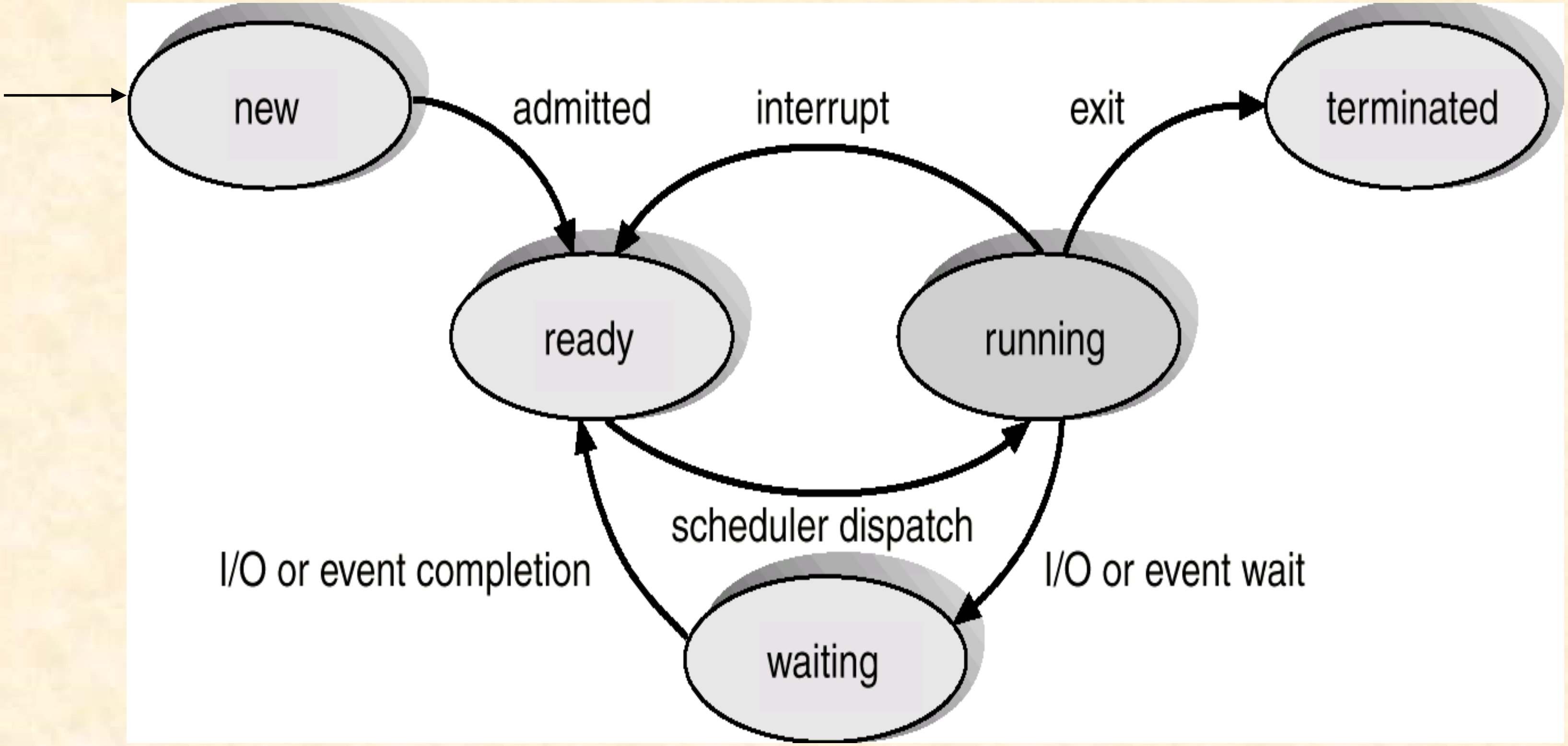


Fig.17.0.4 Process states

Transaction

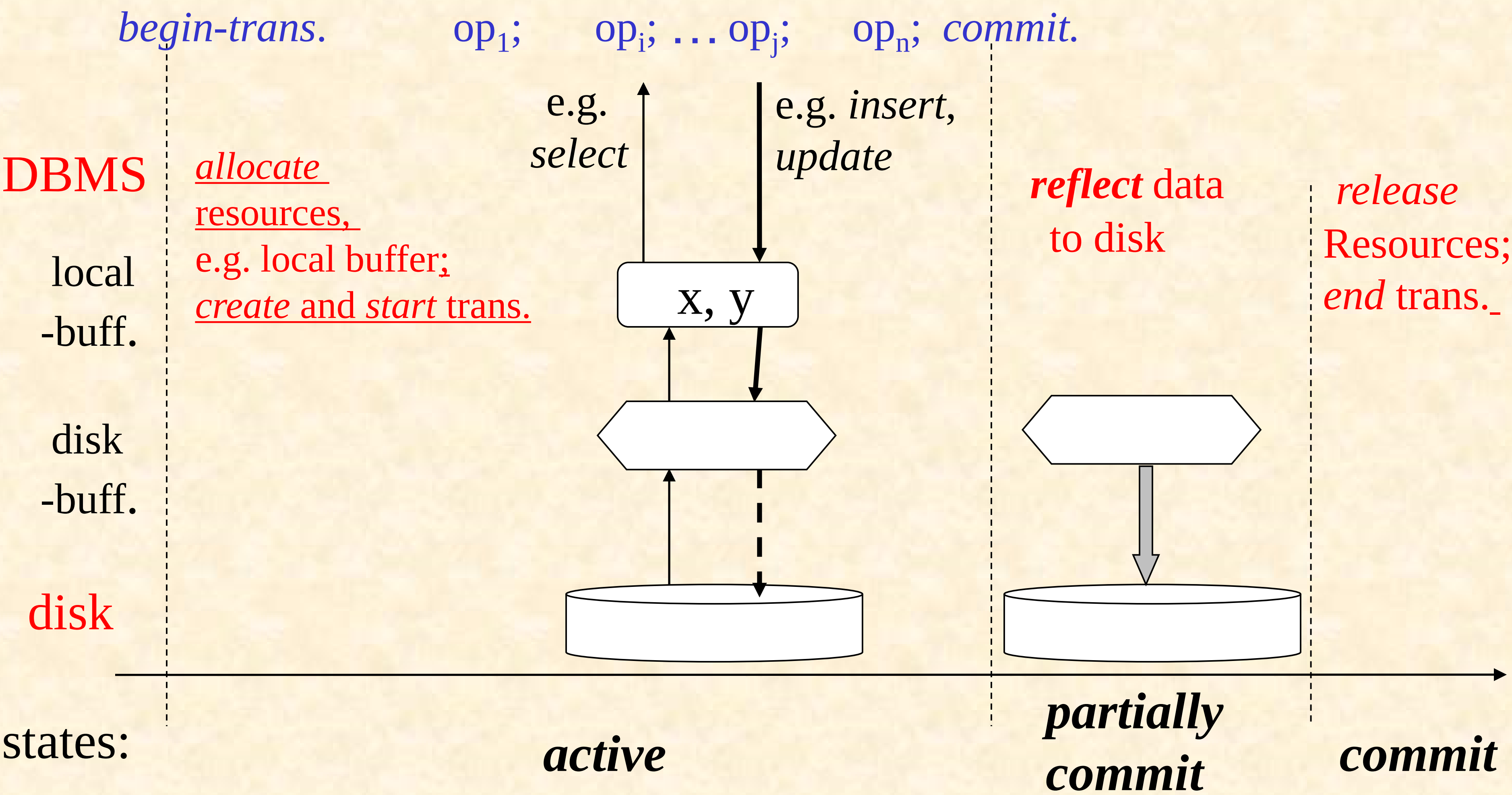


Fig.17.0.5 Life-cycle of a successful/*committed* transaction

Transaction

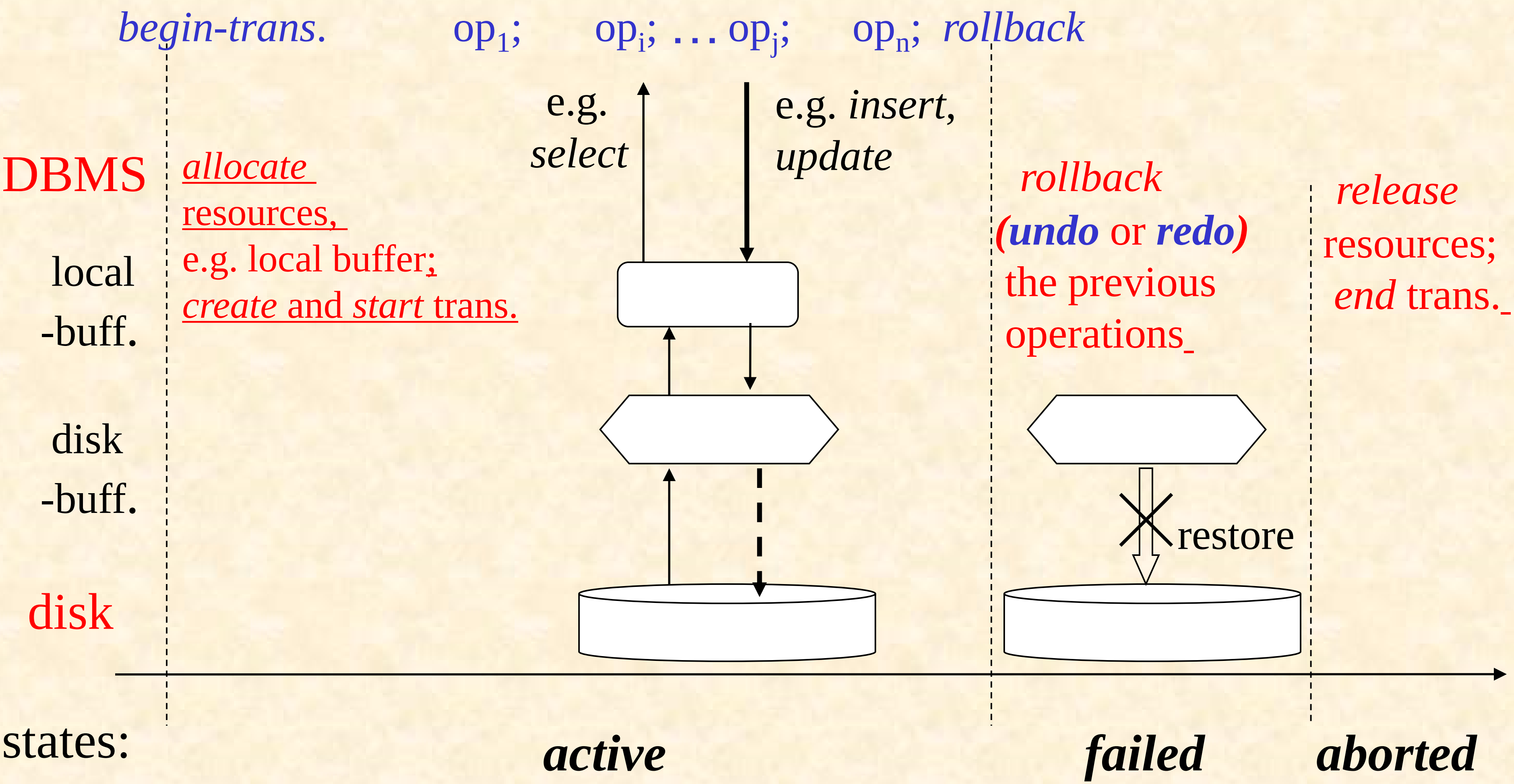


Fig.17.0.6 Life-cycle of a *rollback* transaction

Transaction States (cont.)

- Active
  - the initial state, the transaction stays in this state while it is executing from *begin-transaction*
  - in this state, the transaction is created (after begin-transaction is submitted**)**, resources, e.g. local buffer are allocated to the transaction
    - e.g.
- Note
  - transaction operations are executed, but the modifications on DB issued by operations **may** only stored temporally in the disk buffer, not ***reflected*** to DB file on disks immediately

Transaction States (cont.)

- Partially committed (部分提交) :
  - after the final statement (i.e. *Commit*) has been submitted, the transaction enters this state
  - in this state, the influences that the transaction's operations have made on DB are reflected from the disk buffer to DB by DBMS

  /*DBMS将disk缓冲区中的DB修改结果写入磁盘DB文件中
    - e.g.

Transaction States (cont.)

- Committed
  - after the transaction has successfully completed all its operations, and all the results of these operations have been reflected to the DB in ***partial commit*** state, it enters this state
  - in this state, the transaction
    - releases the resources occupied
    - and then terminated (quits DBS),

    the cycle-life of the transaction is ended
  - e.g.

Transaction States (cont.)

- Failed
  - if it is discovered that normal execution can no longer proceed, the transaction enters this state
    - e.g. *Account_transfer* T2 in Fig.17.0.1
  - in this state, the transaction is *rolled back* to restore DB to the state prior to the start of the transactions (§ 19)

    /*在此状态下进行失败处理工作
  - e.g. Fig.17.0.6

Transaction States (cont.)

- Aborted (夭折/异常结束/中止状态)
  - after the transaction has been rolled back in ***failed*** state, and the DB has been restored to its state prior to the start of the transactions, it enters this state to end the transaction

    /*表示DBMS已完成失败处理工作, 事务将要退出系统
  - in this state, the transaction
    - releases the resources occupied
    - reports to its user that the transaction is ***aborted***
    - and then terminated (quits DBS)

    , the cycle-life of the transaction is ended
  - e.g. Fig.17.0.6

An Example of ***Roll Back*** in Transaction

- Roll back (回滚、滚回)
  - 撤销到目前为止事务已经对数据库做的所有修改，使数据库状态恢复到事务开始前的状态

- E.g. define table ***SC*** as

  create table *SC*
  (*s#* **integer**,
  c# **integer**,
  grade **integer**,
  check (*grade* **between** 0 **AND** 100)
)

define the *update* transaction **update-tran** as:

update *SC*
set *grade* = *grade* + 5
where ….

update on some tuples

restore/rollback the tuples updated

s#	*c#*	*grade*
101	1	85
102	2	55
201	2	75
210	5	**100**
401	3	77
…	…	…
510	2	60

(a) the initial ***SC*** instance

s#	*c#*	*grade*
101	1	**90**
102	2	**60**
201	2	80
210	5	**105**
401	3	77
…	…	…
510	2	60

(b) ***SC*** instance when ***failure*** occurs during execution of transaction, i.e. ***integrity*** is violated

s#	*c#*	*grade*
101	1	85
102	2	55
201	2	75
210	5	**100**
401	3	77
…	…	…
510	2	60

(c) ***SC*** instance after ***roll back***, i.e. canceling all the updates on ***SC*** having been made, equal to the *initial instance*

An Example of ***Roll Back*** in Transaction (cont.)

- When tuple 1, 2, 3 have been successfully updated and tuple 4 is updated, the values of t4[*grade*]=105 >100, the integrity is violated
- DBMS ***rollback***s the transaction **update-tran**
  - restore the values of t1[*grade*], t2[*grade*], t3[*grade*] and t4[*grade*] to its initial values
- Another Example
  - importing data into DB in SQL Server, where existing some error data (e.g. unmatched data type) in the input files

§ 17.5 Transaction Durability
(for Concurrent Executions)

- Concurrent executions of a set of transactions allows
  - high throughput and resource utilization, reduced waiting time
  - e.g. the multi-user ticket-ordering system
- Demerits for allowing transaction concurrency
  - for a set of transactions which contain operations on ***shared data***, the different schedules (serial, concurrent) may result in different final DB states after all these transactions terminate
- Concurrency-control schemes and transaction schedules are needed

Concurrent Executions (cont.)

- A ***schedule*** on a set of concurrent transactions, arranged by DBMS concurrency-control component, specifies the chronological order in which instructions/operations of concurrent transactions are executed
- a schedule for a set of transactions must consist of all instructions of those transactions
- must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

An Example

- T1： transfer $50 funds from account-A to account-B
- T2: transfer 10% of the balance from A to B
- initial value of A : $1000
  initial value of B : $2000
  initial (A + B) : $3000

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

← A=950, B=1050

Final results:

A=$855, B=$2145

(A + B) = $3000

Fig.17.2 Serial S1
i.e. T1; T2

S1:

T1:	T2	account-A *DB file*	account-B *DB file*
		$1000	*$2000*
read (A1)		$1000	
A1:= A1-50		$1000	
***write** (A1)*		$950	
read (B1)			$2000
B1:= B1+50			$2000
***write**(B1)*			$2050
	read (A2)		
	temp:= A2*0.1		
	A2 := A2- temp	$950	
	***write** (A2)*	**$855**	
	read (B2)		$2050
	B2:= B2 + temp		**$2145**
	***write**(B2)*		

time t

temp=95, Final (A + B) = $855 + $2145 = $3000

Fig.17.0.7 Analysis of serial schedule S1

An Example (cont.)

- After T1 and T2 are finished under the constraints of the schedule S2,
  - the sum of A and B is preserved as 3000
  - the final value of ***account-A*** is **850**
  - the final value of ***account-B*** is **2150**
- Serial schedule S1 and S2 result in different final values of shared data items ***A*** and ***B***,
  - though they are all correct schedules

T_1	T_2
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	

Fig.17.3 Serial S2

i.e. T2; T1

T_1	T_2
read (A) $A := A - 50$ write (A)	
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A)
read (B) $B := B + 50$ write (B) commit	
	read (B) $B := B + temp$ write (B) commit

Fig.17.4 S3

并发调度S3将针对A的2次写操作、针对B的2次写操作安排在相近时间执行，利于DBMS在checkpoint时刻一次性地将A、B的修改结果写回数据库文件，减少DB file的写/output操作次数；
类似地，将针对A的2次读操作、针对B的2次读操作安排在相近时间执行，有利于减少对DB file 的读/input操作次数

- Final results:

A=$855, B=$2145

(A + B) = $3000

S3:

T1:	T2	account-A *DB file*	account-B *DB file*
		$1000	*$2000*
read(A1)		$1000	
A1:= A1-50		$1000	
***write**(A1)*		$950	
	read (A2) =950		
	temp:= A2*0.1		
	A2 := A2- temp	$950	
	***write**(A2)*	**$855**	
read (B1)			$2000
B1:= B1+50			$2000
***write**(B1)*			$2050
	read (B2)		
	B2:= B2 + temp		$2050
	***write**(B2)*		**$2145**

time t

temp=95 , final (A + B) = $855 + $2145 = $3000

Fig.17.0.8 Analysis of concurrent schedule S3

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Fig.17.5 Concurrent S4

- Note:

  assuming in Fig.17.5, ***read*** fetches data directly from DB file on disk

Final results:

A=\$950, B=\$2100

(A + B) = \$3050

S4:

T1:	T2	account-A *DB file*	account-B *DB file*
		$1000	*$2000*
read(A1)		$1000	
A1:= A1-50 =950		$1000	
	read(A2)=*1000*	$1000	
	temp:= A2*0.1=100		
	A2 := A2- temp=900		
	***write** (A2)*	<u>$900</u>	
	read(B2)=2000		$2000
***write** (A1)*		**$950**	
read(B1)			$2000
B1:= B1+50 =2050			$2000
***write** (B1)*			<u>$2050</u>
	B2:= B2 + temp		
	write(*B2)*		**$2100**

time t

A1=950, temp=100, final (A + B) = $950+ $2100 = $3050

Fig.17.0.9 Analysis 1 of concurrent schedule S4

Why S3 gives a correct result while S4 leaves a wrong state ?

- In S3, T1 and T2 access shared data ***A*** and ***B*** serially
  - with respect to shared data A and B, S3 has the same executing results as serial *S1*, and *S3* is equivalent to *serial S1*
- In S4
  - T1 and T2 operate on shared data ***A*** and ***B*** in interweaving (交错, 交织) ways
  - S4 has not the same executing results as serial S1, and *S4* is not ***equivalent to*** *serial S1*;
  - S4 leaves the database in an ***inconsistent*** state, and isolation and consistency are violated

§17.6 Serializability

- /*数据库系统中，多个事务的**串行执行**可以保证事务的ACID特性（如S1, S2），但执行效率低
- /*多个事务**并发执行**时，如果事务操作的调度顺序不当（如S4），将影响DBS正确性
  - e.g. S4 in

- /*事务并发控制基本原理
  - 事务调度可串行化:

    相互冲突的操作串行执行，非冲突操作并行/交织执行
  - e.g. S3 in

Serializability (cont.)

- How to identify whether or not a ***concurrent schedule*** S on a set of transactions is right ?
  - S is **equivalent** to a ***serial*** one S' ?

- /*将对S正确性的判断转换为对S可串行性的判断
- Schedule equivalence
  - conflict equivalence
  - view equivalence

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus, serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.
- Different forms of schedule equivalence give rise to the notions of:
  1. **conflict serializability**
  2. **view serializability**

17.5.1 Conflict Serializability

- Given two transactions T_i = { I_i }, T_j = { I_j }, and a schedule S on T_i and T_j

- I_i of transactions T_i and I_j of T_j are **conflict**, if and only if
  - there exists some item *Q* accessed by both I_i and I_j, and at least one of these two instructions is ***write***(*Q)*
  - *i.e.*
    - I_i = **read**(Q), I_j = **write**(Q)
    - I_i = **write**(Q), I_j = **read**(Q)
    - I_i = **write**(Q), I_j = **write**(Q)

l_i = **read**(*Q*), l_j = **read**(*Q*),
l_i and l_j don't conflict.

Conflict Serializability (cont.)

- S and S' are ***conflict equivalent***
  - if the schedule *S* can be transformed into *S'* by a series of swaps of non-conflicting instructions
  - e.g.

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Fig.17.7 S5

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Fig.17.8 S6

Conflict Serializability (cont.)

- Note:
  - only swapping of two operations/instructions I_i and I_j that belong to two different T_i and T_j respectively are permitted
  - swapping should not change the orders of instruction executing in each transaction
    - e.g. in Fig.17.7, swapping of ***read***(B) and ***write***(B) in T_1 is not allowed
  - swapping should not change the orders of conflict instruction executing in S
    - e.g. in Fig.17.7, swapping of ***write***(A) in T_1 and ***read***(A) in T_2 is not allowed

Conflict Serializability (cont.)

- A ***concurrent*** schedule *S* is ***conflict serializable*** if it is conflict equivalent to a serial schedule S'
  - e.g. concurrent S5 in Fig.17.7 is equivalent to S6 in Fig.17.8
- There are some schedules of transactions, which are not conflict serializable
  - e.g.

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

Identification of Conflict Serializability

- Given a concurrent schedule S, how to determine whether or not S is conflict serializable (i.e. is right) ?

- Method 1
  - starting from the concurrent *S*, maintaining executing orders of conflicting operations in *S*, and swapping executing orders of non-conflicting operations in *S*
  - observing whether or not a serial S' can be obtained

- Method 2
  - precedence-graph in Fig.17.10, 17.11

Identification of Conflict Serializability -An Example

- Given the following concurrent schedule S on transactions T1, T2, T3, and T4
  - determine whether or not S is conflict-serializable ?
  - if S is conflict-serializable, give its equivalent serial schedule; and if it is not, give the reason

Identification of Conflict Serializability -An Example (cont.)

Time	T1	T2	T3	T4
1			read(X)	
2		write(X)		
3			write(Y)	
4				read(Y)
5	read(Y)			
6				read(X)
7				write(Z)
8	write(X)			

Fig.14.0.10

- Answer: S is equivalent to <T3; T2; T4; T1>

Precedence-graph

- Given a schedule S of a set of transactions T_1, T_2, ..., T_n , the precedence graph (前趋图) G(S) for S is
  - a direct graph G(V, E) where the vertices are the transactions
  - an arc from $T_i \rightarrow T_j$, for which one of three conditions holds
    - T_i executes write(Q) before T_j executes read(Q),
    - T_i executes read(Q) before T_j executes write(Q)
    - T_i executes write(Q) before T_j executes write(Q)

Precedence-graph (cont.)

T_i	T_j
write(Q)	
	read(Q)

(a)

T_i	T_j
read(Q)	
	write(Q)

(b)

T_i	T_j
write(Q)	
	write(Q)

(c)

t

Fig. 17.0.11 Cases for arc $T_i \rightarrow T_j$ in precedence graphs

Precedence-graph (cont.)

- A schedule is ***conflict serializable*** if and only if its precedence graph is acyclic（无环的）
- E.g. Concurrent schedule S (in the next slide) and its precedence graph

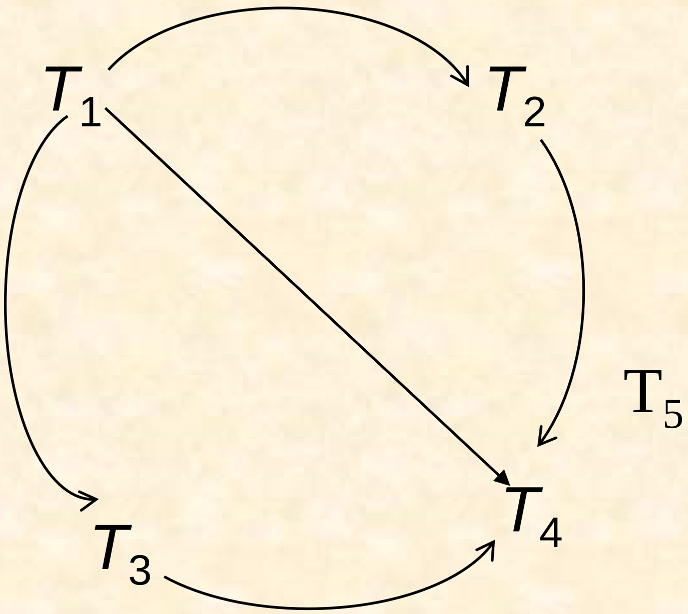


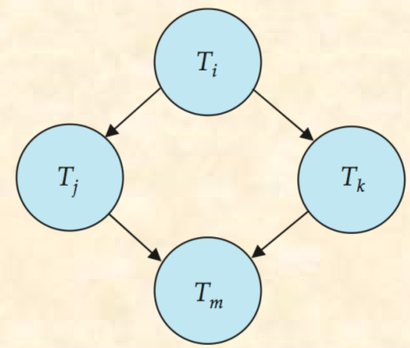
Fig.17.0.12
Precedence graph for S

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				

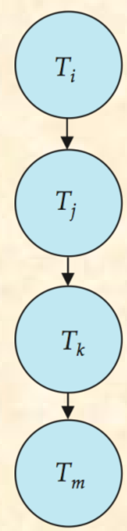
Fig.17.0.12 Concurrent schedule S

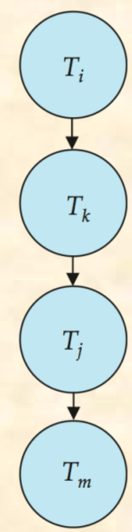
Testing for Conflict Serializability

- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - That is, a linear order consistent with the partial order of the graph.
  - For example, a serializability order for the schedule (a) would be one of either (b) or (c)



(a)





Construction of Conflict Serializable Concurrent Schedule

- Given a serial schedule S, how to construct a concurrent S', which is conflict equivalent to S
  - starting from schedule S, maintaining executing orders of conflicting operations in S, and swapping executing orders of non-conflicting operations in S, resulting in a conflict serializable S'
  - e.g. serial schedule S6 in Fig.17.8 and concurrent S5 in Fig.17.7

§ 17.7 Recoverability

(for transaction isolation and atomicity)

- Recoverability address how to deal with the schedules when transaction failures occur
- Requirements
  - when transaction T_i fails, the operations already executed should be *rollbacked/undone/aborted/canceled* to ensure the atomicity
  - when transaction T_i fails, any T_j *that is dependent on* T_i should also be aborted ,
    - e.g. T_j reads data having been written by T_i

17.7.1 Recoverability (cont.)

- **A recoverable schedule** is a schedule, where for each pair of T_i and T_j
  - if T_j reads a data items previously written by T_i , the ***commit*** operation of T_i appears before the ***commit*** operation of T_j.

T_i	T_j
write(Q)	.
.	.
.	*read(Q)*
commit	.
	commit

/*解释：

T_j 依赖于 T_i； 当T_i中的*write(Q)*需要aborted/rollback时，由于T_j还没有提交，T_j 可以rollback, 因此T_j中的*read(Q)* 操作也随之aborted

Fig.17.0.18 Definition of recoverable schedule

An example of non-recoverable schedule

- S9 in Fig.17.14 is *not* recoverable if T_9 commits immediately after the *read(A)*
  - if T_8 is aborted *after* T_9 *has been committed*, the value of the data item **A** will be ***rollbacked*** to its old value before T_8 executes.
  - but T_9 has read the new value of the data item **A** and has been committed

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

Fig.17.14 S9

17.7.2 Cascadeless Schedules

- **Cascading rollback**
  - a single transaction failure leads to a series of transaction rollbacks
- E.g. before T_{10}, T_{11} and T_{12} are committed, just after T_{12} reads the item A, when T_{10} rollbacks due to failures, its dependent transactions T_{11} should also rollback, resulting in T_{12} also rollback

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

Fig.17.15 S10

Cascadeless Schedules (cont.)

- Cascading rollback is undesirable, because it leads to the undoing of a significant amount of previous works;
- **Cascadeless** schedules is expected for concurrent transactions

- **Cascadeless** schedules are the schedules that cascading rollbacks cannot occur; formally,
  - for each pair of transactions T_i and T_j such that T_j reads a data item Q previously written by T_i, the ***commit*** operation of T_i appears before the ***read*** operation of T_j.
  - Fig.17.0.19

Cascadeless Schedules (cont.)

T_i	T_j
write(Q)	.
..	.
commit	.
	read(Q)
	.
	commit

T_j只能读取已提交事务T_i所更新的数据

Fig.17.0.19
Definition of cascadeless schedule

/*注意与 recoverable schedule定义的区别

T_i	T_j
write(Q)	.
.	.
.	*read(Q)*
commit	.
	commit

Fig.17.0.18 Definition of recoverable schedule

Cascadeless schedule vs recoverable schedule

- **<u>Every cascadeless schedule is also recoverable！ ！</u>**
- But a recoverable may not be a cascadeless schedule

- An example of cascadeless and recoverable schedule in Fig.17.0.20

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
commit		
	read(A)	
	write(A)	
	commit	
		read(A)
		commit

17.8 Transaction Isolation Level
(for Concurrency Control)

- A database must provide a mechanism that will ensure that all possible schedules are both:
  - Conflict serializable.
  - Recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur
- **Goal** – to develop concurrency control protocols that will assure serializability.

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g., a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g., database statistics computed for query optimization can be approximate (why?)
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read, but successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.

Levels of Consistency in SQL-92

- Lower degrees of consistency useful for gathering approximate information about the database
- Warning: some database systems do not ensure serializable schedules by default
  - E.g., Oracle and PostgreSQL by default support a level of consistency called snapshot isolation (not part of the SQL standard)

Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - E.g. in JDBC, connection.setAutoCommit(false);

略

Appendix A
Transactions in SQL Server

- SQL Server系统中有三种**事务执行模式**：显示、自动提交、隐性，用于启动事务执行
- Explicit transactions
  - delimited by ***begin-transaction*** and ***commit*** or ***rollback***
  - defined by user/application program, *explicitly controlled by user*
  - 工作在SQL Server系统的显示事务执行模式下
  - e.g. *account-transfer transaction* in Fig.15.0.2

Appendix A
Transactions in SQL Server (cont.)

- Auto-commit transactions
  - SQL Server 默认的事务管理模式
  - 在此模式下，每条单独的*Transact-SQL*语句均是一个事务
    - e.g.

      CREATE, ALTER TABLE, SELECT, UPDATE, INSERT, DELETE, DROP, FETCH, OPEN
  - 每条语句完成时，可以成功提交(commit)，也可以失败滚回 (rollback)
  - e.g. update SC

    set Grade=Grade +10

    当某个学生Grade已经超过99，再执行本语句将导致事务回滚，SQL Server撤销整个更新语句的执行结果

Appendix A
Transactions in SQL Server (cont.)

- Implicit transactions /隐性事务
  - SQL Server系统利用隐性事务模式设置命令设置系统的事务执行模式

    set implicit_transaction on
  - 在此模式下，每个事务无需（用begin-transition）显示地定义事务的开始，但仍以显示的***commit*** or ***rollback***结束
  - 当一个事务结束后，系统自动启动下一个事务，无需begin-transition，形成连续的事务链

隐性事务执行模式

set implicit_transaction ***on***

（用户定义）

op_1;
op_2;
…
op_m
commit;

事务1

op_{m+1};
op_{m+2};
…
op_k
rollback;

事务2

…

(单独T-SQL语句, 每条语句对应一个单独隐性事务)

Create;
Drop;
….
Select;
Open;
Drop;
Insert;
Delete;
Update;
…

set implicit_transaction ***off***

SQL Server 批处理中的事务

- 批处理
  - 包含1个或多个SQL语句的组，由应用程序从客户端一次性地发送到服务器端的DBMS引擎去执行执行
- 事务与批之间多对多关系
  - 1个事务可以包括多个批，1个批中也可以由多个事务组成
- SQL Server系统中，在大多数情况下，当批中的某些SQL语句执行发生错误时，DBMS将停止执行当前语句及其后面的语句，但发生错误之前已经执行的语句是有效的，不会被回滚(rollback)

Check 语义约束： grade between 0 and 100

```
begin tran
    insert into SC values ('s1', 'c1', 90)
    update SC
    set  GRADE=GRADE + 12
 commit tran
 select  *  from  SC
```

- ------

update 操作被回滚；

但Insert 操作未被回滚，('s1', 'c1', 90)被成功插入；

不满足事务原子性要求！！！

批处理中的事务 (cont.)

在SQL Server 中，如果需要事务中的全部操作/语句作为1个整体，同时提交或回滚，在提交“批”之前，采用下述语句进行设置：

```
set   XACT_ABORT ON
 begin tran
     insert into SC values (‘s1’, ‘c1’, 90)
     update SC
     set  GRADE=GRADE + 12
  commit tran
  select  *  from  SC
```

- update 操作、insert操作均被回滚，(‘s1’, ‘c1’, 90)无法插入

略 Appendix B Savepoint

- SAVEPOINT 语句定义事务执行过程中的中间点/状态
  - 使用保存点将各组相关语句分开可以在事务内标识重要的状态
- 使用 ROLLBACK TO SAVEPOINT 语句可以撤消该点后的所有更改，即将事务的执行退回到以前的状态

```
BEGIN TRANSACTION
USE   student-DB
INSERT INTO student
    VALUES ( " 03402", "王菲",  "CS", "1985/05/15")
SAVE TRAN  My-savepoint                    /* defining save-point */
DELETE  FROM  student
    WHERE  name= "王菲"  or "章立"
ROLLBACK TRAN  My-savepoint
COMMIT  TRAN
GO
```

Note:

/*rollback将操作滚回到保存点 My-savepoint: delete 操作被***rolled back***, 而 insert操作则不被***rolled back***；

DB恢复为delete操作执行前的状态，新插入的元组 ("03402", "王菲", "CS", "1985/05/15")并未被删除，仍然保存在数据库中*/

Fig.17.0.14 An example of savepoint in T-SQL

st-id	st-name	depart-ment	birth-date
03405	章立	CS	1985/01/25
…	…	…	…
03409	李龙	CS	1984/12/20
…	…	…	…
03411	赵新	CS	1985/06/18

(a) DB before the transaction starts

st-id	st-name	depart-ment	birth-date
03402	王菲	CS	1985/05/15
03405	章立	CS	1985/01/25
…	…	…	…
03409	李龙	CS	1984/12/20
…	…	…	…
03411	赵新	CS	1985/06/18

(b) DB after ***insert*** is issued

Fig.17.0.15-I DB instances at different timepoints

st-id	st-name	depart-ment	birth-date
…	…	…	…
03409	李龙	CS	1984/12/20
…	…	…	…
03411	赵新	CS	1985/06/18

(c) DB after ***delete*** is issued

st-id	st-name	depart-ment	birth-date
03402	王菲	CS	1985/05/15
03405	章立	CS	1985/01/25
…	…	…	…
03409	李龙	CS	1984/12/20
…	…	…	…
03411	赵新	CS	1985/06/18

(d) DB after ***rollback*** is issued

Fig.17.0.15-II DB instances at different timepoints

Have a break