

# Database Management Systems

Lecture 1

Introduction - Transactions

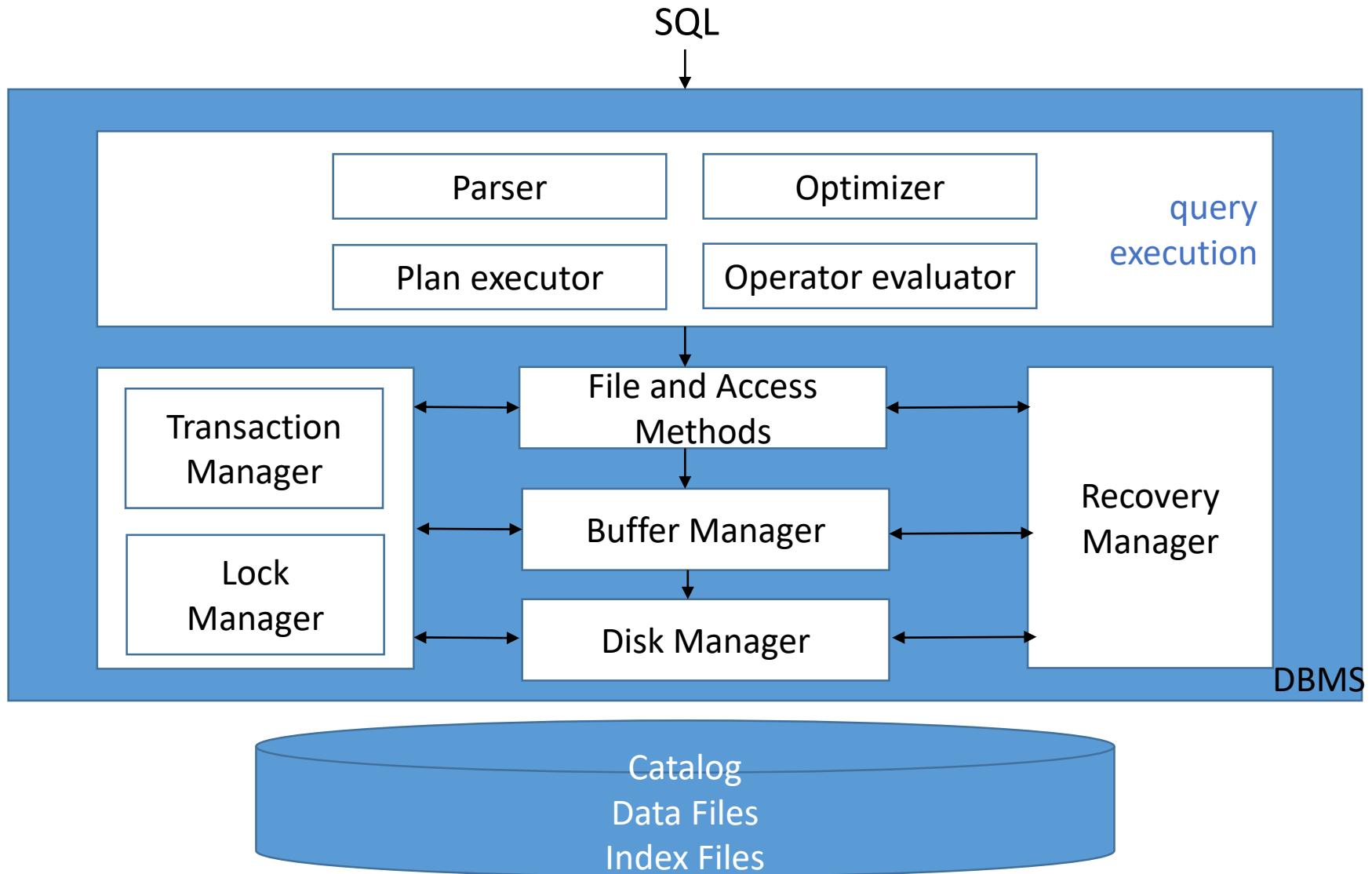
# DBMSs

- Lecture2 + Seminar1 + Laboratory1
- grading
  - written exam (W) - 50%
  - practical exam (P) - 25%
  - labs (L) - 25%
  - W, P  $\geq$  5
- to attend the exam, a student must have at least 6 laboratory attendances and at least 5 seminar attendances, according to the Computer Science Department's decision:
  - <http://www.cs.ubbcluj.ro/wp-content/uploads/Hotarare-CDI-15.03.2017.pdf>
  - <http://www.cs.ubbcluj.ro/~sabina>
  - sabina@cs.ubbcluj.ro

# DBMSs

- lab
  - DBMS: SQL Server
  - application development: .NET / C#
  - data access: ADO.NET

# DBMS Architecture



# Concurrency in a DBMS

- scenarios

1. airline reservation system

- travel agent 1 (TA1) processes customer 1's (C1) request for a seat on flight 10

=> system retrieves from the DB tuple *<flight: 10, available seats: 1>* and displays it to TA1

- C1 takes time to decide whether or not to make the reservation
- travel agent 2 (TA2) processes customer 2's (C2) request for a seat on flight 10

=> system displays the same tuple to TA2: *<flight: 10, available seats: 1>*

- C2 immediately reserves the seat

=> system:

- updates the tuple retrieved by TA2: *<flight: 10, available seats: 0>*

# Concurrency in a DBMS

- scenarios

1. airline reservation system

=> system:

- updates the database, i.e., replaces flight 10's tuple in the DB with the new tuple *<flight: 10, available seats: 0>*
- C1 also decides to reserve the seat
- however, TA1 is still working on the previously retrieved tuple, not knowing another user has changed the data

=> system:

- updates the tuple retrieved by TA1: *<flight: 10, available seats: 0>*
- updates the database, i.e., replaces flight 10's tuple in the DB with the new tuple *<flight: 10, available seats: 0>*

=> the last remaining seat on flight 10 has been sold 2 times!

# Concurrency in a DBMS

- scenarios

## 2. bank system

- 1000 accounts:  $A_1, A_2, \dots, A_{1000}$
- program P1 is computing the total balance:  $\sum_{i=1}^{1000} Balance(A_i)$
- program P2 transfers 500 lei from  $A_1$  to  $A_{1000}$
- if P2's transfer occurs after P1 has “seen”  $A_1$ , but before it has “seen”  $A_{1000}$   
=> the total balance will be incorrect:  $\sum_{i=1}^{1000} Balance(A_i) + 500$
- objective: prevent such anomalies without disallowing concurrent access!

# Concurrency in a DBMS

- what happens when concurrent access is forbidden?
  - consider again scenario 1
  - regard the activity of  $TA_i$  as invoking a copy of the program below:

```
S ← seatsAvailable(flight 10)
if (S > 0) then
    if customer decides to purchase ticket
        S--
        seatsAvailable(flight 10) ← S
    endif
else
    tell customer there are no more seats on flight 10
endif
```

# Concurrency in a DBMS

- what happens when concurrent access is forbidden?
  - if the scheduling policy doesn't allow interleaved execution, TA2's copy of the previous program can begin execution only after TA1's invocation of the program has completed, i.e., after C1 has made the decision and the DB has been updated  
=> if C1 decided to buy the ticket, TA2 sees tuple *<flight: 10, available seats: 0>* => C2 will be correctly informed that flight 10 is sold out

# Concurrency in a DBMS

- what happens when concurrent access is forbidden?
- disallowing concurrent execution can lead to unacceptable delays
  - e.g., while C1 decides whether to make the reservation, travel agent 3 (TA3) receives a request from customer 3 (C3) for a seat on flight 11
  - since C1 and C3 are interested in different flights, their requests can be concurrently processed
  - if concurrent access is not allowed, C3's request can be processed only after C1 has made the decision and TA1's invocation of the program has finished

# Concurrency in a DBMS

- what happens when concurrent access is forbidden?
- obs. when programs are executed serially (i.e., one after the other), results are correct
- however, the DBMS interleaves the operations (reads / writes of database objects) of different user programs, as concurrent execution is fundamental for the performance of the application
- the disk is frequently accessed (and access times are relatively slow) => overlap I/O and CPU activity
- if concurrent execution is allowed, one needs to tackle the effects of interleaving transactions

# Transactions

- database partitioned into items (e.g., *tuple*, *attribute*)
- transaction
  - any *one execution* of a program in a DBMS
  - sequence of one or more operations on a database: *read* / *write* / *commit* / *abort*
  - *commit* occurs if and only if the transaction is not aborted
  - final operation:
    - commit or abort

# Transactions

- $I$  – an item in a DB
- fundamental operations in a transaction  $T$  ( $T$  can be omitted when clear from context):
  - $\text{read}(T, I)$
  - $\text{write}(T, I)$
  - $\text{commit}(T)$ 
    - successful completion:  $T$ 's changes are to become permanent
  - $\text{abort}(T)$ 
    - $T$  is *rolled back*: its changes are undone

# Transaction Properties – ACID

- properties a DBMS must ensure for a transaction in order to maintain data in the presence of concurrent execution / system failures
- atomicity
  - a transaction is atomic
  - either all the operations in the transaction are executed, or none are (*all-or-nothing*)
- consistency
  - a transaction must preserve the consistency of the database after execution, i.e., a transaction is a correct program
- isolation
  - a transaction is protected from the effects of concurrently scheduling other transactions

# Transaction Properties – ACID

- durability
  - committed changes are persisted to the database
  - the effects of a committed transaction should persist even if a system crash occurs before all the changes have been written to disk

# Transaction Properties – Atomicity

- a transaction can commit / abort:
  - *commit* – after finalizing all its operations (successful completion)
  - *abort* – after executing some of its operations; the DBMS can itself abort a transaction
- user's perspective:
  - either all operations in a transaction are executed, or none are
  - the transaction is treated as an indivisible unit of execution
- the DBMS *logs* all transactions' actions, so it can *undo* them if necessary (i.e., undo the actions of incomplete transactions)
- *crash recovery*
  - the process of guaranteeing atomicity in the presence of system crashes

## Transaction Properties – Atomicity

- e.g., consider the following transaction transferring money from AccountA to AccountB:

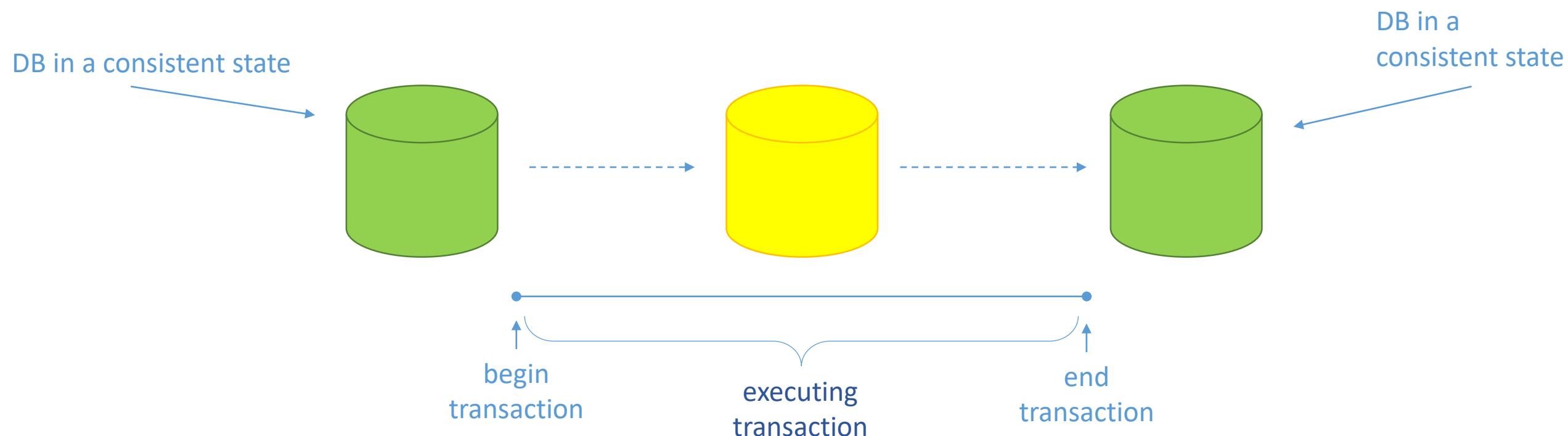
AccountA  $\leftarrow$  AccountA - 100

AccountB  $\leftarrow$  AccountB + 100

- if the transaction fails in the middle, 100 lei must be put back into AccountA, i.e., partial effects are undone

# Transaction Properties – Consistency

- in the absence of other transactions, a transaction that executes on a consistent database state leaves the database in a consistent state



## Transaction Properties – Consistency

- transactions do not violate the integrity constraints (ICs) specified on the database (e.g., restrictions declared via CREATE TABLE statements) and enforced by the DBMS
- however, users can have consistency criteria that go beyond the expressible ICs
- e.g.: consider again the money transfer transaction:

AccountA  $\leftarrow$  AccountA - 100

AccountB  $\leftarrow$  AccountB + 100

- assume the transaction starts execution at time  $t_i$  and completes at  $t_j$
- consistency constraint:  $\text{Total}(\text{AccountA}, \text{AccountB}, t_i) = \text{Total}(\text{AccountA}, \text{AccountB}, t_j)$
- it's the responsibility of the user to write a correct transaction that meets the above constraint

## Transaction Properties – Consistency

- the DBMS does not understand the semantics of the data, of user actions (e.g., how is the interest computed, the fact that AccountB must be credited with the amount of money debited from AccountA, etc)

## Transaction Properties – Isolation

- transactions are protected from the effects of concurrently scheduling other transactions
- users concurrently submit transactions to the DBMS, but can think of their transactions as if they were executing in isolation, independently of other users' transactions
- i.e., users must not deal with arbitrary effects produced by other concurrently running transactions; a transaction is seen as if it were in single-user mode

## Transaction Properties – Durability

- once a transaction commits, the system must guarantee the effects of its operations won't be lost, even if failures subsequently occur
- i.e., once the DBMS informs the user a transaction has committed, its effects should persist, even if a system crash occurs before all changes have been saved on the disk
  - the *Write-Ahead Log* property
    - changes are written to the log (on the disk) before being reflected in the database
    - ensures atomicity, durability

# Interleaved Executions - Example

- consider the two transactions below:

T1: BEGIN	$A \leftarrow A - 100$	$B \leftarrow B + 100$	END
-----------	------------------------	------------------------	-----

T2: BEGIN	$A \leftarrow 1.05 * A$	$B \leftarrow 1.05 * B$	END
-----------	-------------------------	-------------------------	-----

- T1 is transferring 100 lei from account A to account B
- T2 is adding a 5% interest to both accounts
- T1 and T2 are submitted together to the DBMS
- T1 can execute before or after T2 on a database DB, resulting in one of the following database states:
  - State( (T1T2), DB) or
  - State( (T2T1), DB)
- the net effect of executing T1 and T2 *must* be identical to State( (T1 T2), DB) or State( (T2 T1), DB)

## Interleaved Executions - Example

- consider the following interleaving of operations (schedule):

T1	T2
$A \leftarrow A - 100$	$A \leftarrow 1.05 * A$
$B \leftarrow B + 100$	$B \leftarrow 1.05 * B$

✓

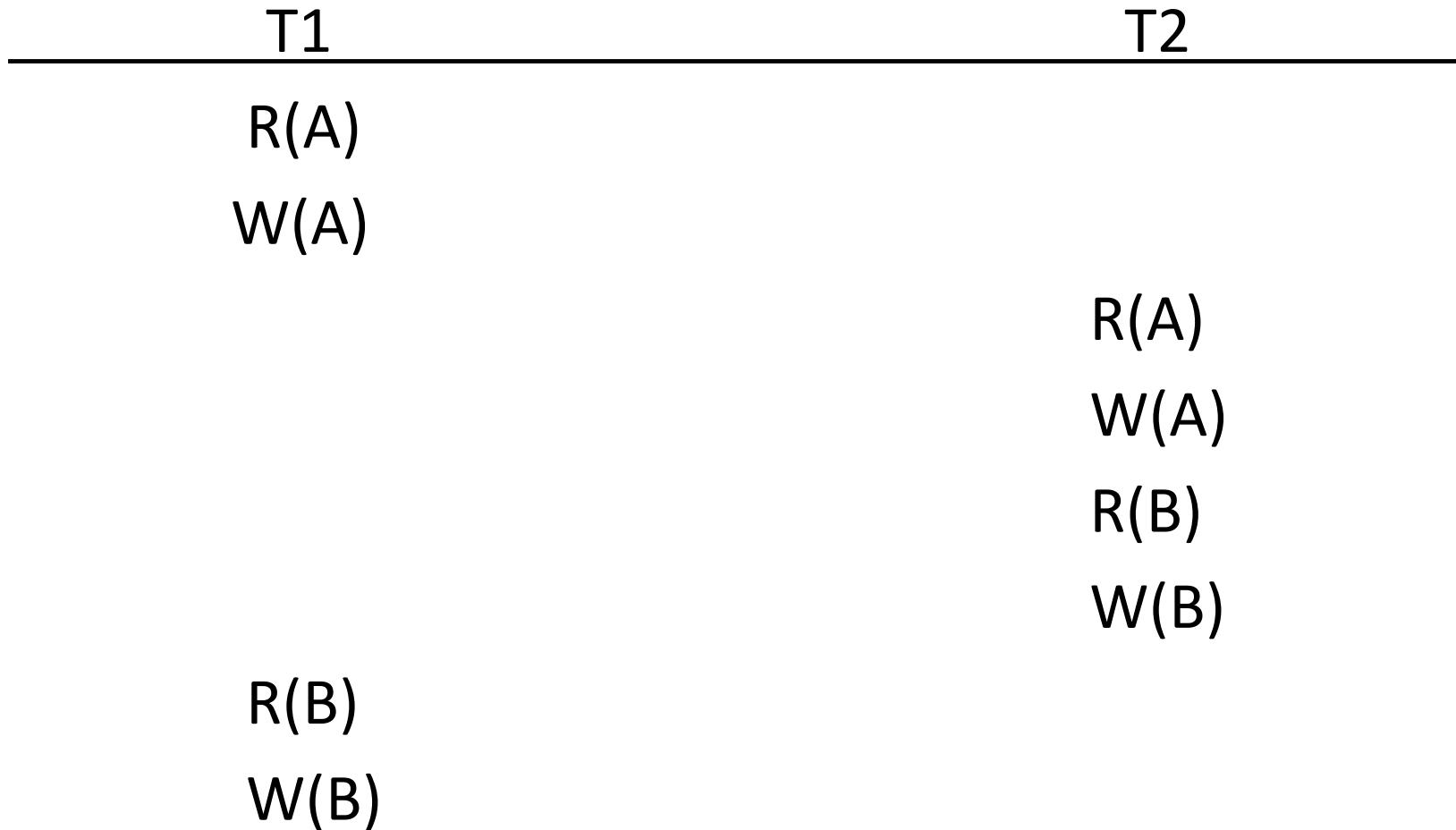
# Interleaved Executions - Example

- but what about the following schedule:



# Interleaved Executions - Example

- the DBMS's view of the second schedule:

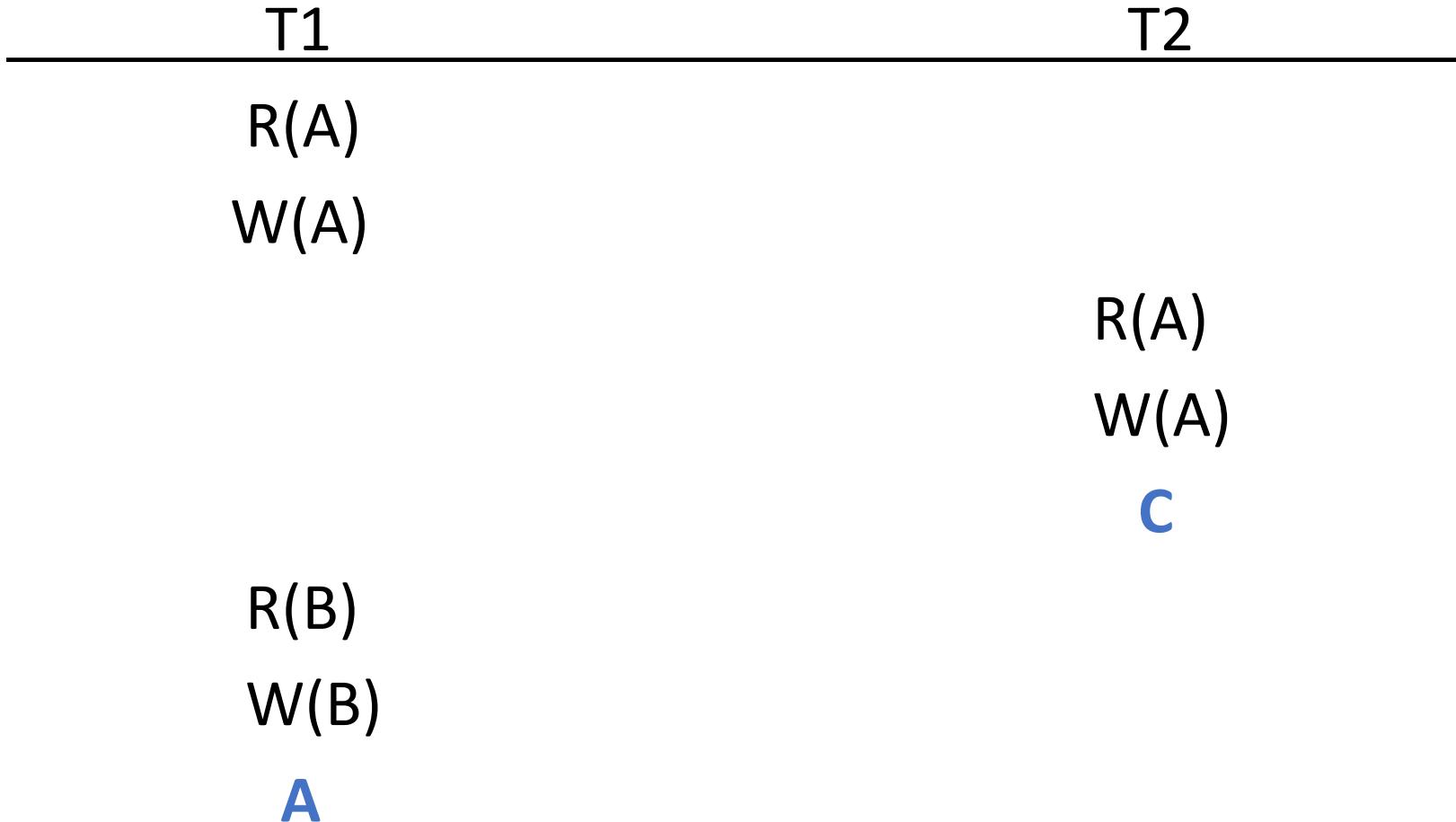


# Interleaved Executions - Anomalies

- two transactions are only reading a data object => no conflict, order of execution not important
- two transactions are reading and / or writing completely separate data objects => no conflict, order of execution not important
- one transaction is writing a data object and another one is either reading or writing the same object => order of execution is important
  - WR conflict
    - T2 is reading a data object previously written by T1
  - RW conflict
    - T2 is writing a data object previously read by T1
  - WW conflict
    - T2 is writing a data object previously written by T1

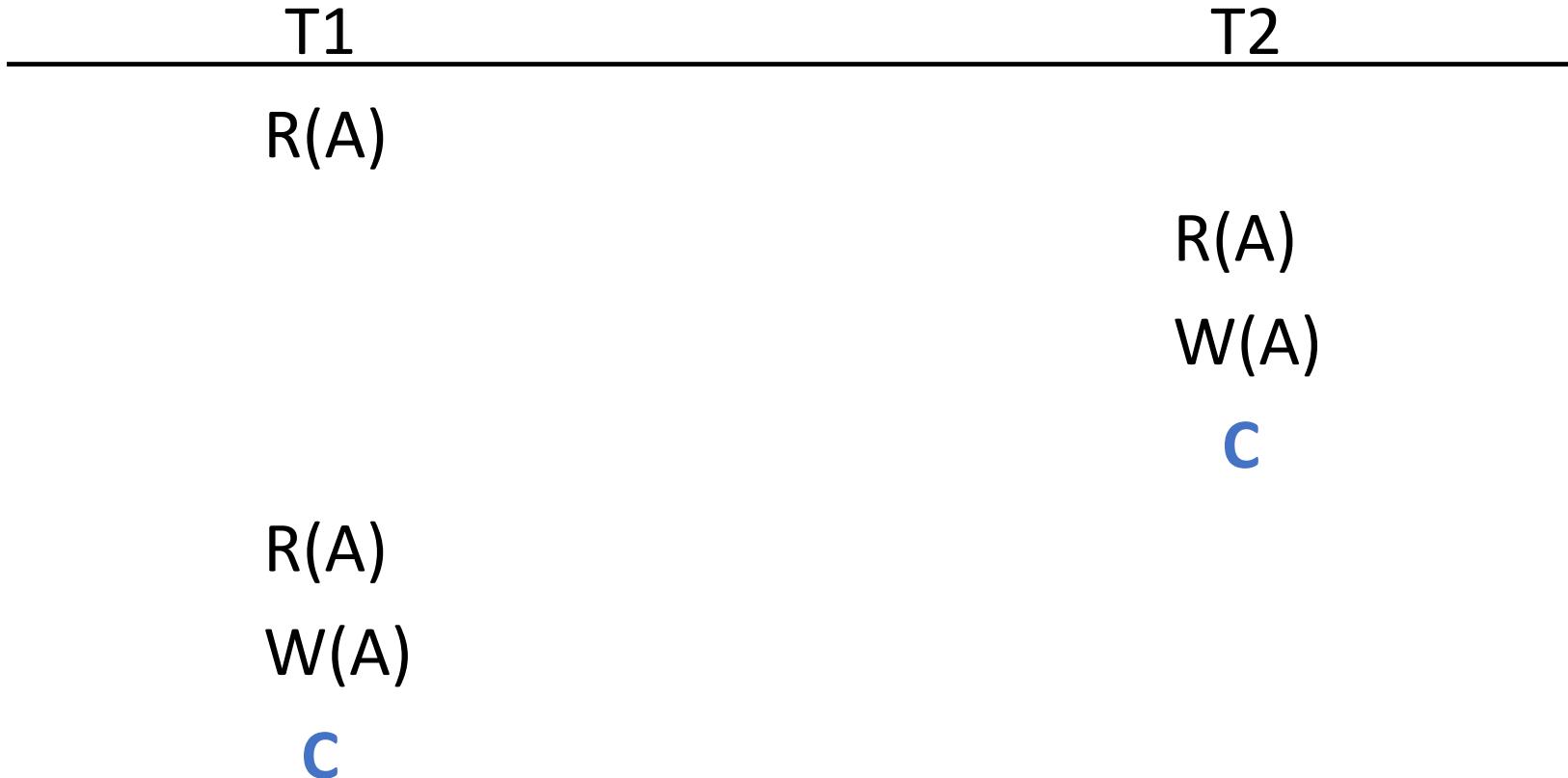
# Interleaved Executions - Anomalies

- reading uncommitted data (WR conflicts, *dirty reads*)



# Interleaved Executions - Anomalies

- unrepeatable reads (RW conflicts)



# Interleaved Executions - Anomalies

- overwriting uncommitted data (WW conflicts):



# Scheduling Transactions

- schedule
  - a list of operations (Read / Write / Commit / Abort) of a set of transactions with the property that the order of the operations in each individual transaction is preserved

# Scheduling Transactions

T1

read(V)

read(sum)

read(V)

sum := sum + V

write(sum)

commit

T2

read(V)

V := V + 50

write(V)

commit

Schedule

read1(V)

read1(sum)

read2(V)

write2(V)

commit2

read1(V)

write1(sum)

commit1

# Serial and Non-Serial Schedules

- serial schedule

- a schedule in which the actions of different transactions are not interleaved

T1

read(V)  
read(sum)  
read(V)  
 $\text{sum} := \text{sum} + V$   
write(sum)  
commit

T2

read(V)  
 $V := V + 50$   
write(V)  
commit

# Serial and Non-Serial Schedules

- non-serial schedule

- a schedule in which the actions of different transactions are interleaved

read1(V)

read1(sum)

read2(V)

write2(V)

commit2

read1(V)

write1(sum)

commit1

# Serializability

- $C$  – set of transactions
- $Sch(C)$  – the set of schedules for  $C$
- *Serializable schedule*
  - a schedule  $S \in Sch(C)$  is serializable if the effect of  $S$  on a consistent database instance is identical to the effect of some serial schedule  $S_0 \in Sch(C)$

# Serializability

- serializability
  - correctness criterion for an interleaved execution schedule
    - consider the serial execution schedule  $(T_1, T_2, \dots, T_n)$ ,  $T_i \in C$
    - assume the database instance is in a correct state prior to executing  $T_1$
    - every transaction must preserve the consistency of the database
  - => the database is in a correct state after  $T_n$  completes execution
  - => if a serializable schedule is executed on a correct database instance, it produces a correct database instance (since it is equivalent to some serial schedule)
  - ensuring serializability prevents inconsistencies generated by concurrent transactions that interfere with one another

# Serializability

- objective
  - finding interleaved schedules enabling transactions to execute concurrently without interfering with each other (such schedules result in a correct database state)
- the order of *read* and *write* operations is important
- actions that cannot be swapped in a schedule:
  - actions belonging to the same transaction
  - actions in different transactions operating on the same object if at least one of them is a *write*

## References

- [Ra00] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (2<sup>nd</sup> Edition), McGraw-Hill, 2000
- [Le99] LEVENE, M., LOIZOU, G., A Guided Tour of Relational Databases and Beyond, Springer, 1999
- [Ga08] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book, Prentice Hall Press, 2008
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007,  
<http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Si10] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, McGraw-Hill, 2010, <http://codex.cs.yale.edu/avi/db-book/>
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems,  
<http://infolab.stanford.edu/~ullman/fcdb.html>

# Database Management Systems

Lecture 2

Transactions. Concurrency Control

## Conflict Serializability

- $C$  – set of transactions
- $Sch(C)$  – the set of schedules for  $C$
- $Op(C)$  – set of operations of the transactions in  $C$
- consider schedule  $S \in Sch(C)$
- the *conflict relation* of  $S$  is defined as:
  - $conflict(S) = \{(op_1, op_2) \mid op_1, op_2 \in Op(C), op_1 \text{ occurs before } op_2 \text{ in } S, op_1 \text{ and } op_2 \text{ are in conflict}\}$

## Conflict Serializability

- $C$  – set of transactions
- $Sch(C)$  – the set of schedules for  $C$
- two schedules  $S_1$  and  $S_2 \in Sch(C)$  are conflict equivalent, written  $S_1 \equiv_c S_2$ , if  $conflict(S_1) = conflict(S_2)$ , i.e.:
  - $S_1$  and  $S_2$  contain the same operations of the same transactions and
  - every pair of conflicting operations is ordered in the same manner in  $S_1$  and  $S_2$

# Conflict Serializability

S1

T1	T2
Read(A)	
A := A - 100	
Write(A)	
Read(B)	
B := B + 200	
Write(B)	
	Read(A)
	A := A * 0.2
	Write(A)
	Read(B)
	B := B + 300
	Write(B)

S2

T1	T2
Read(A)	
A := A - 100	
Write(A)	
	Read(A)
	A := A * 0.2
	Write(A)
Read(B)	
B := B + 200	
Write(B)	
	Read(B)
	B := B + 300
	Write(B)

conflict(S1) = conflict(S2)  
=> S1  $\equiv_c$  S2

conf(S1) = {(Read(T1, A), Write(T2, A)), (Write(T1, A), Read(T2, A)), (Write(T1, A), Write(T2, A)),  
(Read(T1, B), Write(T2, B)), (Write(T1, B), Read(T2, B)), (Write(T1, B), Write(T2, B))}

# Conflict Serializability

S1

T1	T2
Read(A)	
A := A - 100	
Write(A)	
Read(B)	
B := B + 200	
Write(B)	
	Read(A)
	A := A * 0.2
	Write(A)
	Read(B)
	B := B + 300
	Write(B)

S3

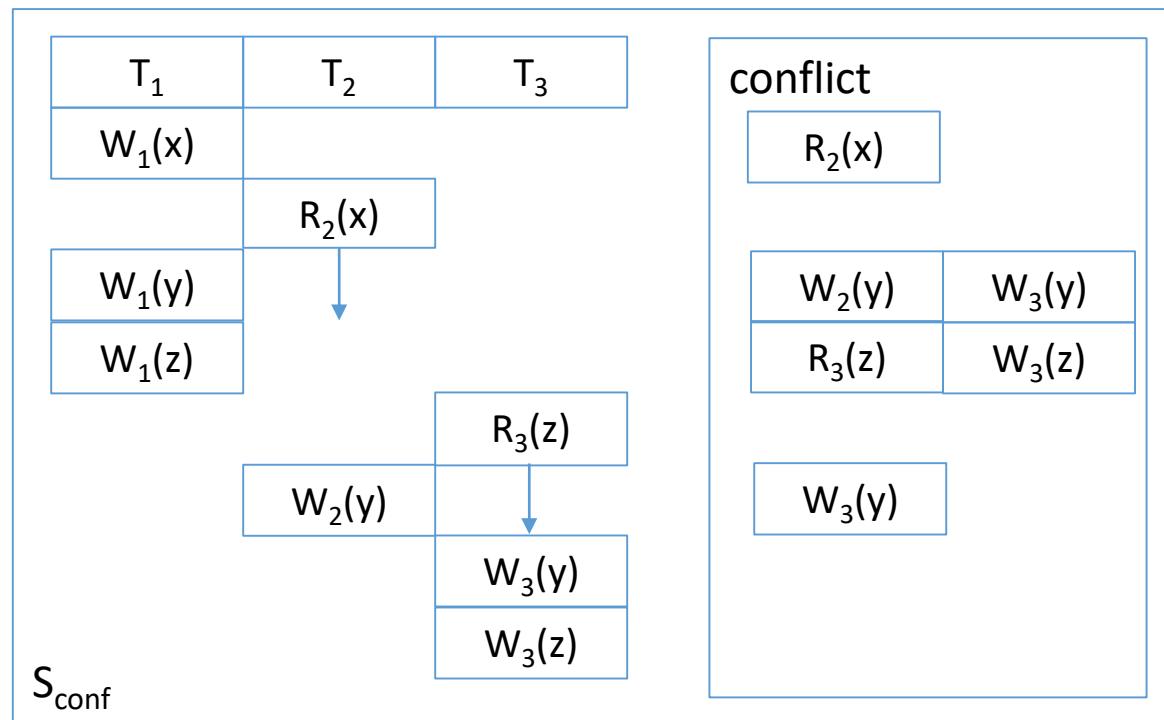
T1	T2
	Read(A)
	A := A * 0.2
	Write(A)
	Read(B)
	B := B + 300
	Write(B)
Read(A)	
A := A - 100	
Write(A)	
Read(B)	
B := B + 200	
Write(B)	

$\text{conflict}(S1) \neq \text{conflict}(S3)$   
 $\Rightarrow S1 \not\equiv_c S3$

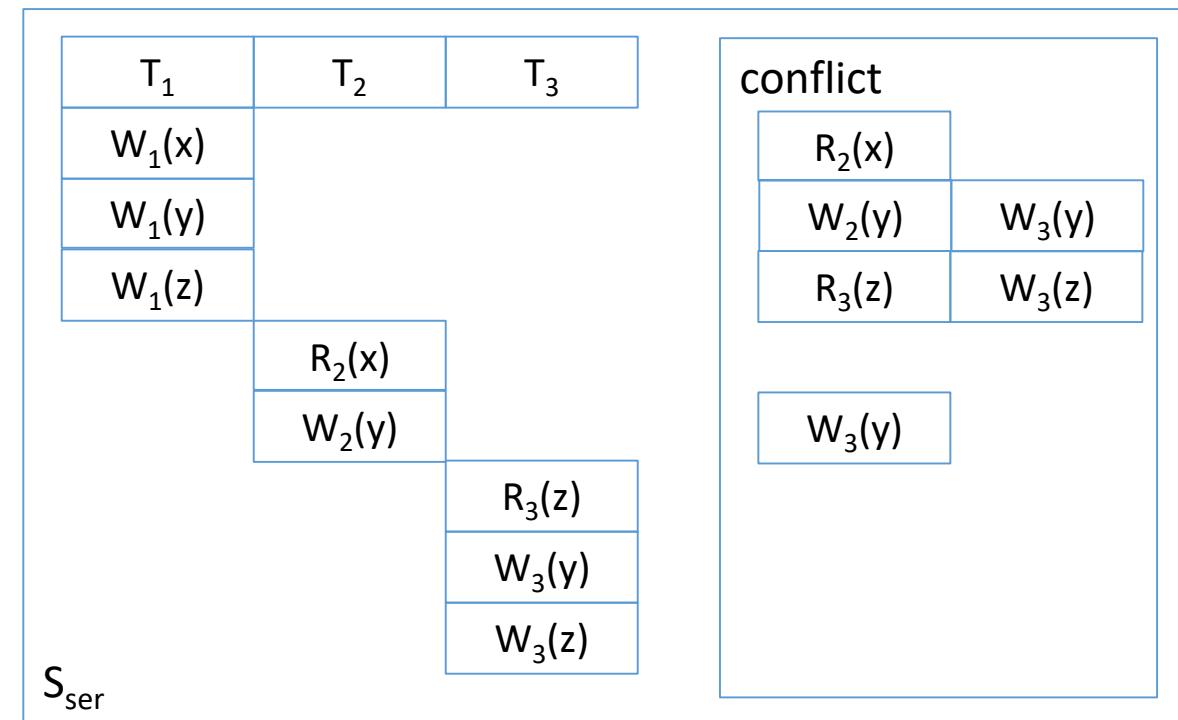
## Conflict Serializability

- $C$  – set of transactions
- $Sch(C)$  – the set of schedules for  $C$
- let  $S$  be a schedule in  $Sch(C)$
- schedule  $S$  is conflict serializable if there exists a serial schedule  $S_0 \in Sch(C)$  such that  $S \equiv_c S_0$ , i.e.,  $S$  is conflict equivalent to some serial schedule

# Conflict Serializability



conflict serializable schedule



serial schedule

## Conflict Serializability - Precedence Graph

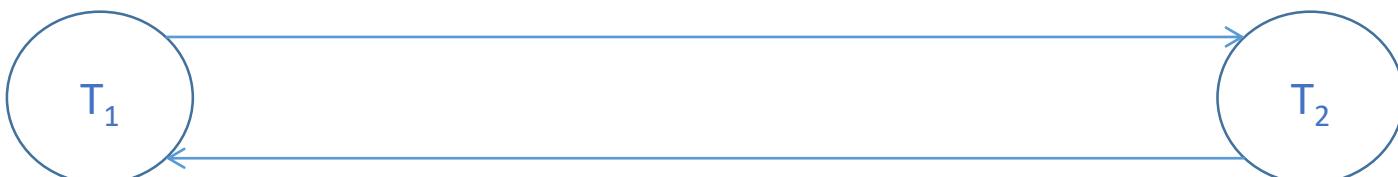
- let  $S$  be a schedule in  $Sch(C)$
- the precedence graph (serializability graph) of  $S$  contains:
  - one node for every committed transaction in  $S$
  - an arc from  $T_i$  to  $T_j$  if an action in  $T_i$  precedes and conflicts with one of the actions in  $T_j$
- Theorem:
  - a schedule  $S \in Sch(C)$  is conflict serializable if and only if its precedence graph is acyclic

## Conflict Serializability - Precedence Graph

- example - a schedule that is not conflict serializable:



- the precedence graph has a cycle:

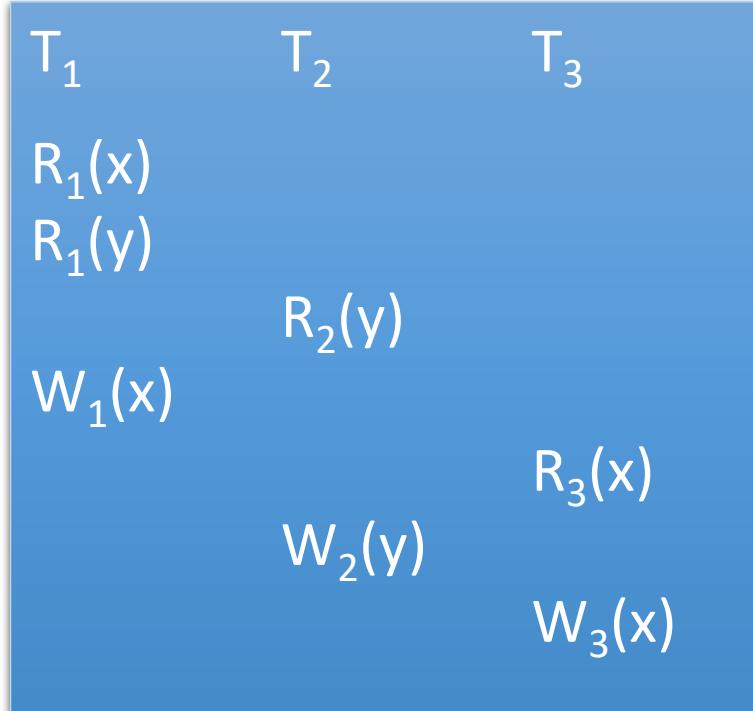


## Conflict Serializability - Precedence Graph

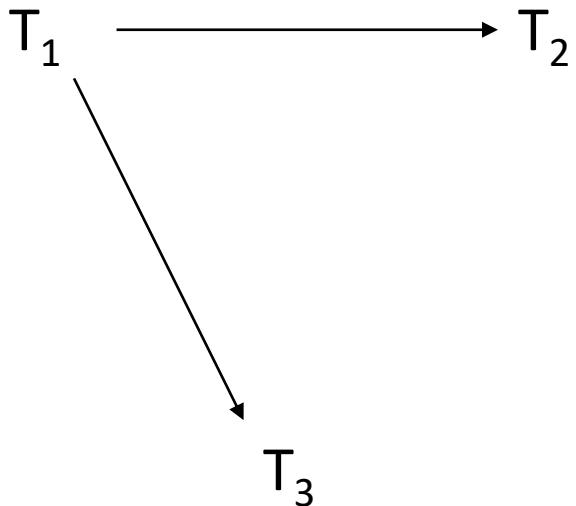
- algorithm to test the conflict serializability of a schedule  $S \in Sch(C)$ 
  1. create a node labeled  $T_i$  in the precedence graph for every committed transaction  $T_i$  in the schedule
  2. create an arc  $(T_i, T_j)$  in the precedence graph if  $T_j$  executes a  $\text{Read}(A)$  after a  $\text{Write}(A)$  executed by  $T_i$
  3. create an arc  $(T_i, T_j)$  in the precedence graph if  $T_j$  executes a  $\text{Write}(A)$  after a  $\text{Read}(A)$  executed by  $T_i$
  4. create an arc  $(T_i, T_j)$  in the precedence graph if  $T_j$  executes a  $\text{Write}(A)$  after a  $\text{Write}(A)$  executed by  $T_i$
  5.  $S$  is conflict serializable if and only if the resulting precedence graph has no cycles

## Conflict Serializability - Precedence Graph

- examples
- let  $S_1$  be a schedule over  $\{T_1, T_2, T_3\}$ :

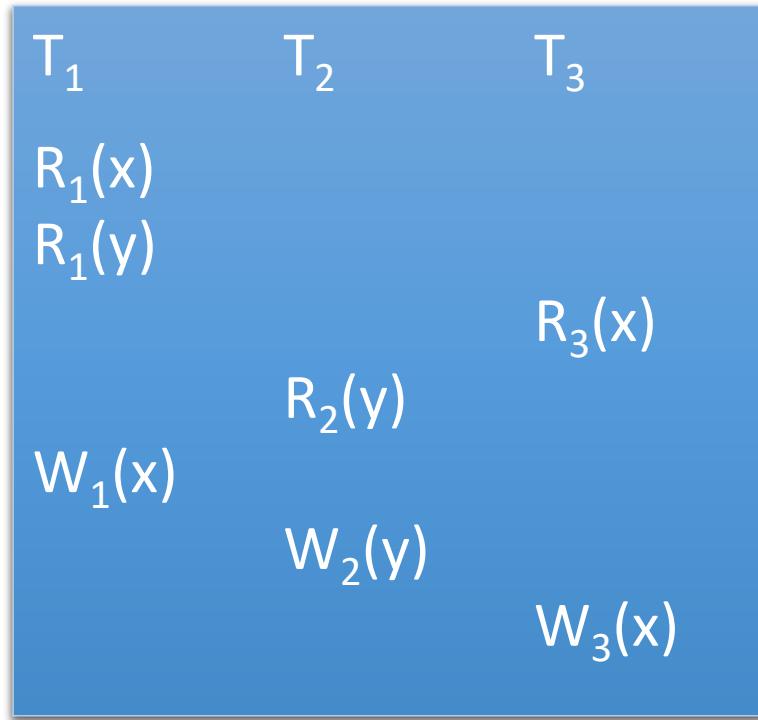


- the precedence graph for  $S_1$ :

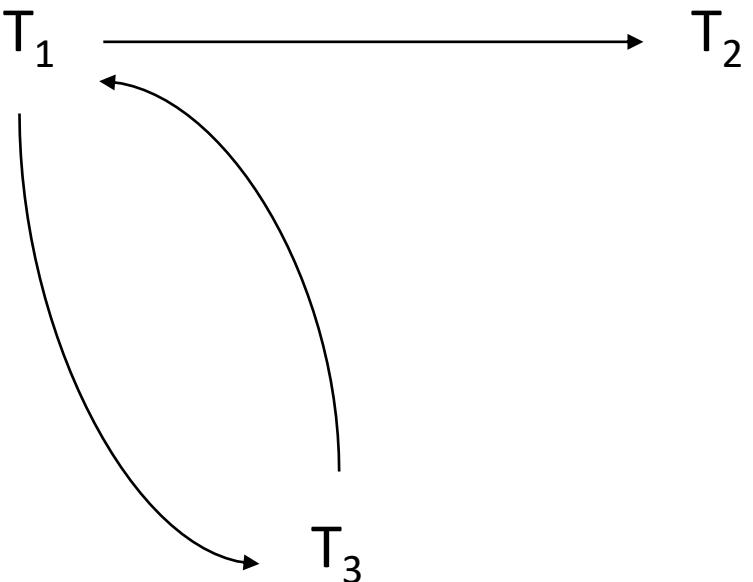


## Conflict Serializability - Precedence Graph

- examples
- let  $S_2$  be a schedule over  $\{T_1, T_2, T_3\}$ :

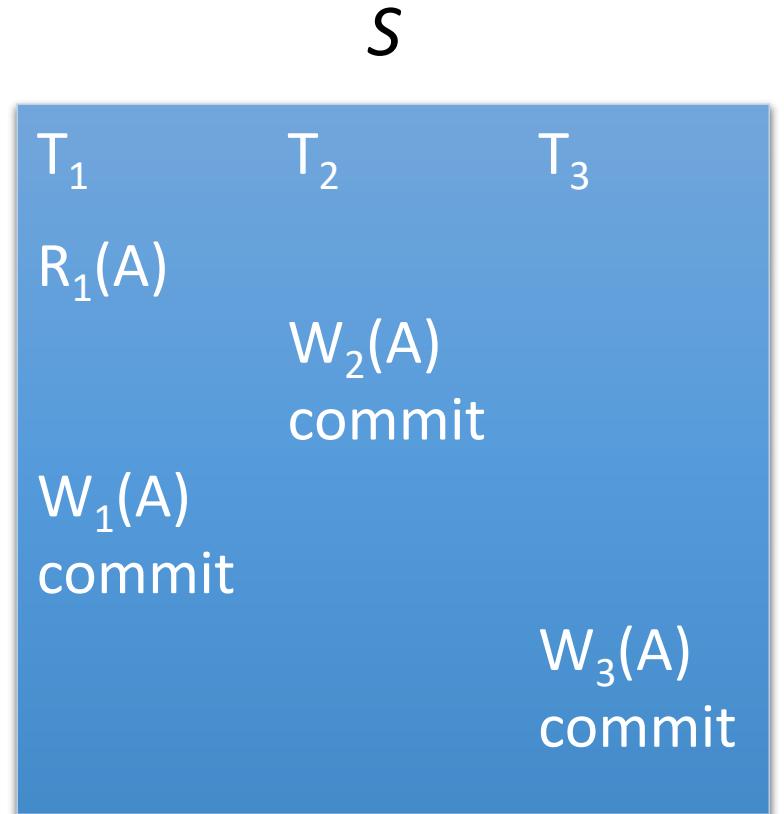


- the precedence graph for  $S_2$ :



## Conflict Serializability

- every conflict serializable schedule is serializable (in the absence of inserts / deletes, when items can only be updated)
- there are serializable schedules that are not conflict serializable
- $S$  is equivalent to the serial execution of transactions  $T_1, T_2, T_3$  (in this order), but it is not conflict equivalent to this serial schedule (the write operations in  $T_1$  and  $T_2$  are ordered differently)



## View Serializability

- conflict serializability is a sufficient condition for serializability, but it is not a necessary one
- *view serializability*
  - a more general, sufficient condition for serializability
  - based on *view-equivalence*, a less stringent form of equivalence

## View Serializability

- $C$  – set of transactions
- $Sch(C)$  – the set of schedules for  $C$
- let  $T_i, T_j \in C, S_1, S_2 \in Sch(C)$ ;  $S_1$  and  $S_2$  are view equivalent, written  $S_1 \equiv_v S_2$ , if the following conditions are met:
  - if  $T_i$  reads the initial value of  $V$  in  $S_1$ , then  $T_i$  also reads the initial value of  $V$  in  $S_2$ ;
  - if  $T_i$  reads the value of  $V$  written by  $T_j$  in  $S_1$ , then  $T_i$  also reads the value of  $V$  written by  $T_j$  in  $S_2$ ;
  - if  $T_i$  writes the final value of  $V$  in  $S_1$ , then  $T_i$  also writes the final value of  $V$  in  $S_2$ .
- i.e.:
  - each transaction performs the same computation in  $S_1$  and  $S_2$  and
  - $S_1$  and  $S_2$  produce the same final database state.

# View Serializability

S1

T1	T2
Read(A)	
A := A - 100	
Write(A)	
Read(B)	
B := B + 200	
Write(B)	
	Read(A)
	A := A * 0.2
	Write(A)
	Read(B)
	B := B + 300
	Write(B)

S2

T1	T2
Read(A)	
A := A - 100	
Write(A)	
	Read(A)
	A := A * 0.2
	Write(A)
Read(B)	
B := B + 200	
Write(B)	
	Read(B)
	B := B + 300
	Write(B)

$S1 \equiv_v S2$

# View Serializability

S1

T1	T2
Read(A)	
A := A - 100	
Write(A)	
Read(B)	
B := B + 200	
Write(B)	
	Read(A)
	A := A * 0.2
	Write(A)
	Read(B)
	B := B + 300
	Write(B)

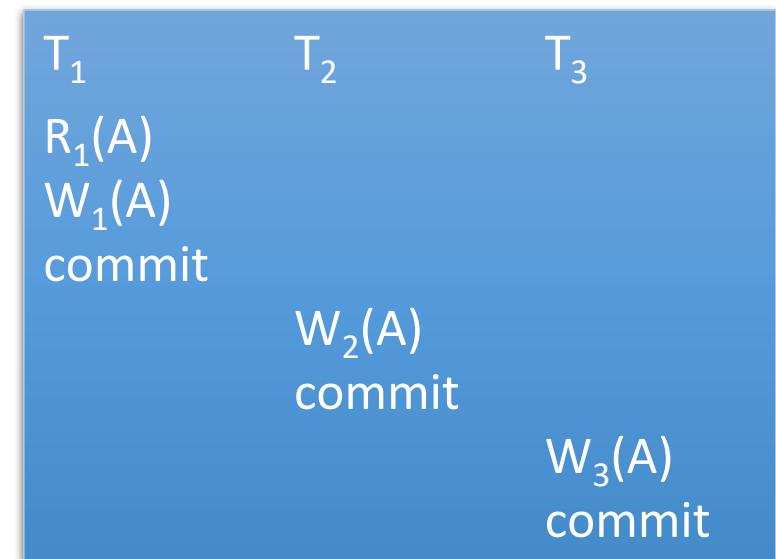
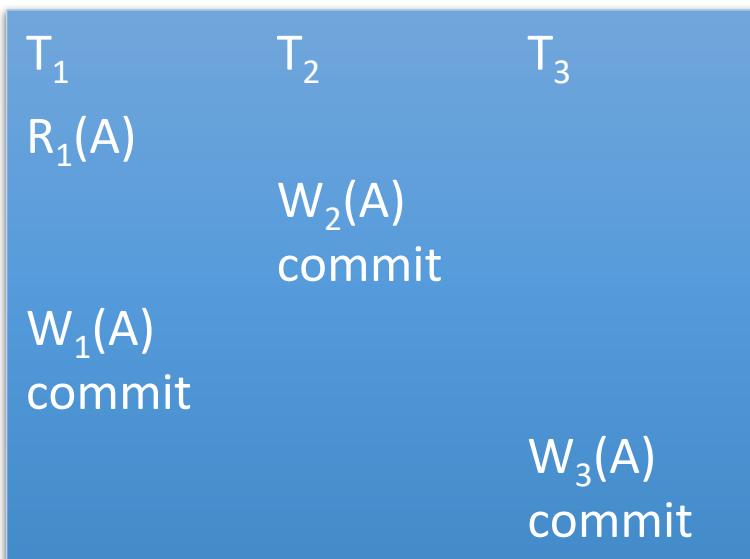
S3

T1	T2
	Read(A)
	A := A * 0.2
	Write(A)
	Read(B)
	B := B + 300
	Write(B)
Read(A)	
A := A - 100	
Write(A)	
Read(B)	
B := B + 200	
Write(B)	

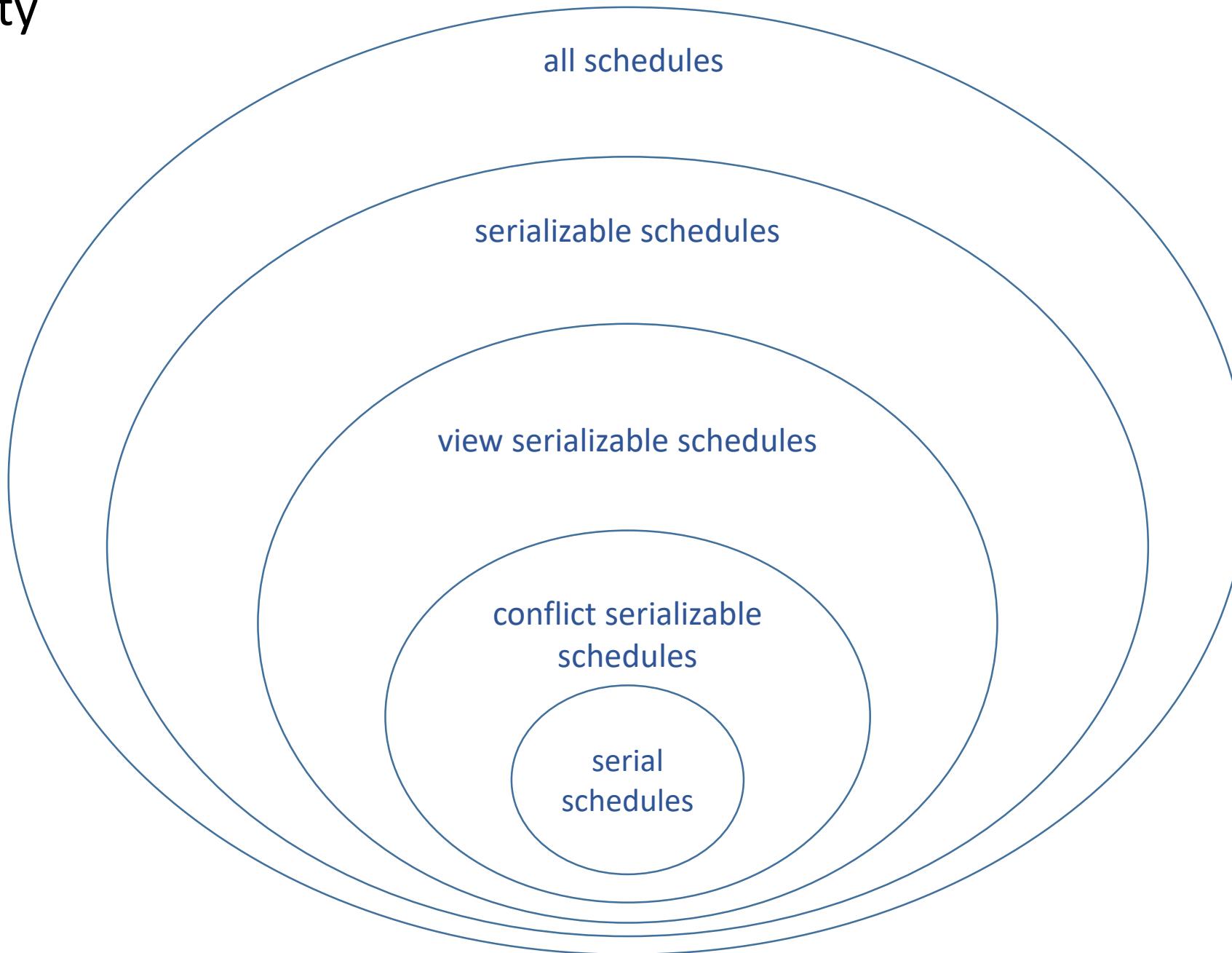
$S1 \not\equiv_v S3$

## View Serializability

- $C$  – set of transactions
- $Sch(C)$  – the set of schedules for  $C$
- a schedule  $S \in Sch(C)$  is view serializable if there exists a serial schedule  $S_0 \in Sch(C)$  such that  $S \equiv_v S_0$ , i.e.,  $S$  is view equivalent to some serial schedule

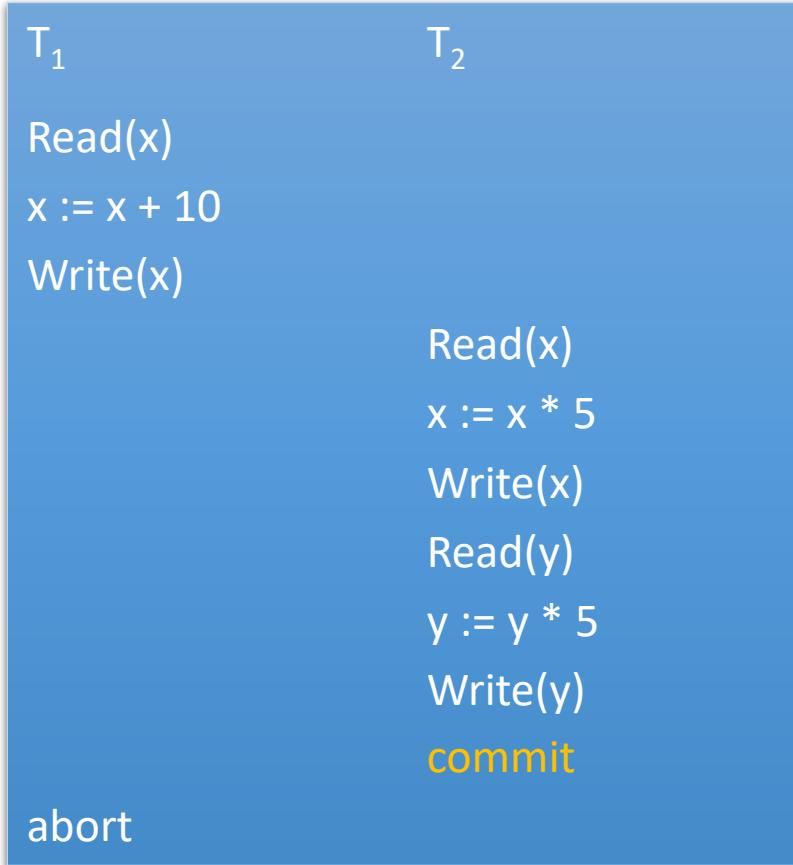


# Serializability



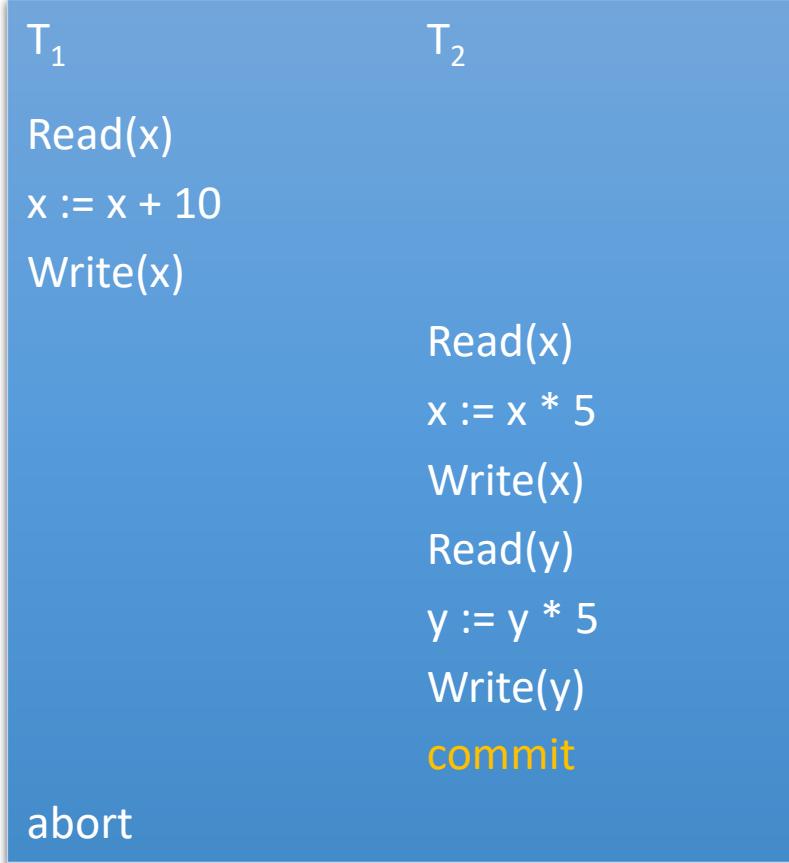
## Recoverable Schedules

- consider schedule  $S$  over  $\{T_1, T_2\}$



- $T_2$  operates on a value of  $x$  that shouldn't have been in the database, since  $T_1$  aborted

# Recoverable Schedules

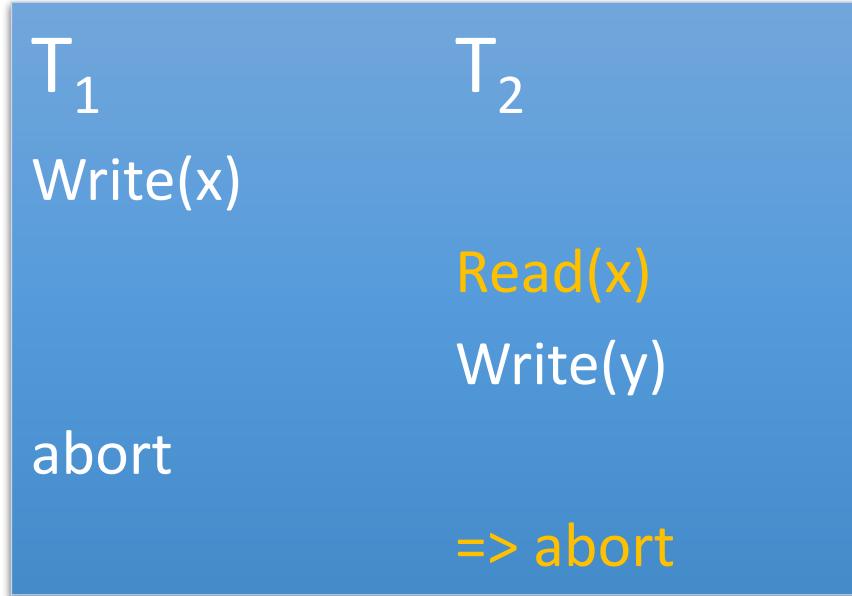


- cannot cascade the abort of  $T_1$ , since  $T_2$  has already committed
- schedule  $S$  is *unrecoverable*

## Recoverable Schedules

- recoverable schedule
  - a schedule in which a transaction T commits only after all transactions whose changes T read commit

# Avoiding Cascading Aborts



- a schedule in which a transaction  $T$  is reading only changes of committed transactions is said to avoid cascading aborts
- avoiding cascading aborts  $\Rightarrow$  recoverable schedules

## Lock-Based Concurrency Control

- technique used to guarantee serializable, recoverable schedules
- *lock*
  - a tool used by the transaction manager to control concurrent access to data
  - prevents a transaction from accessing a data object while another transaction is accessing the object
- *transaction protocol*
  - a set of rules enforced by the transaction manager and obeyed by all transactions
  - example – simple protocol: before a transaction can read / write an object, it must acquire an appropriate lock on the object
  - locks in conjunction with transaction protocols allow interleaved executions

## Lock-Based Concurrency Control

- *transaction protocol*
  - it's impractical for the DBMS to test the serializability of schedules, since the operating system could determine the interleaving of operations
  - instead, the DBMS uses protocols known to produce serializable schedules

## Lock-Based Concurrency Control

- SLock (*shared or read lock*)
  - if a transaction holds an SLock on an object, it can read the object, but it cannot modify it
- XLock (*exclusive or write lock*)
  - if a transaction holds an XLock on an object, it can both read and write the object
- if a transaction holds an SLock on an object, other transactions can be granted SLocks on the object, but they cannot acquire XLocks on it
- if a transaction holds an XLock on an object, other transactions cannot be granted either SLocks or XLocks on the object

	Shared	Exclusive
Shared	Yes	No
Exclusive	No	No

## Lock-Based Concurrency Control

- lock upgrade
  - an SLock granted to a transaction can be upgraded to an XLock
- transactions are issuing lock requests to the lock manager
- locks are held until being explicitly released by transactions
- lock acquire / lock release requests are automatically inserted into transactions by the DBMS (not the user's responsibility)
- locking / unlocking
  - atomic operations

# Lock-Based Concurrency Control

- *lock table*
  - structure used by the lock manager to keep track of granted locks / lock requests
  - entry in the lock table (corresponding to one data object):
    - number of transactions holding a lock on the data object
    - lock type (SLock / XLock)
    - pointer to a queue of lock requests

## Lock-Based Concurrency Control

- *transactions table*
  - structure maintained by the DBMS
  - one entry / transaction
  - keeps a list of locks held by every transaction

## References

- [Ra00] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (2<sup>nd</sup> Edition), McGraw-Hill, 2000
- [Le99] LEVENE, M., LOIZOU, G., A Guided Tour of Relational Databases and Beyond, Springer, 1999
- [Ga08] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book, Prentice Hall Press, 2008
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007,  
<http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Si10] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, McGraw-Hill, 2010, <http://codex.cs.yale.edu/avi/db-book/>
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems,  
<http://infolab.stanford.edu/~ullman/fcdb.html>

# Database Management Systems

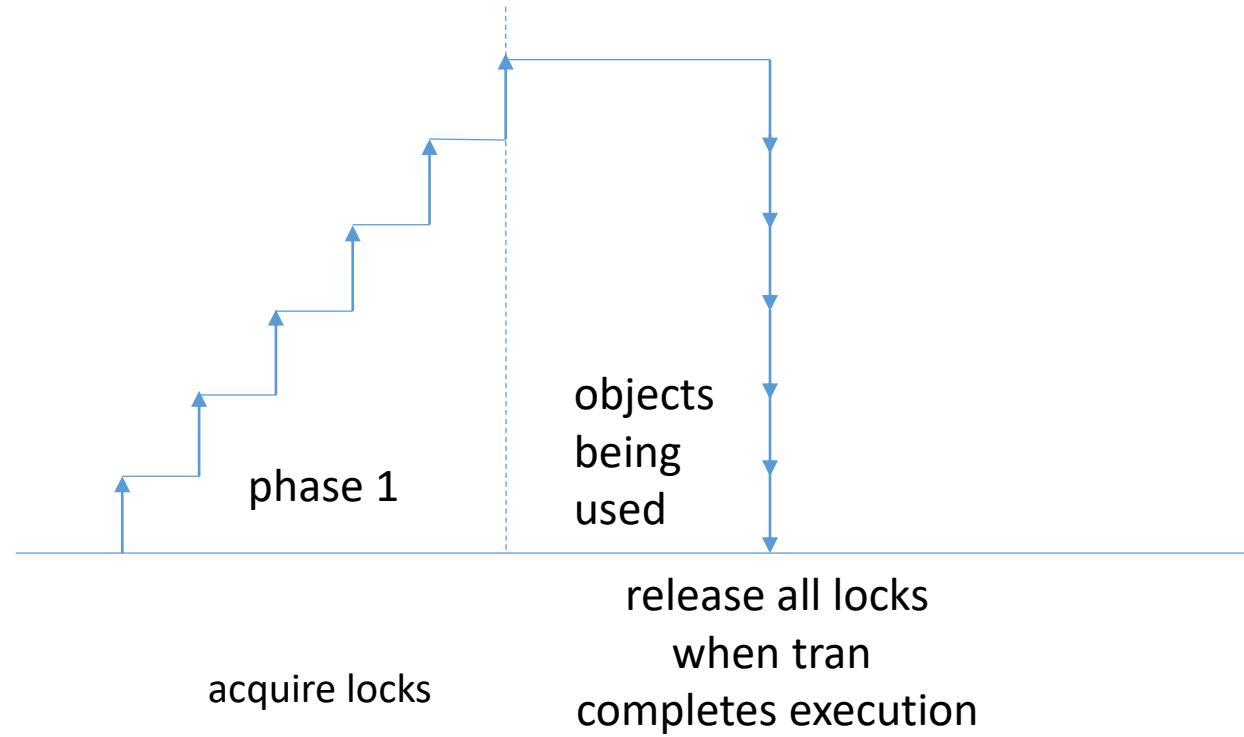
Lecture 3

Transactions. Concurrency Control

- locking *protocols*
  - Strict Two-Phase Locking
  - Two-Phase Locking
- *deadlocks*
  - prevention (Wait-die, Wound-wait)
  - detection (waits-for graph, timeout mechanism)
- the *phantom* problem
- *isolation levels*
  - read uncommitted
  - read committed
  - repeatable read
  - serializable

## Strict Two-Phase Locking (*Strict 2PL*)

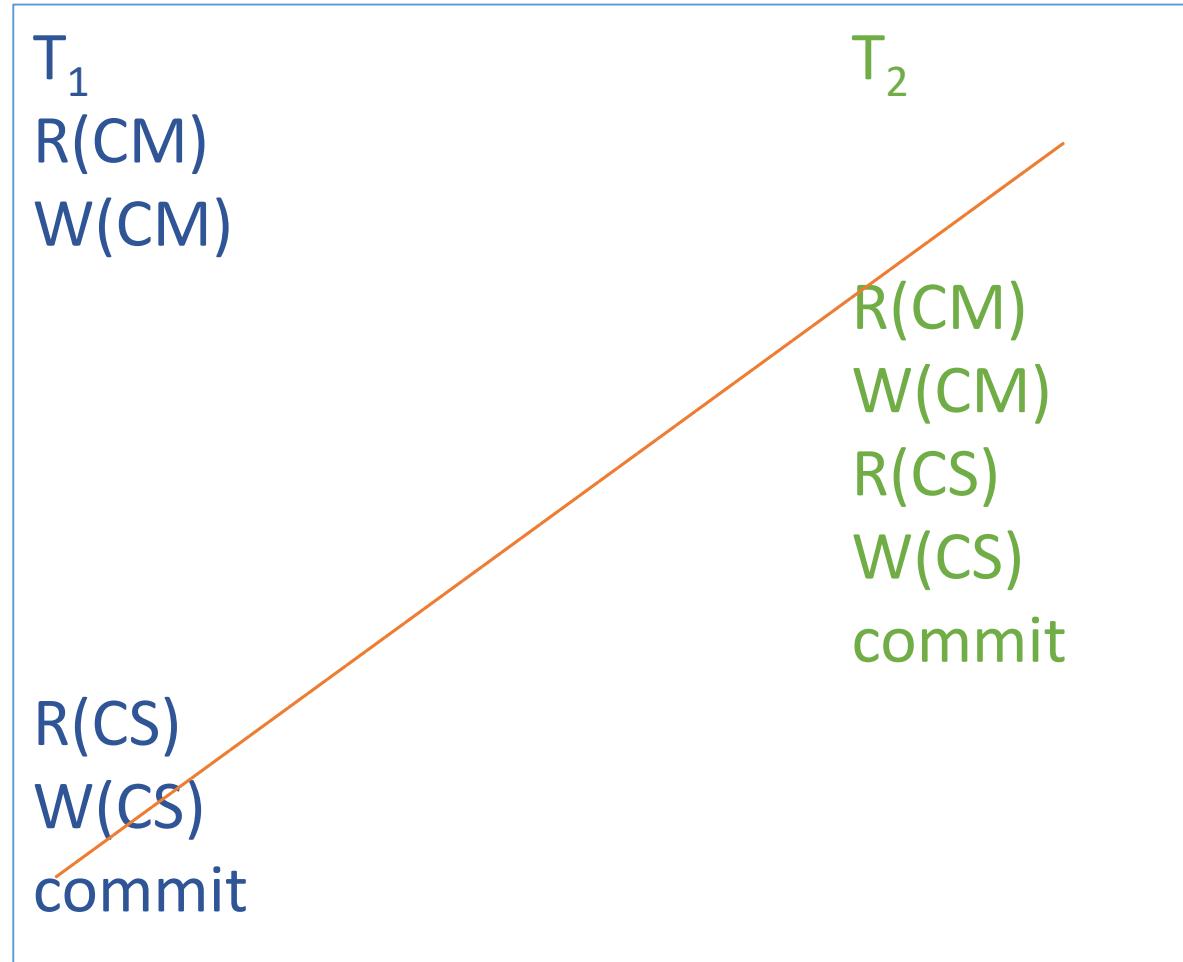
- \* before a transaction can read / write an object, it must acquire a S / X lock on the object
- \* all the locks held by a transaction are released when it completes execution



- the Strict 2PL protocol allows only serializable schedules (only schedules with acyclic precedence graphs are allowed by this protocol)

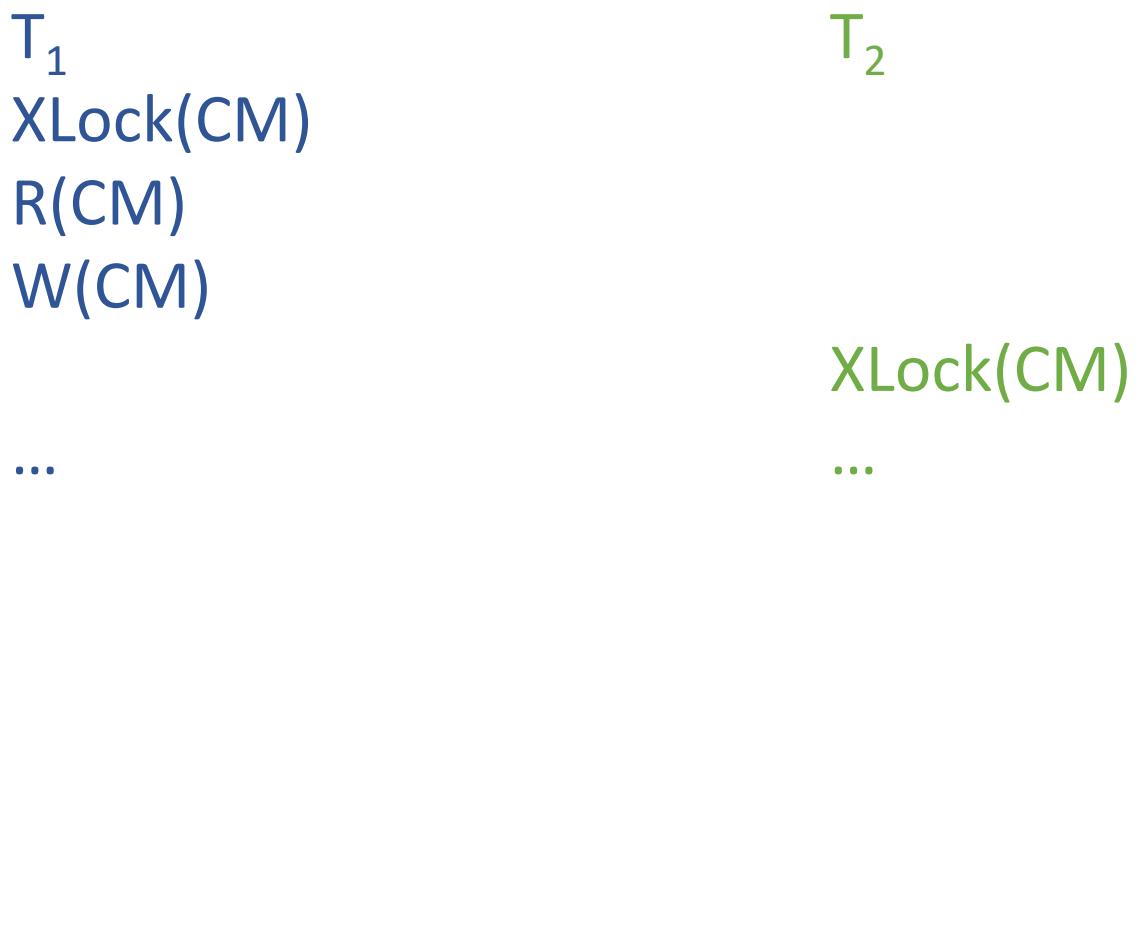
## Strict Two-Phase Locking

- the interleaving below is not allowed by Strict 2PL:



->

## Strict Two-Phase Locking



- $T_1$  acquires an X lock on object CM, reads and writes CM
- $T_1$  is still in progress when  $T_2$  requests a lock on the same object, CM
- $T_2$  cannot acquire an exclusive lock on CM, since  $T_1$  already holds a conflicting lock on this object
- $T_1$  will release its lock on CM only when it completes execution (with *commit* or *abort*)
- since it cannot grant  $T_2$  the requested lock on CM, the DBMS suspends  $T_2$

- in this example, we denote by  $XLock(O)$  the action of the current transaction requesting an X lock on object O

## Strict Two-Phase Locking

T<sub>1</sub>

XLock(CM)

R(CM)

W(CM)

XLock(CS)

R(CS)

W(CS)

commit

T<sub>2</sub>

XLock(CM)

R(CM)

W(CM)

XLock(CS)

R(CS)

W(CS)

commit

- T1 continues execution
- when T1 commits, it releases both locks (X lock on CM, X lock on CS)
- T2 can now be granted an X lock on CM
- T2 can now proceed

## Strict Two-Phase Locking

- the interleaving below is allowed by Strict 2PL:

T<sub>1</sub>  
XLock(CM)  
R(CM)  
W(CM)

T<sub>2</sub>  
XLock(CT)  
R(CT)  
W(CT)  
commit

XLock(CS)  
R(CS)  
W(CS)  
commit

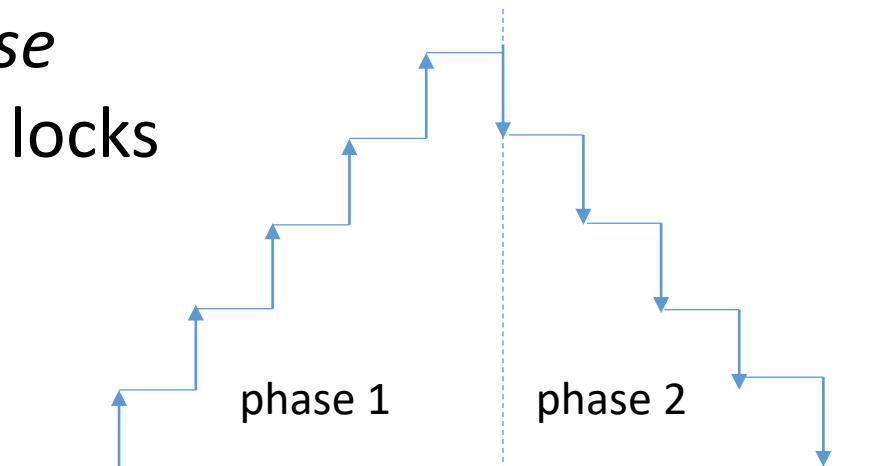
- in this example, since T1 and T2 are operating on separate data objects (CM, CT, CS), they can concurrently obtain all requested locks (same would be true if T1 and T2 had, say, read the same data object A)

## Two-Phase Locking (2PL)

- variant of Strict Two-Phase Locking

- \* before a transaction can read / write an object, it must acquire a S / X lock on the object
- \* once a transaction releases a lock, it cannot request other locks

- phase 1 - *growing phase*
  - transaction acquires locks
- phase 2 - *shrinking phase*
  - transaction releases locks



## Two-Phase Locking

- $C$  – set of transactions
- $Sch(C)$  – set of schedules for  $C$
- if all transactions in  $C$  obey 2PL, then any schedule  $S \in Sch(C)$  that completes normally is serializable

## Two-Phase Locking

- the following execution is allowed by the protocol:
- T1 can release its X lock on A prior to completion
- so T2 can acquire an X lock on A while T1 is still in progress
- however, T1 cannot acquire any other locks once it released a lock (in this case, its X lock on A)
- notation: *Release(O)* - the current transaction releases its lock on object O

T1

XLock(A)

XLock(B)

R(A)

A := A + 100

W(A)

**Release(A)**

T2

XLock(A)

XLock(C)

R(A)

A := A + 200

W(A)

R(C)

C := C + 200

W(C)

Release(A)

Release(C)

Commit

...

R(B)

B := B + 200

W(B)

Release(B)

**Commit**

## Two-Phase Locking

T1	T2	Values
XLock(A)		
XLock(B)		
R(A)		100
A := A + 100		
W(A)		200
Release(A)		
	XLock(A)	
	XLock(C)	
	R(A)	200
	A := A + 200	
	W(A)	400
	R(C)	
	C := C + 200	
	W(C)	
	Release(A)	
	Release(C)	
	Commit	
...		

- suppose T1 is forced to terminate at time t  
=> T1's updates are undone (value of A is restored to 100)  
=> T2's update to A is lost (incorrect, as atomicity is compromised, T2 also changed the value of C)
- problem – T1 released its exclusive lock on A prior to completion (under Strict 2PL, T1 can release its lock on A, as well as any other locks, only when it commits / aborts)

t

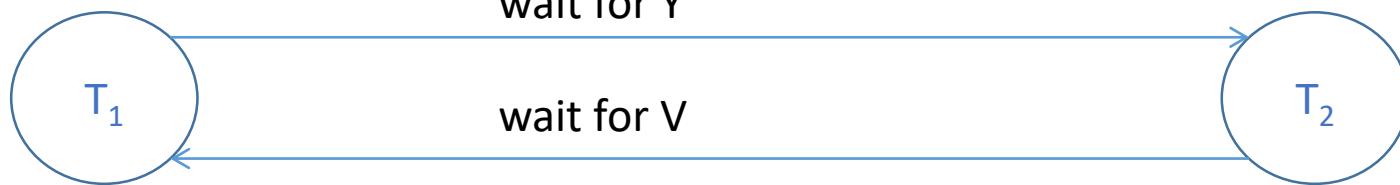
## Strict Schedules

- if transaction  $T_i$  has written object A, then transaction  $T_j$  can read and / or write A only after  $T_i$ 's completion (commit / abort)
- strict schedules:
  - avoid cascading aborts
  - are recoverable schedules
  - if a transaction is aborted, its operations can be undone
- Strict 2PL only allows strict schedules

## Deadlocks

- lock-based concurrency control techniques can lead to deadlocks
- deadlock
  - cycle of transactions waiting for one another to release a locked resource
  - normal execution can no longer continue without an external intervention, i.e., deadlocked transactions cannot proceed until the deadlock is resolved
- deadlock management
  - deadlock prevention
  - deadlock detection
    - allow deadlocks to occur and resolve them when they arise

# Deadlocks



T1

BEGIN TRAN

XLock(V)

Read(V)

V := V + 100

Write(V)

XLock(Y)

Wait

Wait

...

T2

BEGIN TRAN

XLock(Y)

Read(Y)

Y = Y \* 5

Write(Y)

XLock(V)

Wait

Wait

...

- T1 cannot obtain an X lock on Y, since T2 holds a conflicting lock on Y
- similarly, T2 cannot obtain an X lock on V, since T1 holds a conflicting lock on V

## Deadlocks - Prevention

- assign transactions timestamp-based priorities (each transaction has a timestamp - the moment it begins execution)
  - the lower the timestamp, the older the transaction
  - the older a transaction is, the higher its priority, with the oldest transaction having the highest priority
- 
- 2 deadlock prevention policies: Wait-die and Wound-wait

## Deadlocks - Prevention

- assume  $T_1$  wants to access an object locked by  $T_2$  (with a conflicting lock)
  - Wait-die
    - if  $T_1$ 's priority is higher,  $T_1$  can wait; otherwise,  $T_1$  is aborted
- in the following execution, 2 transactions are reading and / or writing 2 objects, x and y;  $T_1$ 's priority is higher:



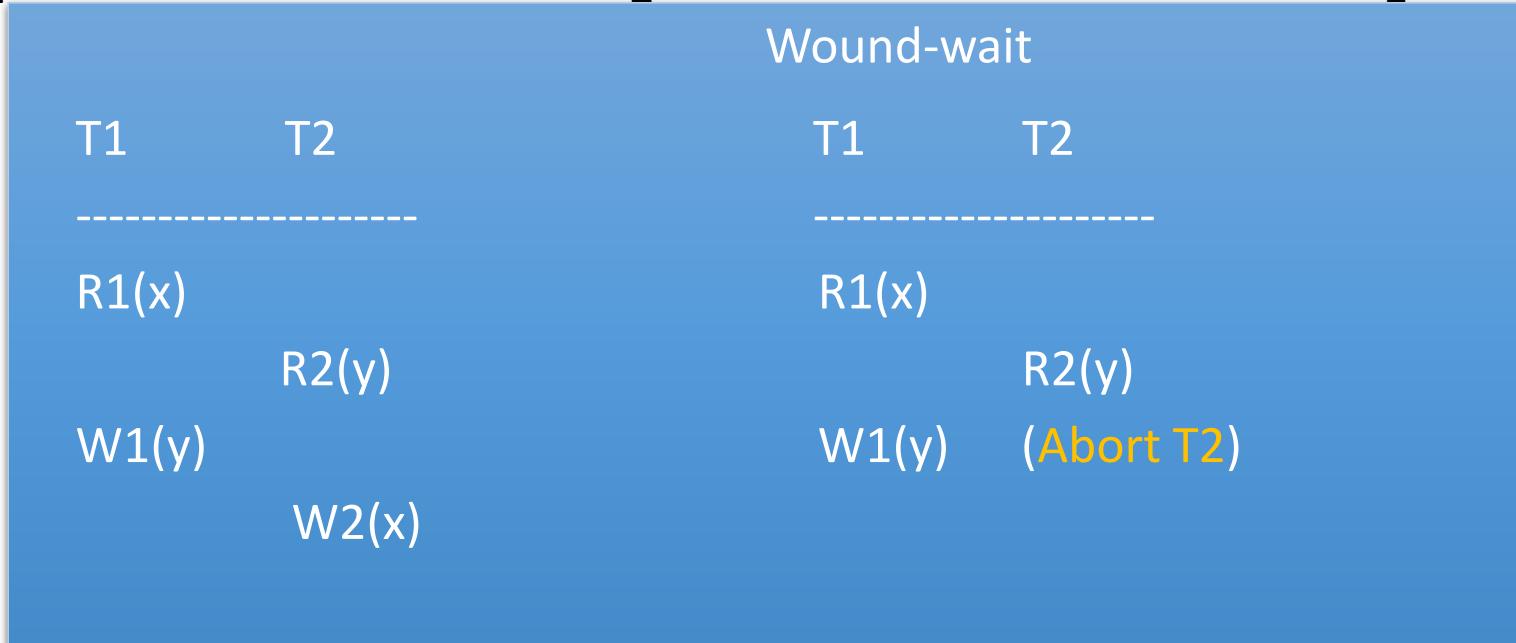
## Deadlocks - Prevention



- T1 requests an X lock on object y, which is already locked with a conflicting lock by T2
- since T1 has a higher priority, it is allowed to wait
- T2 asks for an X lock on object x, already locked with a conflicting lock by T1
- since T2 has a lower priority, it is aborted
- T1 now obtains the requested lock on object y and proceeds with the write operation

## Deadlocks - Prevention

- assume  $T_1$  wants to access an object locked by  $T_2$  (with a conflicting lock)
  - Wound-wait
    - if  $T_1$ 's priority is higher,  $T_2$  is aborted; otherwise,  $T_1$  can wait



- $T_1$  requests an X lock on object y, which is already locked with a conflicting lock by  $T_2$
- since  $T_1$  has a higher priority,  $T_2$  is aborted
- $T_1$  obtains the requested lock on object y and continues execution

## Deadlocks - Prevention

- under these policies (Wait-die / Wound-wait), deadlock cycles cannot develop
- if an aborted transaction is restarted, it's assigned its original timestamp

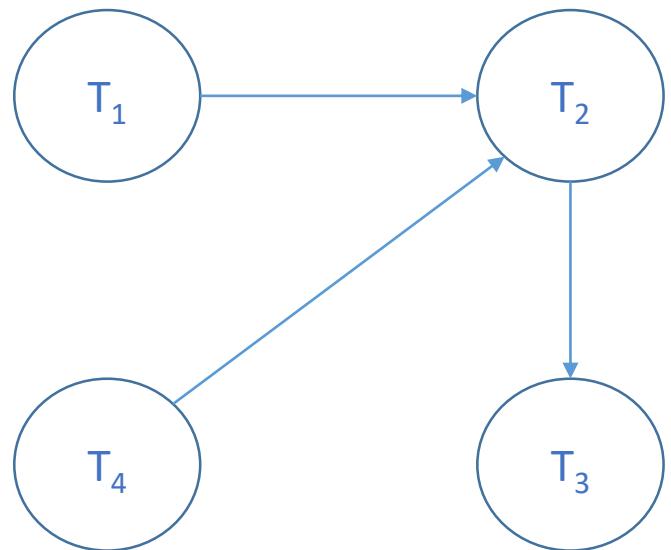
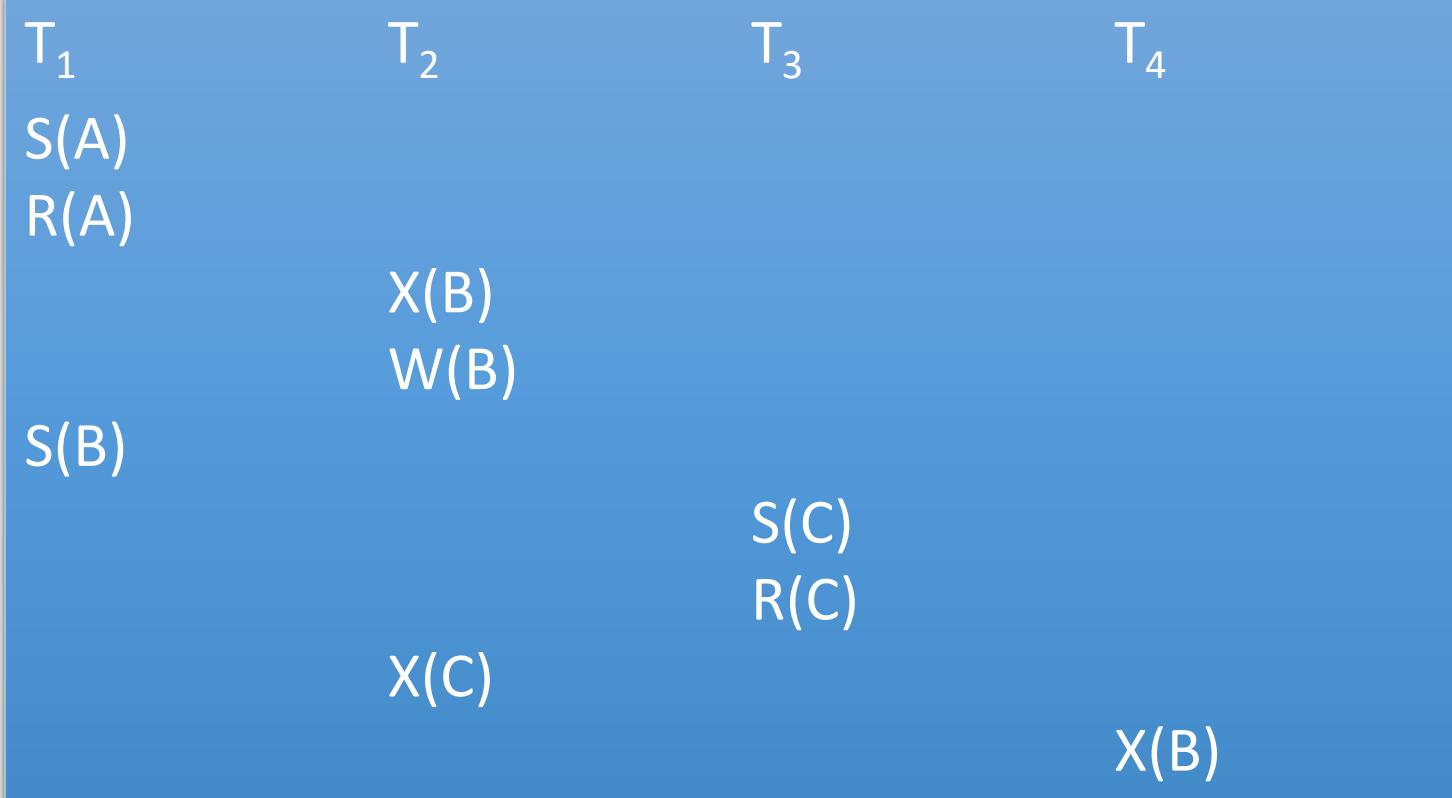
## Deadlocks - Detection

### a. waits-for graph

- structure maintained by the lock manager to detect deadlock cycles
  - a node / active transaction
  - arc from  $T_i$  to  $T_j$  if  $T_i$  is waiting for  $T_j$  to release a lock
- cycle in the graph  $\Rightarrow$  deadlock
- DBMS periodically checks whether there are cycles in the waits-for graph

# Deadlocks - Detection

## a. waits-for graph



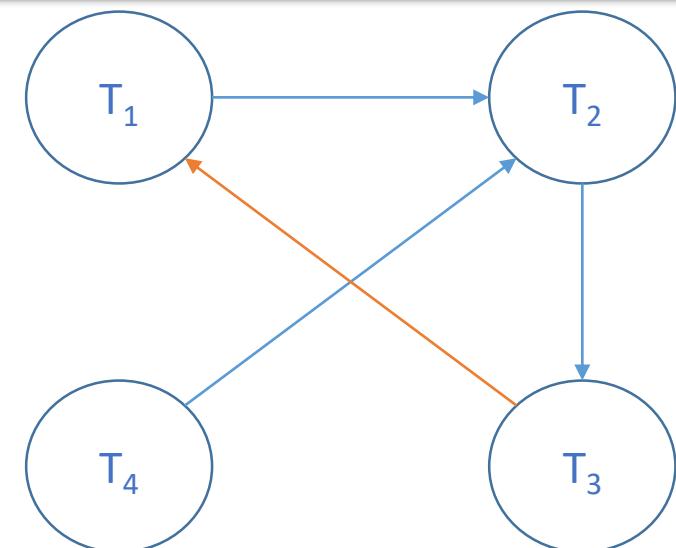
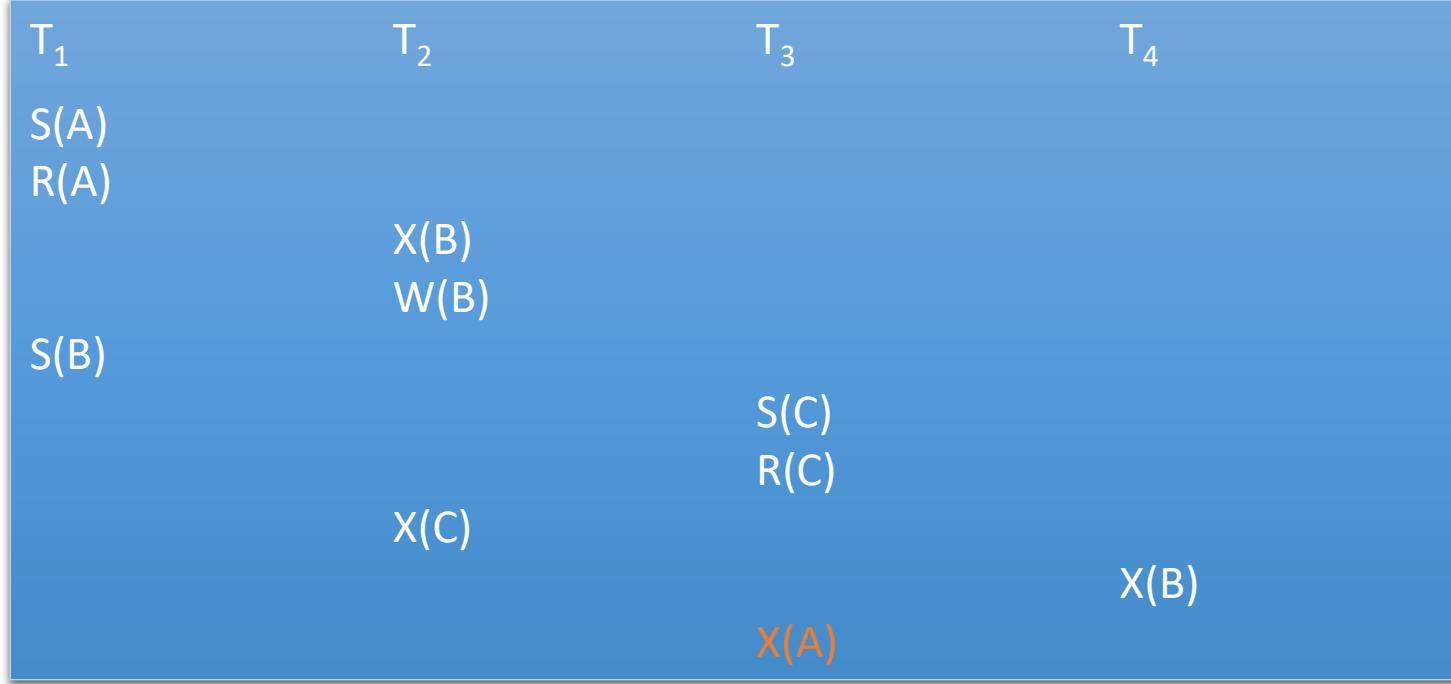
## Deadlocks - Detection

### a. waits-for graph

- if operation  $X(A)$  is also part of T3, there will be an arc from T3 to T1 in the graph (since T1 holds a conflicting lock on A, and T3 is waiting for T1 to release this lock)

- i.e., the graph contains a cycle (T1 is also waiting for T2 to release a lock, which in turn is waiting for T3 to release a lock)  
=> deadlock

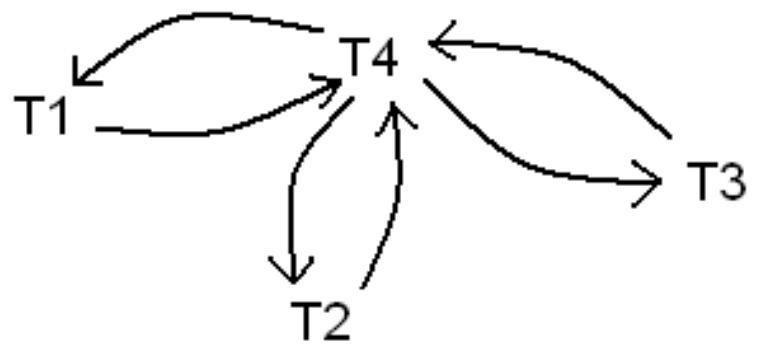
- aborting a transaction that appears on a cycle allows several other transactions to proceed



# Deadlocks - Detection

## a. waits-for graph

$T_1$	$T_2$	$T_3$	$T_4$
$S(A)$	$S(A)$	$S(A)$	
$R(A)$	$R(A)$	$R(A)$	
$X(B)$	$X(B)$	$X(B)$	
...	...	...	
			$S(B)$ $R(B)$ $X(A)$ ...



## Deadlocks - Detection

### b. timeout mechanism

- very simple, practical method of detecting deadlocks
- if a transaction T has been waiting too long for a lock on an object, a deadlock is assumed to exist and T is terminated

## Deadlocks – Choosing the Deadlock Victim

- possible criteria to consider when choosing the deadlock victim
  - the number of objects modified by the transaction
  - the number of objects that are to be modified by the transaction
  - the number of locks held
- the policy should be “fair”, i.e., if a transaction is repeatedly chosen as a victim, it should be eventually allowed to proceed

## The Phantom Problem

example 1. Researchers[RID, ..., ImpactFactor, Age]

- Page1: <R1, 5, 30>, <R2, 5, 20>
- Page2: <R4, 5, 100>
- Page3: <R8, 6, 18>, <R9, 6, 19>
- concurrent transactions T1 and T2
  - transaction T1
    - retrieve the age of the oldest researcher for each of the impact factor values 5 and 6
  - transaction T2
    - add new researcher with impact factor 5
    - remove researcher R9
- T1 and T2 obey Strict 2PL

## The Phantom Problem

- Page1: <R1, 5, 30>, <R2, 5, 20>
- Page2: <R4, 5, 100>
- Page3: <R8, 6, 18>, <R9, 6, 19>
- T1 identifies and locks pages holding researchers with IF 5 (Page1, Page2)
- T1 computes max age for IF 5 (100)
- T2 can acquire X locks on two different pages: Page4 (to which it adds a new researcher with IF 5 and age 102) and Page3 (from which it deletes researcher R9, with IF 6 and age 19)
- T2 then commits, releasing all its locks
- T1 now obtains an S lock on Page3 (containing all researchers with IF 6), and computes max age for IF 6 (18)

T1

SLock(Page1)

SLock(Page2)

compute max age for IF 5 => 100

SLock(Page3)

compute max age for IF = 6 => 18

...

T2

XLock(Page4)

XLock(Page3)

add record <R5, 5, 102> to Page4

delete researcher R9

commit – all locks are released

## The Phantom Problem

- Page1: <R1, 5, 30>, <R2, 5, 20>
- Page2: <R4, 5, 100>
- Page3: <R8, 6, 18>, <R9, 6, 19>
- outcome of interleaved schedule on the right:
  - IF 5, Max Age 100
  - IF 6, Max Age 18
- outcome of serial schedule (T1T2):
  - IF 5, Max Age 100
  - IF 6, Max Age 19
- outcome of serial schedule (T2T1):
  - IF 5, Max Age 102
  - IF 6, Max Age 18

T1

SLock(Page1)

SLock(Page2)

compute max age for IF 5 => 100

SLock(Page3)

compute max age for IF = 6 => 18

...

T2

XLock(Page4)

XLock(Page3)

add record <R5, 5, 102> to Page4

delete researcher R9

commit – all locks are released

->

## The Phantom Problem

=> the interleaved schedule is not serializable, as its outcome is not identical to the outcome of any serial schedule

- however, the schedule is conflict serializable (the precedence graph is acyclic)

=> in the presence of insert operations, i.e., if new objects can be added to the database, conflict serializability does not guarantee serializability

# The Phantom Problem

## example 2.

- T1 executes the same query twice
- between the 2 read operations, another transaction T2 inserts a row that meets the condition in T1's query; T2 commits

T1

```
SELECT *
FROM Students
WHERE GPA >= 8
```

T2

```
INSERT INTO Students VALUES
(12, 'Mara', 'Dobse', 10)
COMMIT
```

```
SELECT *
FROM Students
WHERE GPA >= 8
```

...

result set for T1's query

row corresponding to student with sid 12  
is not in the result set

row corresponding to student with sid 12  
now appears in the result set

## Transaction Support in SQL - Isolation Levels

- SQL provides support for users to specify various aspects related to transactions, e.g., isolation levels
- *isolation level*
  - a transaction's characteristic
  - determines the degree to which a transaction is isolated from the changes made by other concurrently running transactions
- greater concurrency -> concurrency anomalies

## Transaction Support in SQL - Isolation Levels

- 4 isolation levels
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ
  - SERIALIZABLE
- isolation levels can be set with the following command:
  - SET TRANSACTION ISOLATION LEVEL *isolevel*
  - e.g., SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

## Isolation Levels

- READ UNCOMMITTED

- a transaction must acquire an exclusive lock prior to writing an object
- no locks are requested when reading objects
- exclusive locks are released at the end of the transaction
- lowest degree of isolation

T1	T2	Values
R(A)		10
W(A)		20
	R(A)	20
	...	
	Commit	
...		
Abort		

- dirty reads can occur under this isolation level
- T1 acquires an X lock on A, reads A (value 10), writes A (value 20)
- T1's X lock on A will only be released when T1 commits / aborts
- T1 is still in progress when T2 attempts to read A; since no S lock is required when reading data under READ UNCOMMITTED, T2 is able to read value 20 for A

## Isolation Levels

- READ UNCOMMITTED

unrepeatable reads ✓

T1	T2
R(A)	
	R(A)
	W(A)
C	
R(A)	
W(A)	
C	

phantoms ✓

T1	T2
R(students with id between 100 and 110)	
	insert student with id 101
	C
R(students with id between 100 and 110)	
C	

- it is easy to see that this isolation level also exposes transactions to unrepeatable reads and phantoms

## Isolation Levels

- READ COMMITTED
  - a transaction must acquire an exclusive lock prior to writing an object
  - a transaction must acquire a shared lock prior to reading an object (i.e., the last transaction that modified the object is finished)
  - exclusive locks are released at the end of the transaction
  - shared locks are immediately released (as soon as the read operation is completed)
- dirty reads can no longer occur under this isolation level, but unrepeatable reads and phantoms are still possible

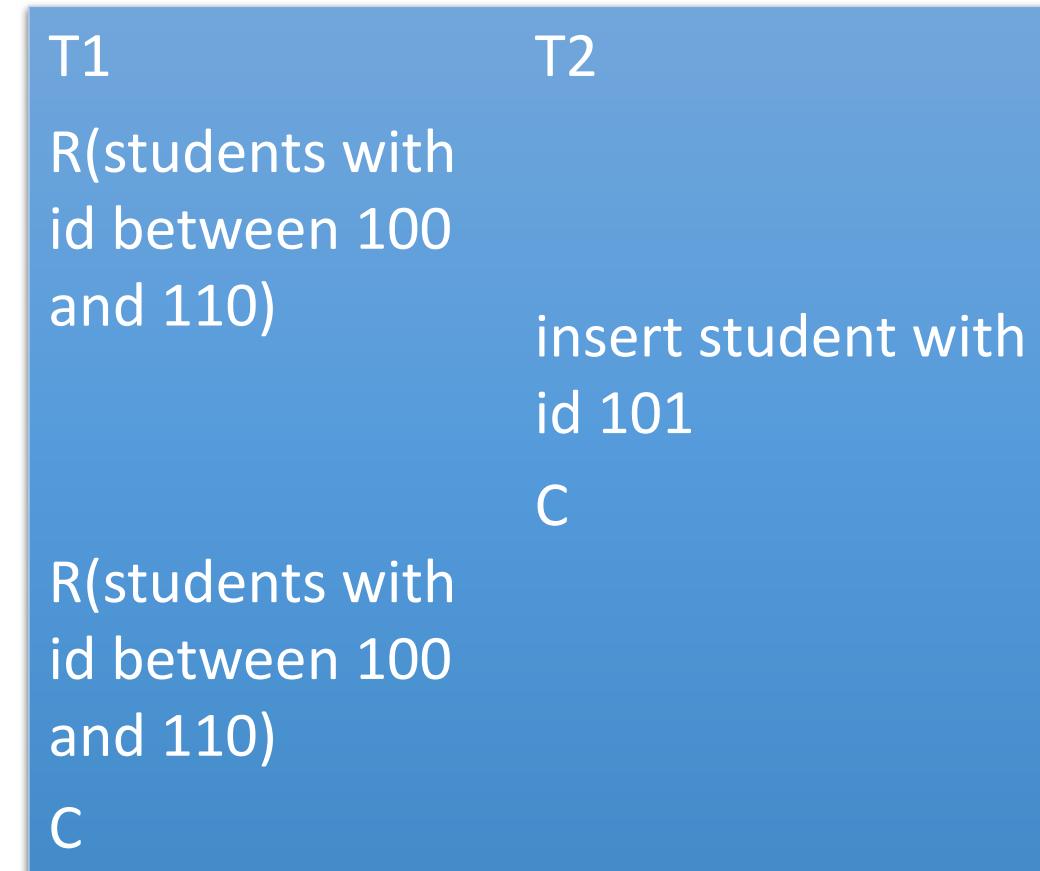
# Isolation Levels

- READ COMMITTED

- dirty reads      ✗

unrepeatable reads ✓

phantoms ✓



## Isolation Levels

- REPEATABLE READ
  - a transaction must acquire an exclusive lock prior to writing an object
  - a transaction must acquire a shared lock prior to reading an object
  - exclusive locks are released at the end of the transaction
  - shared locks are released at the end of the transaction
- dirty reads and unrepeatable reads cannot occur under REPEATABLE READ, but phantoms are still possible

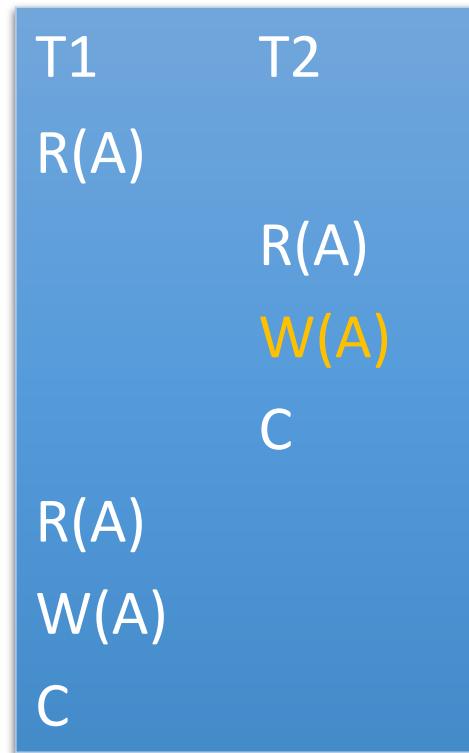
# Isolation Levels

- REPEATABLE READ

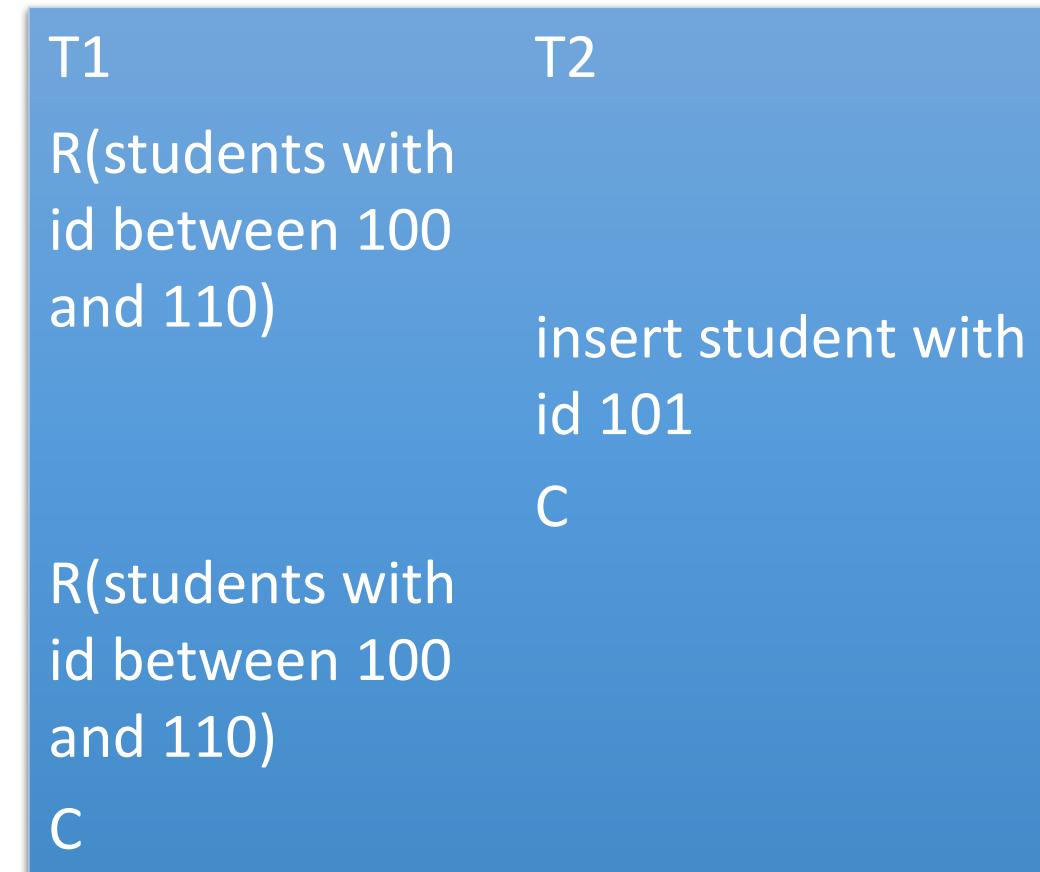
- dirty reads ✗



- unrepeatable reads ✗



- phantoms ✓



## Isolation Levels

- SERIALIZABLE
  - a transaction must acquire locks on objects before reading / writing them
  - a transaction can also acquire locks on sets of objects that must remain unmodified
    - if a transaction T reads a set of objects based on a search predicate, this set cannot be changed while T is in progress (if query *Return all students with GPA >= 8* is executed twice within a transaction, it must return the same answer set)
  - locks are held until the end of the transaction
  - highest degree of isolation
- dirty reads, unrepeatable reads, phantoms can't occur under this isolation level

# Isolation Levels

- SERIALIZABLE

- dirty reads

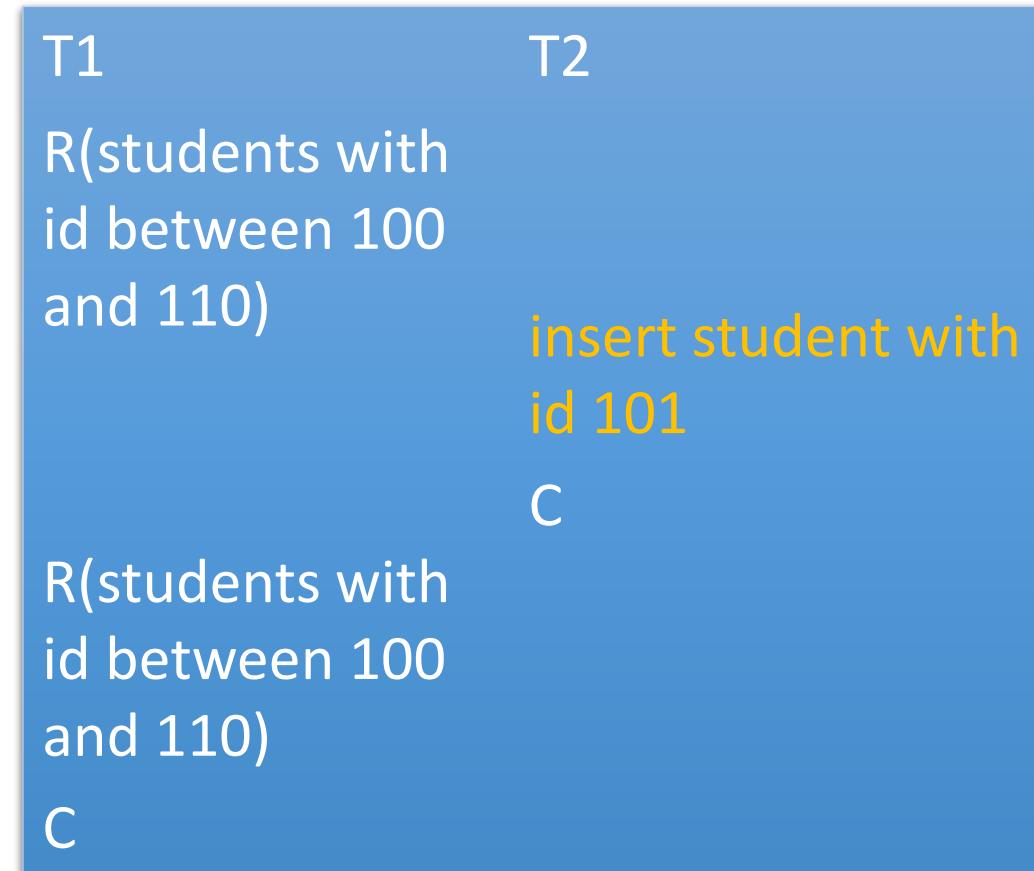
✗

- unrepeatable reads

✗

- phantoms

✗



\* Jim Gray \*

primary research interests – databases, transaction processing systems

## References

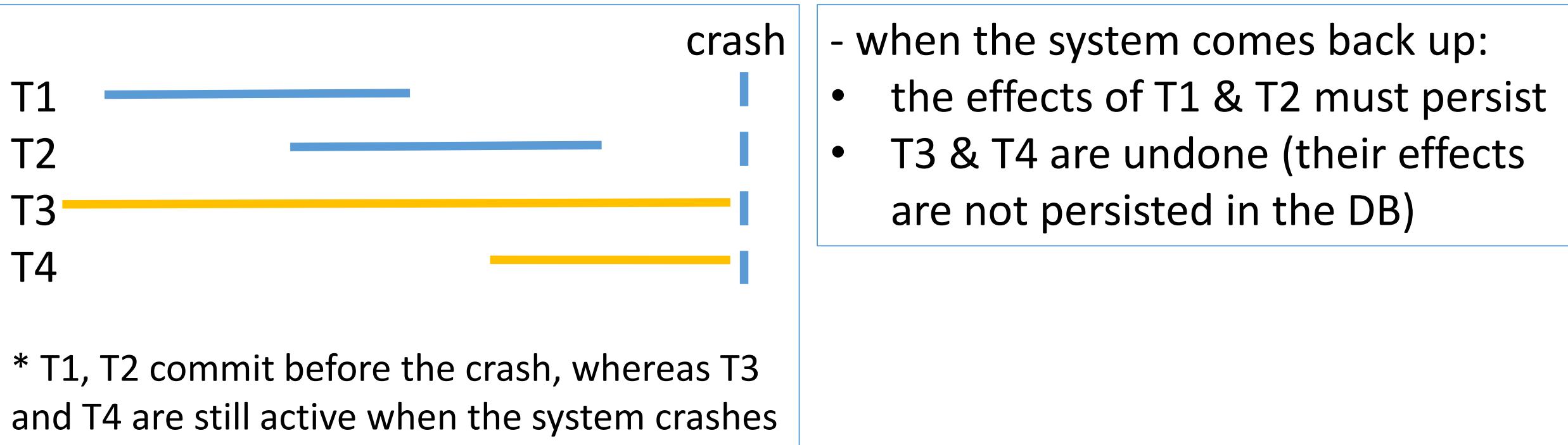
- [Ra00] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (2<sup>nd</sup> Edition), McGraw-Hill, 2000
- [Le99] LEVENE, M., LOIZOU, G., A Guided Tour of Relational Databases and Beyond, Springer, 1999
- [Ga08] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book, Prentice Hall Press, 2008
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007,  
<http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Si10] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, McGraw-Hill, 2010, <http://codex.cs.yale.edu/avi/db-book/>
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems,  
<http://infolab.stanford.edu/~ullman/fcdb.html>

# Database Management Systems

Lecture 4  
Crash Recovery

## Recovery - ACID

- the Recovery Manager (RM) in a DBMS ensures two important properties of transactions:
  - atomicity - the effects of uncommitted transactions (i.e., transactions that do not commit prior to the crash) are undone
  - durability - the effects of committed transactions survive system crashes



## Transaction Failure - Causes

- system failure (hardware failures, bugs in the operating system, database system, etc)
  - all running transactions terminate
  - contents of internal memory – affected (i.e., lost)
  - contents of external memory – not affected
- application error (“bug”, e.g., division by 0, infinite loop, etc)
  - => transaction fails; it should be executed again only after the error is corrected
- action by the Transaction Manager (TM)
  - e.g., deadlock resolution scheme
  - a transaction is chosen as the deadlock victim and terminated
  - the transaction might complete successfully if executed again

## Transaction Failure - Causes

- self-abort
  - based on some computations, a transaction can decide to terminate and undo its actions
  - there are special statements for this purpose, e.g., *ABORT*, *ROLLBACK*
  - it can be seen as a special case of *action by the TM*

## Normal Execution

- during normal execution, transactions read / write database objects
- reading database object O:
  - bring O from the disk into a frame in the Buffer Pool (BP)\*
  - copy O's value into a program variable
- writing database object O:
  - modify an in-memory copy of O (in the BP)
  - write the in-memory copy to disk

\* see the *Databases* course in the 1<sup>st</sup> semester (lecture 8 - Buffer Manager)

## Writing Objects

- options for the Buffer Manager (BM): *steal / no-steal, force / no-force*
- transaction T changes object O (in frame F in the BP)
- transaction T2 needs a page; the BM chooses F as a replacement frame (while T is still in progress)
  - *steal* approach:
    - T's changes can be written to disk while T is in progress (T2 steals a frame from T)
  - *no-steal* approach:
    - T's changes cannot be written to disk before T commits
- *force* approach
  - T's changes are immediately forced to disk when T commits
- *no-force* approach
  - T's changes are not forced to disk when T commits

## Writing Objects

- *no-steal approach*
  - advantage - changes of aborted transactions don't have to be undone (such changes are never written to disk!)
  - drawback - assumption: all pages modified by active transactions can fit in the BP
- *force approach*
  - advantage - actions of committed transactions don't have to be redone
    - by contrast, when using *no-force*, the following scenario is possible: transaction T commits at time  $t_0$ ; its changes are not immediately forced to disk; the system crashes at time  $t_1 \Rightarrow$  T's changes have to be redone!
  - drawback - can result in excessive I/O
- *steal, no-force approach* – used by most systems

## ARIES

- recovery algorithm; *steal, no-force* approach
- system restart after a crash - three phases:
  - analysis – determine:
    - active transactions at the time of the crash
    - *dirty pages*, i.e., pages in BP whose changes have not been written to disk
  - redo - reapply all changes (starting from a certain record in the log), i.e., bring the DB to the state it was in when the crash occurred
  - undo - undo changes of uncommitted transactions
- fundamental principle - *Write-Ahead Logging*
  - a change to an object O is first recorded in the log (e.g., in log record LR)
  - LR must be written to disk before the change to O is written to disk

## ARIES

### \* example

- analysis
  - active transactions at crash time: T1, T3 (to be undone)
  - committed transactions: T2 (its effects must persist)
  - potentially dirty pages: P1, P2, P3
- redo
  - reapply all changes in order (1, 2, ...)
- undo
  - undo changes of T1 and T3 in reverse order (6, 5, 1)

LSN	Log
1	T1 writes P1
2	T2 writes P2
3	T2 commit
4	T2 end
5	T3 writes P3
6	T3 writes P2
	crash & restart

## The Log (journal)

- history of actions executed by the DBMS
- stored in *stable storage*, i.e., keep  $\geq 2$  copies of the log on different disks (locations) - ensures the “durability” of the log
- records are added to the end of the log
- *log tail* - the most recent fragment of the log
  - kept in main memory and periodically forced to stable storage
- *Log Sequence Number (LSN)* - unique id for every log record
  - monotonically increasing (e.g., address of 1<sup>st</sup> byte of log record)
- every page P in the DB contains the *pageLSN*: the LSN of the most recent record in the log describing a change to P
- *log record* – fields:
  - prevLSN – linking a transaction’s log records
  - transID – id of the corresponding transaction
  - type – type of the log record

## The Log

- each of the following actions results in a log record being written: *update page, commit, abort, end, undoing an update*
- update page P
  - add an *update type* log record ULR to the log tail (with  $LSN_{ULR}$ )
  - $pageLSN(P) := LSN_{ULR}$
- transaction T commits
  - add a *commit type* log record CoLR to the log tail
  - force log tail to stable storage (including CoLR)
  - complete subsequent actions (remove T from transaction table)
- transaction T aborts
  - add an *abort type* log record to the log
  - initiate Undo for T
- transaction T ends
  - T commits / aborts - complete required actions

## The Log

- transaction T ends
  - add an *end type* log record to the log
- undoing an update
  - i.e., when the change described in an update log record is undone
    - add a *compensation log record* (CLR) to the log
- obs. committed transaction – a transaction whose log records (including the *commit log record*) have been written to stable storage
- *update log record* – additional fields: *pageID* (id of the changed page), *length* (length of the change (in bytes)), *offset* (offset of the change), *before-image* (value before the change), *after-image* (value after the change)
  - can be used to undo / redo the change

# The Log

- *compensation log record*
  - let U be an update log record describing an update of transaction T
  - let C be the compensation log record for U, i.e., C describes the action taken to undo the changes described by U
  - C has a field named *undoNextLSN*:
    - the LSN of the next log record to be undone for T
    - set to the value of *prevLSN* in U

\* example: undo T10's update  
to P10

=> CLR with *transID* = T10,  
*pageID* = P10, *length* = 2,  
*offset* = 10,  
*before-image* = JH and

*undoNextLSN* = LSN of 1<sup>st</sup> log record (i.e., the next  
record that is to be undone for transaction T10)

prevLSN	transID	type	pageID	length	offset	before-image	after-image
	T10	update	P100	2	10	AB	CD
	T15	update	P2	2	10	YW	ZA
	T15	update	P100	2	9	EC	YW
	T10	update	P10	2	10	JH	AB

log

# The Transaction Table and the Dirty Page Table

- contain important information for the recovery process
- *transaction table*:
  - 1 entry / active transaction
  - fields: *transID*, *lastLSN* (LSN of the most recent log record for the transaction), *status* (in progress / committed / aborted)
  - example (*status* not displayed):

transID	lastLSN	prevLSN	transID	type	pageID	length	offset	before-image	after-image
T10			T10	update	P100	2	10	AB	CD
T15			T15	update	P2	2	10	YW	ZA
			T15	update	P100	2	9	EC	YW
			T10	update	P10	2	10	JH	AB

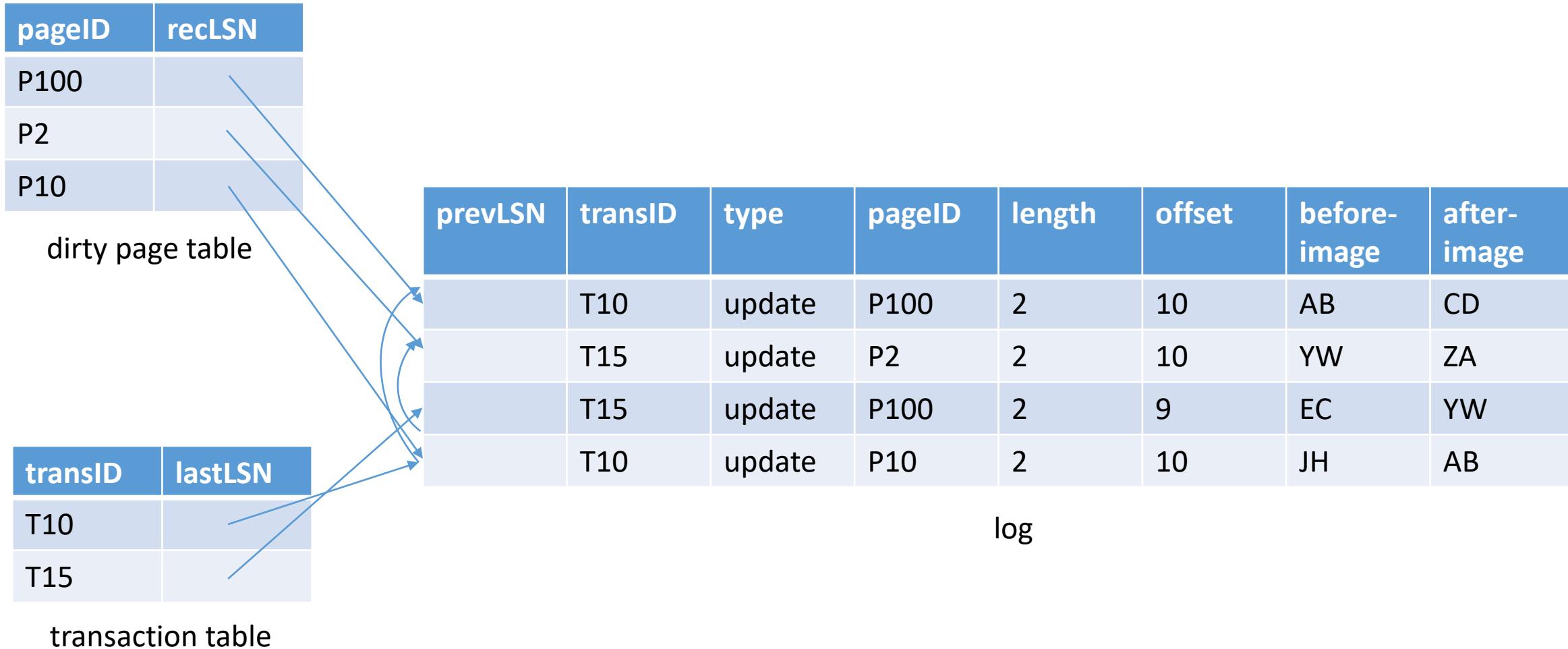
transaction table

log

The diagram illustrates the connection between the Transaction Table and the Dirty Page Table. It shows two tables side-by-side. The Transaction Table has columns: transID, lastLSN, prevLSN, transID, type, pageID, length, offset, before-image, and after-image. The Dirty Page Table has columns: transID, lastLSN, prevLSN, transID, type, pageID, length, offset, before-image, and after-image. Arrows indicate that the 'lastLSN' value in the Transaction Table's 'lastLSN' column points to the 'lastLSN' row in the Dirty Page Table, and the 'lastLSN' value in the Dirty Page Table's 'lastLSN' column points back to the Transaction Table's 'lastLSN' column.

# The Transaction Table and the Dirty Page Table

- *dirty page table*:
  - 1 entry / dirty page in the Buffer Pool
  - fields: *pageID*, *recLSN* (the LSN of the 1<sup>st</sup> log record that dirtied the page)



## Checkpointing

- objective: reduce the amount of work performed by the system when it comes back up after a crash
- *checkpoints* taken periodically; 3 steps:
  - write a *begin\_checkpoint* record (it indicates when the checkpoint starts; let its LSN be  $LSN_{BCK}$ )
  - write an *end\_checkpoint* record
    - it includes the current Transaction Table and the current Dirty Page Table
  - after the *end\_checkpoint* record is written to stable storage:
    - write a *master* record to a known place on stable storage
    - it includes  $LSN_{BCK}$
- crash -> restart -> system looks for the most recent checkpoint
- normal execution begins with a checkpoint with an empty Transaction Table and an empty Dirty Page Table

## Recovery - overview

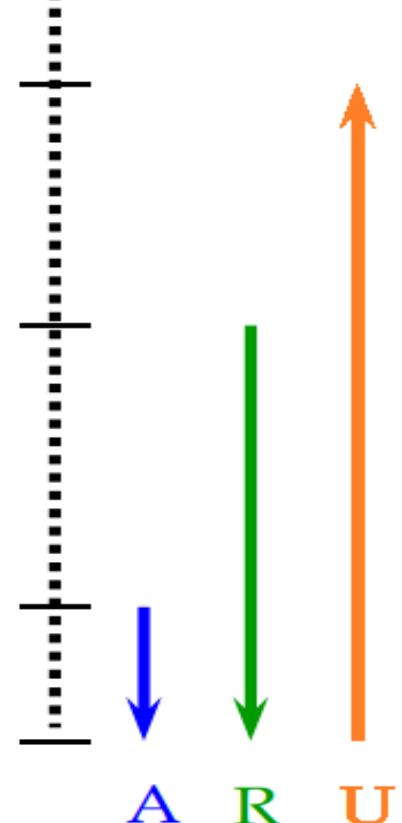
- system restart after a crash – 3 phases:
  - **Analysis**
    - reconstructs state at the most recent checkpoint
    - scans the log forward from the most recent checkpoint
    - identifies:
      - active transactions at the time of the crash (to be undone)
      - potentially dirty pages at the time of the crash
      - the starting point for the Redo pass
  - **Redo**
    - repeats history, i.e., reapplies changes to dirty pages

the oldest log record of transactions that were active at the time of the crash

the smallest recLSN in the dirty page table (determined during the Analysis pass)

the most recent checkpoint (master record)

system crash



## Recovery - overview

- system restart after a crash – 3 phases

- **Redo**

- all updates are reapplied (regardless of whether the corresponding transaction committed or not)
  - starting point is determined in the Analysis pass
  - scans the log forward until the last record

- **Undo**

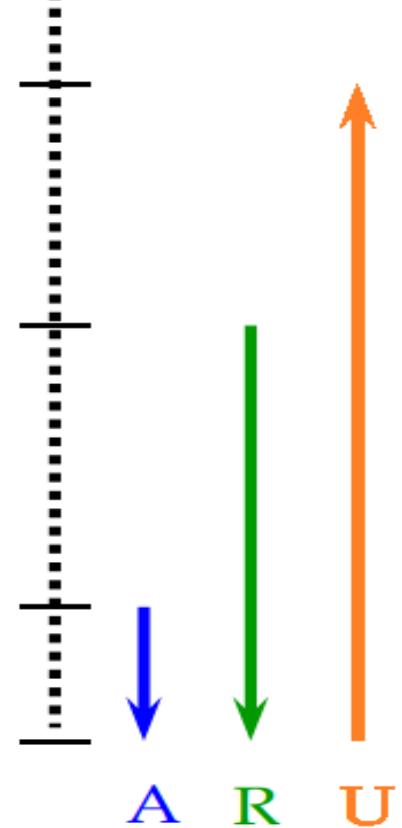
- the effects of transactions that were active at the time of the crash are undone
  - such changes are undone in the opposite order (i.e., Undo scans the log backward from the last record)

the oldest log record of transactions that were active at the time of the crash

the smallest recLSN in the dirty page table (determined during the Analysis pass)

the most recent checkpoint (master record)

system crash



## \* Analysis

- investigate the most recent *begin\_checkpoint* log record
  - get the next *end\_checkpoint* log record EC
- set Dirty Page Table to the copy of the Dirty Page Table in EC
- set Transaction Table to the copy of the Transaction Table in EC

->

\* Analysis - scan the log forward from the most recent checkpoint:

- transactions:
  - encounter **end log record** for transaction T:
    - remove T from transaction table
  - encounter **other log records (LR)** for transaction T:
    - add T to Transaction Table if not already there
    - set T.lastLSN to LR.LSN
    - if LR is a commit type log record:
      - set T's status to *C*
    - otherwise, set status to *U* (i.e., to be undone)
- pages:
  - encounter **redoable log record (LR)** for page P:
    - if P is not in the Dirty Page Table:
      - add P to Dirty Page Table
      - set P.recLSN to LR.LSN

## Example 1

- first 5 log records are written to stable storage
- system crashes before the 6<sup>th</sup> log record is written to stable storage

prevLSN	transID	type	pageID	length	offset	before-image	after-image
	T10	update	P100	2	10	AB	CD
	T15	update	P2	2	10	YW	ZA
	T15	update	P100	2	9	EC	YW
	T10	update	P10	2	10	JH	AB
	T15	commit					
	T10	update	P11	3	20	GFX	YTR

## Analysis

- most recent checkpoint – beginning of execution (empty Transaction Table, empty Dirty Page Table)
- 1<sup>st</sup> log record
  - add T10 to the Transaction Table
  - add P100 to the Dirty Page Table (recLSN = LSN(1<sup>st</sup> log record))

## Analysis

- 2<sup>nd</sup> log record
  - add T15 to the Transaction Table
  - add P2 to the Dirty Page Table (recLSN = LSN(2<sup>nd</sup> log record))



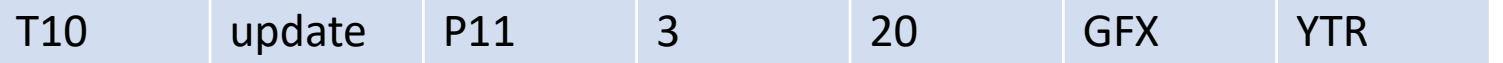
prevLSN	transID	type	pageID	length	offset	before-image	after-image
	T10	update	P100	2	10	AB	CD
	T15	update	P2	2	10	YW	ZA
	T15	update	P100	2	9	EC	YW
	T10	update	P10	2	10	JH	AB
	T15	commit					

- 4<sup>th</sup> log record
  - add P10 to the Dirty Page Table (recLSN = LSN(4<sup>th</sup> log record))
- active transactions at the time of the crash:
  - transactions with status *U*, i.e., T10 (T15 is a committed transaction)
- Dirty Page Table:
  - can include pages that were written to disk prior to the crash
  - assume P2's update is the only change written to disk before the crash, i.e., P2 is not dirty, but it's in the Dirty Page Table
  - the pageLSN on page P2 is equal to the LSN of the 2<sup>nd</sup> log record

## Analysis

prevLSN	transID	type	pageID	length	offset	before-image	after-image
	T10	update	P100	2	10	AB	CD
	T15	update	P2	2	10	YW	ZA
	T15	update	P100	2	9	EC	YW
	T10	update	P10	2	10	JH	AB
	T15	commit					

log

- log record  is not seen during Analysis (it was not written to disk before the crash)
- Write-Ahead Logging protocol => the corresponding change to page P11 cannot have been written to disk

## \* Redo

- *repeat history*: reconstruct state at the time of the crash
  - reapply *all* updates (even those of aborted transactions!), reapply CLRs
- scan the log forward from the log record with the smallest recLSN in the Dirty Page Table
- for each **redoable** log record **LR** affecting page P, redo the described action unless:
  - page P is not in the Dirty Page Table
  - page P is in the Dirty Page Table, but P.recLSN > LR.LSN
  - P.pageLSN (in DB)  $\geq$  LR.LSN
- to redo an action:
  - reapply the logged action
  - set P.pageLSN to LR.LSN
  - no additional logging!

## \* Redo

- at the end of Redo:
  - for every transaction  $T$  with status  $C$ :
    - add an end log record
    - remove  $T$  from the Transaction Table

## Redo

- previously stated assumption: P2's update is the only change written to disk before the crash, i.e., P2 is not dirty, but it's in the Dirty Page Table

log	prevLSN	transID	type	pageID	length	offset	before-image	after-image
		T10	update	P100	2	10	AB	CD
		T15	update	P2	2	10	YW	ZA
		T15	update	P100	2	9	EC	YW
		T10	update	P10	2	10	JH	AB
		T15	commit					

- Dirty Page Table -> smallest recLSN is the LSN of the 1<sup>st</sup> log record
- 1<sup>st</sup> log record
  - fetch page P100 (its pageLSN is less than the LSN of the current log record) => reapply update, set P100.pageLSN to the LSN of the 1<sup>st</sup> log record
- 2<sup>nd</sup> log record
  - fetch page P2
  - P2.pageLSN = LSN of the current log record => update is not reapplied
- 3<sup>rd</sup>, 4<sup>th</sup> log records – processed similarly

## \* Undo

- *loser transaction* – transaction that was active at the time of the crash
- $\text{ToUndo} = \{ l \mid l - \text{lastLSN of a } \text{loser} \text{ transaction}\}$
- repeat:
  - choose the largest LSN in ToUndo and process the corresponding log record LR; let T be the corresponding transaction
  - if LR is a CLR:
    - if  $\text{undoNextLSN} == \text{NULL}$ 
      - write an end log record for T
    - else { $\text{undoNextLSN} != \text{NULL}$ }
      - add  $\text{undoNextLSN}$  to ToUndo
  - else {LR is an update log record}
    - undo the update
    - write a CLR
    - add  $\text{LR.prevLSN}$  to ToUndo
- until ToUndo is empty

## Undo

- active transaction at the time of the crash: T10
- lastLSN of T10: LSN of the 4<sup>th</sup> log record
- 4<sup>th</sup> log record
  - undo update, write CLR
  - add LSN of 1<sup>st</sup> log record to ToUndo
- 1<sup>st</sup> log record
  - undo update
  - write CLR
  - write end log record for T10
- obs. if Strict 2PL is used, T15 cannot write P100 while T10 is active (T10 has also modified P100)

prevLSN	transID	type	pageID	length	offset	before-image	after-image
	T10	update	P100	2	10	AB	CD
	T15	update	P2	2	10	YW	ZA
	T15	update	P100	2	9	EC	YW
	T10	update	P10	2	10	JH	AB
	T15	commit					

log



## Example 2 – system crashes during Undo

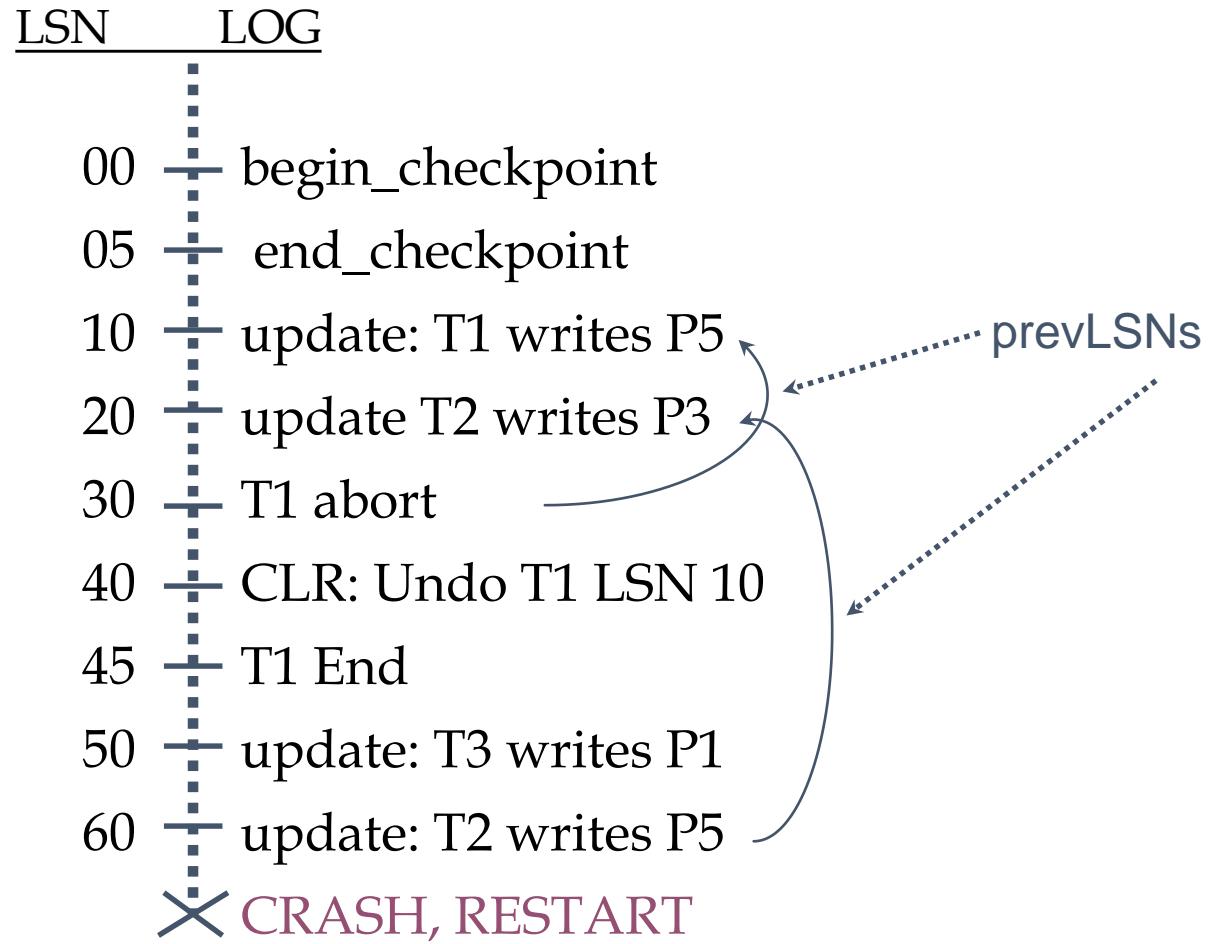
- consider the execution history below:

LSN	LOG
00	begin_checkpoint
05	end_checkpoint
10	update: T1 writes P5
20	update T2 writes P3
30	T1 abort
40	CLR: Undo T1 LSN 10
45	T1 End
50	update: T3 writes P1
60	update: T2 writes P5
	CRASH, RESTART

The diagram illustrates a log sequence number (LSN) timeline with various log entries. A solid blue arrow points from the 'T1 abort' entry at LSN 30 to the 'CLR: Undo T1 LSN 10' entry at LSN 40. A dotted blue arrow labeled 'prevLSNs' points from the 'T1 abort' entry to the 'CRASH, RESTART' entry at the bottom.

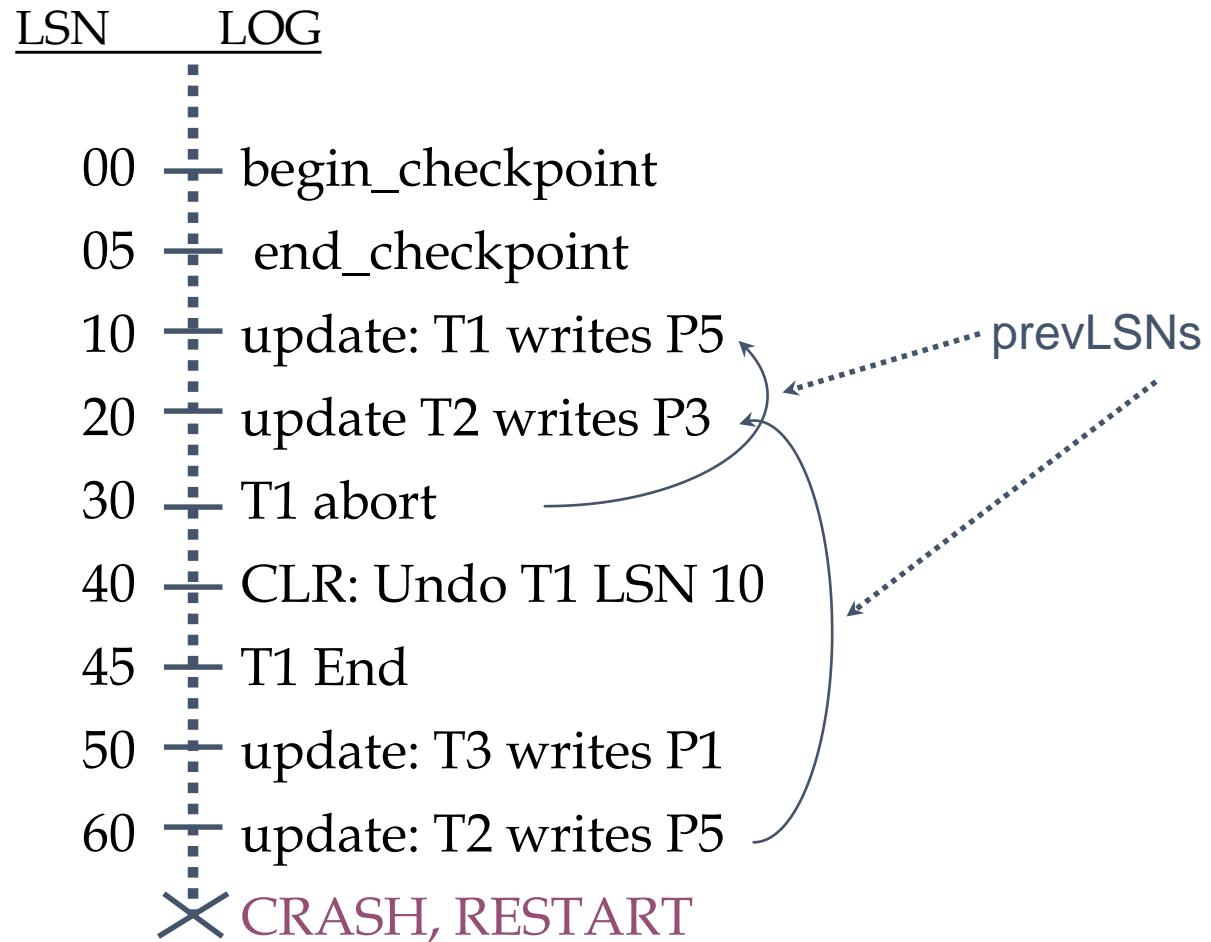
## Example 2

- T1 aborts
  - => its only update is undone (CLR with LSN 40)
  - T1 - terminated
- 1<sup>st</sup> crash:
  - Analysis:
    - dirty pages: P5 (recLSN 10), P3 (recLSN 20), P1 (recLSN 50)
    - active transactions at the time of the crash: T2 (lastLSN 60), T3 (lastLSN 50)



## Example 2

- 1<sup>st</sup> crash:
  - Redo:
    - starting point
      - log record with LSN = 10 (smallest recLSN in the Dirty Page Table)
    - reapply required actions in update log records / compensation log records



## Example 2

- 1<sup>st</sup> crash:
  - Undo:
    - T2, T3 – loser transactions  
=> ToUndo = {60, 50}
    - process log record with LSN 60:
      - undo update
      - write CLR (LSN 70) with undoNextLSN 20 (i.e., the next log record that should be processed for T2)

LSN	LOG
00,05	+ begin_checkpoint, end_checkpoint
10	+ update: T1 writes P5
20	+ update T2 writes P3
30	- T1 abort
40,45	- CLR: Undo T1 LSN 10, T1 End
50	+ update: T3 writes P1
60	+ update: T2 writes P5
70	X CRASH, RESTART
70	- CLR: Undo T2 LSN 60
80,85	- CLR: Undo T3 LSN 50, T3 end
	X CRASH, RESTART

A red circle highlights the log records at LSN 70 and 80, both marked with a large red X and labeled "CRASH, RESTART". A red arrow points from the text "undonextLSN" to the LSN 20 log entry.

## Example 2

- 1<sup>st</sup> crash:
  - Undo:
    - process log record with LSN 50:
    - undo update
    - write CLR (LSN 80) with *undoNextLSN null* (i.e., T3 completely undone, write end log record for T3)
  - log records with LSN 70, 80, 85 are written to stable storage
- 2<sup>nd</sup> crash (during undo)!

LSN	LOG
00,05	+ begin_checkpoint, end_checkpoint
10	+ update: T1 writes P5
20	+ update T2 writes P3
30	- T1 abort
40,45	- CLR: Undo T1 LSN 10, T1 End
50	+ update: T3 writes P1
60	+ update: T2 writes P5
X	CRASH, RESTART
70	- CLR: Undo T2 LSN 60
80,85	- CLR: Undo T3 LSN 50, T3 end
X	CRASH, RESTART

## Example 2

- 2<sup>nd</sup> crash:
  - Analysis:
    - the only active transaction: T2
    - dirty pages: P5 (recLSN 10), P3 (recLSN 20), P1 (recLSN 50)
  - Redo:
    - process log records with LSN between 10 and 85

LSN	LOG
00,05	+ begin_checkpoint, end_checkpoint
10	- update: T1 writes P5
20	- update T2 writes P3
30	- T1 abort
40,45	- CLR: Undo T1 LSN 10, T1 End
50	- update: T3 writes P1
60	- update: T2 writes P5
60	X CRASH, RESTART
70	- CLR: Undo T2 LSN 60
80,85	- CLR: Undo T3 LSN 50, T3 end
80,85	X CRASH, RESTART
90	- CLR: Undo T2 LSN 20, T2 end

## Example 2

- 2<sup>nd</sup> crash:
  - Undo:
    - lastLSN of T2: 70
    - ToUndo = {70}
    - process log record with LSN 70:
      - add 20 (undoNextLSN) to ToUndo
    - process log record with LSN 20:
      - undo update
      - write CLR (LSN 90) with undoNextLSN null => write end log record for T2

LSN	LOG
00,05	+ begin_checkpoint, end_checkpoint
10	- update: T1 writes P5
20	- update T2 writes P3
30	- T1 abort
40,45	- CLR: Undo T1 LSN 10, T1 End
50	- update: T3 writes P1
60	- update: T2 writes P5
70	X CRASH, RESTART
70	- CLR: Undo T2 LSN 60
80,85	- CLR: Undo T3 LSN 50, T3 end
80,85	X CRASH, RESTART
90	- CLR: Undo T2 LSN 20, T2 end

## Example 2

- 2<sup>nd</sup> crash:
    - Undo:
      - ToUndo empty
- => recovery complete!

LSN	LOG
00,05	+ begin_checkpoint, end_checkpoint
10	- update: T1 writes P5
20	- update T2 writes P3
30	- T1 abort
40,45	- CLR: Undo T1 LSN 10, T1 End
50	- update: T3 writes P1
60	- update: T2 writes P5
X	CRASH, RESTART
70	- CLR: Undo T2 LSN 60
80,85	- CLR: Undo T3 LSN 50, T3 end
X	CRASH, RESTART
90	- CLR: Undo T2 LSN 20, T2 end

undonextLSN

- obs. aborting a transaction
  - special case of Undo in which the actions of a single transaction are undone
- obs. system crash during the Analysis pass
  - all the work is lost
  - when the system comes back up, the Analysis phase has the same information as before
- obs. system crash during the Redo pass
  - some of the changes from the Redo pass may have been written to disk prior to the crash
  - the pageLSN will indicate such a situation, so these changes will not be reapplied in the subsequent Redo pass

## References

- [Ra00] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (2<sup>nd</sup> Edition), McGraw-Hill, 2000
- [Le99] LEVENE, M., LOIZOU, G., A Guided Tour of Relational Databases and Beyond, Springer, 1999
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007,  
<http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Si10] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, McGraw-Hill, 2010, <http://codex.cs.yale.edu/avi/db-book/>

# Database Management Systems

Lecture 5

Security

## \* database protection

- security & integrity
- security
  - protecting the data against unauthorized users (who may want to read, modify, destroy the data)
  - i.e., users have the right to do what they are trying to do
- integrity
  - protecting the data against authorized users
  - i.e., the operations that users are trying to execute are correct
  - the consistency of the database must be maintained at all times

\* aspects related to information security:

- legal, ethical aspects
  - certain rules are being broken
  - e.g., a person searches for a password or accidentally finds such a password and uses it
- physical controls
  - e.g., the computer room is locked (or guarded)
- software controls
  - files are protected (unauthorized access is not allowed)
  - there are no programs intercepting actions of authorized users
- operational problems
  - if a password mechanism is used, how are the passwords kept secret and how often should a user change his / her password?

\* data security and data integrity – similarities:

- the system enforces certain constraints (users cannot violate these constraints)
- security constraints, integrity constraints
- constraints are:
  - specified in a declarative language
  - saved in the system catalog

example: *primary key constraint*, specified in SQL and saved in the system catalog (retrieving info about PK constraints in SQL Server:

```
SELECT *
  FROM sys.objects
 WHERE type = 'PK')
```

- the system monitors users' actions to enforce the specified constraints
- example: INSERT, UPDATE statements cannot violate a PK constraint

- \* the DBMS's authorization subsystem (security subsystem)
  - checks any given access request against the applicable constraints
  - access request:
    - requested object + requested operation + requesting user
- example: *Alice wants to delete 10 rows from table Customers.*
- identifying the applicable constraints for an access request:
  - the system must recognize the source of the request, i.e., the requesting user
  - an authentication mechanism is used for this purpose, e.g.:
    - password scheme
      - users supply their user ID (users say who they are) and their password (users prove they are who they say they are)
      - fingerprint readers, voice verifiers, retinal scanners, etc

- \* main approaches to data security in a DBMS:

- discretionary control & mandatory control

- discretionary control

- users have different access rights (privileges) on different DB components

- every component in the database (table, view, procedure, trigger, etc) can have certain privileges that can be given to users

- a privilege allows the execution of a SQL statement

- e.g.,

- SELECT [ (column\_list) ]

- UPDATE [ (column\_list) ]

- INSERT

- DELETE

- ...

- the names of privileges vary among DBMSs

\* main approaches to data security in a DBMS:

- discretionary control

- the operations users are allowed to perform are explicitly specified
- anything not explicitly authorized is implicitly forbidden

examples:

*Alice and Bob are both allowed to SELECT from table Customers.*

*Jane is allowed to SELECT, INSERT, DELETE from / into table Customers.*

- SQL statements:

```
GRANT privilege_list ON component TO user_list [options]
```

```
REVOKE privilege_list ON component FROM user_list [options]
```

\* main approaches to data security in a DBMS:

- mandatory control

- every component has a classification level (e.g., *top secret*, *secret*, etc)
  - levels – strict ordering (e.g., *top secret* > *secret*)
- every user has a clearance level (same options as for classification levels)
- Bell and La Padula rules:

- user *x* can retrieve component *y* only if the clearance level of *x* is  $\geq$  the classification level of *y*

example: *Alice's clearance level is top secret, Bob's is secret.*

*Component C1's classification level is top secret, C2's is secret.*

*Alice can retrieve both C1 and C2, Bob can only retrieve C2.*

- user *x* can update component *y* only if the clearance level of *x* is equal to the classification level of *y*

example: *Alice can only update C1, Bob can only update C2.*

Q: why do you think Alice is not allowed to update C2?

## \* database user management

- user:
  - username
  - password
  - additional information: time intervals during which access is allowed; privileges; answers to some questions
- SQL statements:
  - CREATE / ALTER / DROP user

- \* recommendations
- using views
  - external structures that hide the conceptual structure\*
  - the user is allowed to see only a certain subset of the data
- database administrator should monitor the database and keep track of all operations performed by users via an audit trail
  - if there is any suspicion of wrongdoing, the audit trail can be analyzed
  - an audit trail can contain:
    - the executed SQL statement, with the corresponding: user name, IP address, date and time, affected object, old values and new values for changed records

\* see the *Databases* course in the 1<sup>st</sup> semester (lecture 1)

## \* SQL injection

- application -> execution of a SQL statement (fixed statement, statement that is generated using input data from the user)
- a statement can be changed (when being generated) due to data supplied by the user
- such a change is attempted when the user is trying to obtain additional access rights
- the user enters code into input variables; the code is concatenated with SQL statements and executed
- the DBMS executes all statements that are syntactically valid
- obs.
  - string separators: single quotes, double quotes
  - statement separator (if a command can include multiple statements): semicolon
  - comments in SQL: statement --comment

## \* SQL injection

### 1. changing a user's authentication statement

- errors when executing the statement => use the error messages in subsequent executions
- authenticating with a username and a password - in many cases, the following statement is executed:

```
SELECT ... WHERE user = "uname" AND password = "upassword"
```

columns in the *users*  
table (can have various  
other names)

could be an email

supplied by the client

- if the statement is successfully executed and returns at least one record, the authentication is successful

## \* SQL injection

### 1. changing a user's authentication statement

- values that change the SQL statement (modify the action intended by the programmer):

uname	upassword	condition in the statement
a	" OR 1 = 1 --	user="a" AND password="" OR 1=1 -- "
admin" --	?	user="admin" --" AND password ... (other values besides <i>admin</i> can be chosen from various lists on the internet)
" OR 0=0 --	?	user="" OR 0=0 -- ...

SELECT ... WHERE user = "uname" AND password = "upassword"

- for the values in the 1<sup>st</sup> row, the statement becomes:

SELECT ... WHERE user = "a" AND password = "" OR 1 = 1 --"

SELECT ... WHERE user = "a" AND password = "" OR 1 = 1 --" => all rows are retrieved from the DB, successful authentication

## \* SQL injection

provided by the client

### 2. obtaining information from the database

- the client is asked to provide value *v*:
- possible values entered by the user:

```
SELECT ... WHERE ... = "v" ...
```

- *x*" OR 1 = (SELECT COUNT(\*) FROM table) --
  - if an error occurs, the statement can be executed again, with a different table name; a malevolent user can obtain, in this way, the name of a table in the DB (for instance, a table with passwords)
- *x*" AND user IS NULL --
  - if an error occurs, the statement can be executed again, with a different column name (instead of *user* in the example); a malevolent user can obtain the name of a column
- *x*" AND user LIKE "%pop%" --
  - if an error occurs, or no data is returned => change the column name or the string in the condition (%pop%); one can obtain the name of a user

## \* SQL injection

### 3. changing data in tables

- the client is asked to provide value *v*:

- possible values entered by the user:

- 0; INSERT INTO users ...

- the SELECT statement is executed, then data about a new user is inserted into the *users* table

- 0 UNION SELECT CONCAT(name, password) FROM users

- retrieving all usernames and passwords while executing the SELECT statement

- 0; DROP TABLE users

- the SELECT statement is executed, then table *users* is dropped

```
SELECT ... WHERE ... v ...
```

supplied by the client

## \* SQL injection

### 4. changing a user's password

- the client is asked to provide value  $v$ :
- possible values entered by the user for  $v$  (the new password):
  - $x" --$ 
    - setting the password for all users to  $x$
  - $x" WHERE user LIKE "%admin%" --$ 
    - setting the password for all usernames matching the criterion specified by the LIKE operator

UPDATE table

SET password = " $v$ "

WHERE user = " " ...

supplied by the client

- \* SQL injection
- prevention
  - data validation
    - use regular expressions to validate data
    - allow the client to use only certain types of characters in the input data
  - modify problematic characters
    - double single quotes and double quotes; or precede single and double quotes with "\\" (one can use functions for such changes)  
=> single and double quotes in the statement won't modify the statement  
(like in the previous examples)

example: SELECT ... WHERE user = "uname" AND password = "upassword"

user enters: *a* and "*OR 1 = 1 --*" for *uname* and *upassword* (like in a prev. ex.)

double the "

=> statement: SELECT ... WHERE user = "a" AND password = " " " " OR 1 = 1 -- ", which retrieves rows with user *a* and password "*OR 1 = 1 --*"

- \* SQL injection

- prevention

- use parameterized statements

SELECT . . . WHERE user=? AND password=?, where “?” is a parameter

- such a statement has a collection of parameters

## \* data encoding

- protecting the data when the normal security mechanisms in the DBMS are insufficient (e.g., an intruder gets physical access to the server or the data center and steals the drives containing the data, an enemy taps into a communication line, etc)
- storing / transmitting encoded data => the stolen data is illegible for the intruder
- some DBMSs store data (all data or only a part of the data) in an encoded form; if the files containing the database can be copied, using the data in these files is impossible (or extremely difficult, as the decoding cost is enormous)

- \* data encoding
  - codes and ciphers
    - code
      - replace one word or phrase with another word / number / symbol
      - e.g., replace expression *The enemy is moving* with code word *Binary*
    - cipher
      - replace each letter with another letter (number / symbol)
      - e.g., replace each letter in the alphabet with the previous one (B with A, C with B, etc); *The enemy is moving* becomes *Sgd dmdlx hr Inuhmf*
  - steganography and cryptography
    - steganography
      - hide the *existence* of the message
      - e.g., shave messenger's head, write message on scalp, wait for the hair to grow back; use invisible ink; etc

## \* data encoding

- steganography and cryptography
  - cryptography
    - hide the *meaning* of the message – encryption
    - branches in cryptography:
      - transposition & substitution
  - transposition
    - rearrange letters in the message (create anagrams)
    - every letter retains its identity, but changes its position
    - e.g., 6 ways of rearranging 3 letters: *cat, cta, act, atc, tca, tac*
  - substitution
    - pair the letters of the alphabet (randomly, e.g., A's pair could be X)
    - replace each letter in the message with its pair
    - every letter retains its position, but changes its identity

->

- \* data encoding

- substitution

- *plaintext* - message before encryption
- *ciphertext* - message after encryption
- *plain alphabet* - alphabet used to write the message
- *cipher alphabet* - alphabet used to encrypt the message
- examples:

- shift the original alphabet:

- by 1 position (replace *A* with *B*, *B* with *C*, etc)
- by 2 positions (replace *A* with *C*, *B* with *D*, etc), etc

=> there are 25 possible ciphers

- if any rearrangement of the plain alphabet is allowed (don't restrict to shift-based rearrangements) => over  $4 \cdot 10^{26}$  possible rearrangements (i.e., over  $4 \cdot 10^{26}$  ciphers)

- \* data encoding

- substitution

- example:

- *plain alphabet:*

a	b	c	d	...
---	---	---	---	-----
- *cipher alphabet:*

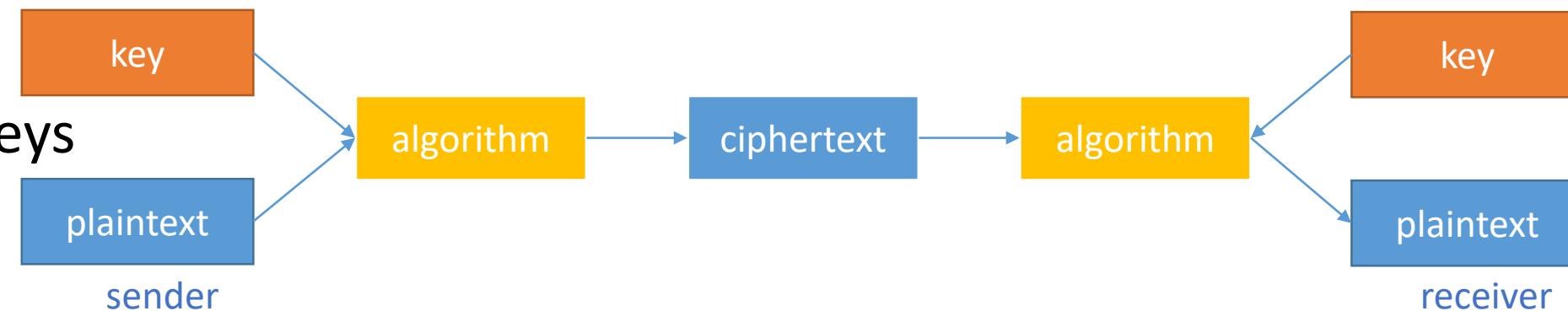
B	C	D	E	...
---	---	---	---	-----

(shift the original alphabet by one position)

- *plaintext:* *the enemy is moving*
- *ciphertext:* *UIF FOFNZ JT NPWJOH*

- \* data encoding

- algorithms and keys



- algorithm - not secret

- general encryption method (e.g., replace each letter in the plain alphabet with a letter from the cipher alphabet; cipher alphabet can be any rearrangement of the plain alphabet)

- key - must be kept secret!

- details of a particular encryption (e.g., the chosen cipher alphabet)
- important - large number of keys:
  - suppose the enemy intercepts a ciphertext; if they know a shift cypher has been used, they only need to check 25 keys; but if any rearrangement of the original alphabet is possible, they need to check  $4 \times 10^{26}$  keys!

## \* data encoding

- the key distribution problem - optional
  - if two parties want to exchange a message, they must first exchange a secret key:
    - meet in person, use couriers (banks used such couriers back in the '70s to communicate securely with their customers)
      - logistical difficulties, costs way too high
  - Whitfield Diffie, Martin Hellman, Ralph Merkle
    - came up with a solution that would enable 2 people to exchange a key without meeting in person and without relying on a 3<sup>rd</sup> party
    - solution based on one-way functions in modular arithmetic

->

## \* data encoding

- the key distribution problem - optional
- examples from *SINGH, Simon, Cartea codurilor – istoria secretă a codurilor și a spargerii lor, Humanitas, 2005*
- two-way functions - they are easy to do and to undo  
example: doubling a number is a two-way function  
double 8 => 16; given 16, it's easy to get back to the original number, 8  
example: turning on a light switch can also be thought of as a function  
turn on switch => light bulb turned on; can easily turn off the bulb by  
turning off the switch
- one-way functions - easy to do and extremely difficult to undo  
example: mixing yellow and blue paint to obtain green paint is a one-way  
function  
example: modular arithmetic function  $453^x \pmod{21997}$

one-way function: $Y^x \pmod{P}$ e.g., $7^x \pmod{11}$ 7, 11 – numbers chosen by Alice & Bob; not secret	Alice	Bob
1	Alice chooses a number A = 3 <i>A - secret</i>	Bob chooses a number B = 6 <i>B - secret</i>
2	Alice computes: $\alpha = 7^A \pmod{11} = 2$	Bob computes: $\beta = 7^B \pmod{11} = 4$
3	Alice <i>sends</i> $\alpha$ to Bob	Bob <i>sends</i> $\beta$ to Alice
exchange	An eavesdropper can intercept $\alpha$ and $\beta$ . This poses no problems, since these numbers are not the key!	
4	Alice computes: $\beta^A \pmod{11} = 4^3 \pmod{11} = 9$	Bob computes: $\alpha^B \pmod{11} = 2^6 \pmod{11} = 9$
the key	Alice and Bob obtained the same number: 9 - the <i>key</i> .	

- Alice and Bob are able to establish a key without meeting beforehand
- Mark (eavesdropper, trying to intercept exchanged messages) only knows the  $7^x \pmod{11}$  function,  $\alpha = 2, \beta = 4$
- Mark doesn't know the key and cannot obtain it, since:
  - he doesn't know A and B (these numbers are kept secret);
  - it's extremely difficult for him obtain A from  $\alpha$  (or B from  $\beta$ ), since  $7^x \pmod{11}$  is a one-way function; this is especially true if the chosen numbers are very large

## \* data encoding methods - examples

### 1. use a secret encryption key

- example

	data:	disciplina baze de date
	secret key:	student

- algorithm
  - a. create a table of codes
    - every character is associated with a number, for instance:

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

- $n$  - number of characters in the table

## \* data encoding methods - examples

1.

- b. divide the message into blocks of length  $L$ , where  $L$  is the number of characters in the key

d	i	s	c	i	p	l	i	n	a		b	a	z	e		d	e		d	a	t	e	
s	t	u	d	e	n	t																	

- replace every character in the message and every character in the key with their associated codes

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

s	t	u	d	e	n	t
19	20	21	04	05	14	20

d	i	s	c	i	p	l	i	n	a		b	a	z	e		d	e		d	a	t	e
04	09	19	03	09	16	12	09	14	01	00	02	01	26	05	00	04	05	00	04	01	20	05

## \* data encoding methods - examples

1.

b. add every number that corresponds to a character in a block with the number of the corresponding character in the key

- if the obtained value is greater than  $n$  (in the example,  $n = 27$ ), compute the remainder of the division by  $n$

d	i	s	c	i	p	l	i	n	a		b	a	z	e		d	e		d	a	t	e
04	09	19	03	09	16	12	09	14	01	00	02	01	26	05	00	04	05	00	04	01	20	05
19	20	21	04	05	14	20	19	20	21	04	05	14	20	19	20	21	04	05	14	20	19	20
23	02	13	07	14	03	05	01	07	22	04	07	15	19	24	20	25	09	05	18	21	12	25

c. replace the obtained numbers with their corresponding characters in the table of codes

=> a string representing the result of the encoding algorithm

- this string can be stored / transmitted
- the obtained code in the example: *wbmgnceagvdgosxtyeruly*

## \* data encoding methods - examples

**1.**

- decoding
  - similar
  - in step b, perform *subtraction (modulo n)* (instead of addition)

**2.** permute the original message and add values (to the characters' codes) for every position

**3.** combine methods **1** and **2**

\* data encoding methods – examples - optional

#### 4. DES (Data Encryption Standard)

- US standard - 1977
- secret key
- specify the encryption key  $K$  - blocks of 64 bits
  - 56 bits used for encoding =>  $2^{56}$  possible combinations (possible keys)
- message divided into 7 character blocks (corresponding to the 56 bits used for encoding)
- characters in the blocks - permuted
- 16 successive encryption rounds with encryption key  $K_i \leftarrow (K, \text{step } i)$
- decryption algorithm
  - identical to the encryption algorithm
  - keys  $K_i$  used in reverse order

\* data encoding methods – examples - optional  
obs.

- some encryption algorithms use 128 bits for the key
- AES (Advanced Encryption Standard)
  - US - new standard (2000)
  - based on the Rijndael algorithm
  - keys - 128, 192, 256 bits (16, 24, 32 bytes)
- .NET
  - classes that implement encryption algorithms with secret key (*symmetric encryption algorithms*)

\* data encoding methods - examples

## 5. the RSA encoding scheme

- Ron Rivest, Adi Shamir, Leonard Adleman

1. randomly choose two large prime numbers  $p$  and  $q$

- compute their product:

$$r = p * q$$

- value  $r$  can be publicly announced

- $p$  and  $q$  are kept secret

- estimation

- $p, q$  – numbers with 63 decimal digits

- $\Rightarrow$  determining  $p$  and  $q$  from  $r$  would require  $40 * 10^{15}$  years on a powerful computer

\* data encoding methods - examples

## 5. the RSA encoding scheme

2. choose number  $c$  such that:

$$c \text{ co-prime with } (p-1)(q-1)$$

$$c > (p-1)(q-1)$$

- $c$  is used as a public encryption key

3. determine a secret decryption key  $d$  that verifies:

$$d*c \equiv 1 \text{ modulo } (p-1)(q-1)$$

\* data encoding methods - examples

## 5. the RSA encoding scheme

4.  $m$  – message (integer)

- encryption
  - determine code  $v$  using the public key  $c$  and the value  $r$

$$v \equiv m^c \text{ modulo } r$$

5. decryption

$$v^d \text{ modulo } r$$

=> value  $m$

- \* data encoding methods - examples

## 5. the RSA encoding scheme

- example
- $p=3; q=5 \Rightarrow r = 15$
- choose  $c = 11 > (p-1)(q-1) = 8$
- determine  $d$  such that:  $d * 11 \equiv 1 \text{ modulo } 8$

$\Rightarrow d = 3 \text{ or } d = 11 \text{ or } d = 19 \dots$  (we'll use  $d = 3$ )

- let  $m = 13$  (text to encode)

$\Rightarrow$

$v = \text{the code} = m^c \text{ modulo } r = 13^{11} \text{ modulo } 15 = 1.792.160.394.037 \text{ modulo } 15 = 7$

- decryption:  $v^d \text{ modulo } r = 7^3 \text{ modulo } 15 = 343 \text{ modulo } 15 = 13$

## References

- [Ta13] ȚÂMBULEA, L., Curs Baze de date, Facultatea de Matematică și Informatică, UBB, 2013-2014
- [Da03] DATE, C.J., An Introduction to Database Systems (8<sup>th</sup> Edition), Addison-Wesley, 2003
- [Si05] SINGH, Simon, Cartea codurilor – istoria secretă a codurilor și a spargerii lor, Humanitas, 2005
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007,  
<http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Ra00] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (2<sup>nd</sup> Edition), McGraw-Hill, 2000
- [WWW1] The Open Web Application Security Project,  
[https://www.owasp.org/index.php/SQL\\_Injection](https://www.owasp.org/index.php/SQL_Injection)
- [WWW4] SQL injection, [http://en.wikipedia.org/wiki/Sql\\_injection](http://en.wikipedia.org/wiki/Sql_injection)

# Database Management Systems

Lecture 6  
Evaluating Relational Operators  
Query Optimization

\* queries – composed of relational operators\*:

- selection ( $\sigma$ )
  - selects a subset of records from a relation
- projection ( $\pi$ )
  - eliminates certain columns from a relation
- join ( $\otimes$ )
  - combines data from two relations
- cross-product ( $R_1 \times R_2$ )
  - returns every record in  $R_1$  concatenated with every record in  $R_2$
- set-difference ( $R_1 - R_2$ )
  - returns records that belong to  $R_1$  and don't belong to  $R_2$
- union ( $R_1 \cup R_2$ )
  - returns all records in relations  $R_1$  and  $R_2$

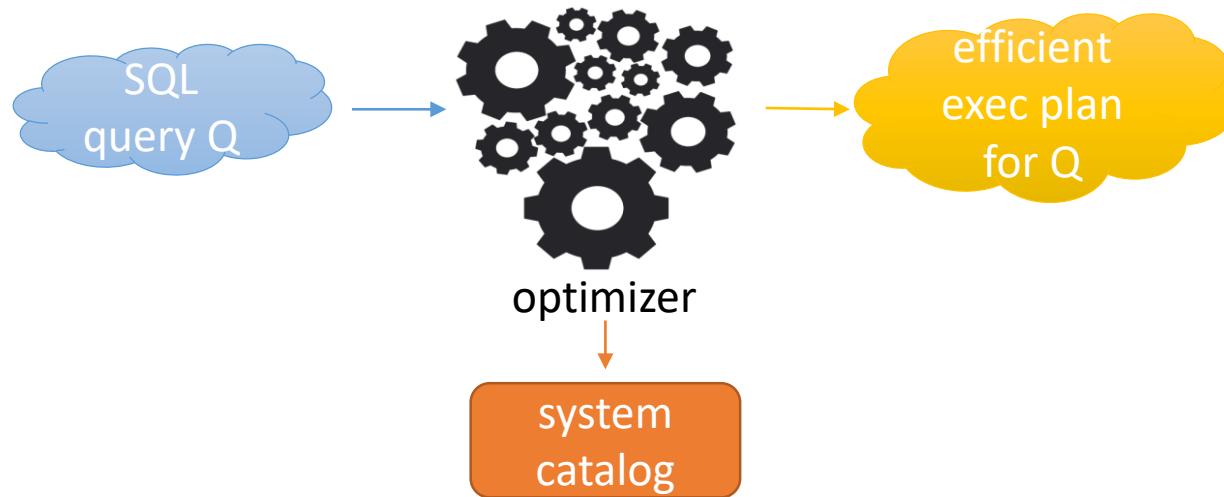
\*Review *Relational Algebra* - lecture notes (*Databases* course)

\* queries – composed of relational operators:

- intersection ( $R_1 \cap R_2$ )
  - returns records that belong to both  $R_1$  and  $R_2$
- grouping and aggregate operators (algebra extensions)
- every operation returns a relation => operations can be composed
- each operator has several implementation algorithms

## \* optimizer

- input: SQL query Q
- output: an efficient execution plan for evaluating Q



- takes into account the manner in which data is stored (such information is available in the system catalog)
- chooses an implementation for each operator in the query, taking into account the size of relations, the size of available buffer pool; the existence of indexes and sort orders; the buffer replacement policy
- determines an application order for the operators in the query

\* the algorithms for various operators are based on 3 techniques:

- iteration:
  - examine iteratively:
    - all tuples in input relations
    - or
    - data entries in indexes, provided they contain all the necessary fields  
(data entries are smaller than data records)
- indexing:
  - used when the query contains a selection condition or a join condition
  - examine only the tuples that meet the condition, using an index
- partitioning:
  - partition the tuples
  - decompose operation into collection of cheaper operations on partitions

\* the algorithms for various operators are based on 3 techniques:

- partitioning:

- partitioning techniques
  - sorting
  - hashing

## \* access paths

- *access path* = way of retrieving tuples from a relation; 2 possibilities:
  - file scan
  - or
  - an index  $I$  + a matching selection condition  $C$
- condition  $C$  matches index  $I$  if  $I$  can be used to retrieve just the tuples\* satisfying  $C$
- if relation  $R$  has an index  $I$  that matches selection condition  $C$ , then there are at least 2 access paths for  $R$  (scan; index + condition)

\*Review *Indexes* - lecture notes (*Databases* course)

\* access paths - example:

- relation *Students*[*SID, Name, City*]
- there's a tree index *I* on *Students* with search key <*Name*>
- query Q:  
SELECT \*  
FROM Students  
WHERE Name = 'Ionescu'
- condition *C*: *Name* = 'Ionescu'
- *C* matches *I*, i.e., index *I* can be used to retrieve only the *Students* tuples satisfying *C*
- the same would be true for, say, *C*: *Name* > 'Ionescu'

\* access paths - example:

- relation *Students*[*SID*, *Name*, *City*]
- there's a hash index *I* on *Students* with search key <*Name*>
- query Q:  
SELECT \*  
FROM Students  
WHERE Name = 'Ionescu'
- condition *C*: *Name* = 'Ionescu'
- *C* matches *I*, i.e., index *I* can be used to retrieve only the *Students* tuples satisfying *C*
- however, condition *Name* > 'Ionescu' doesn't match *I* (since *I* is a hash index; it cannot be used to retrieve just the tuples where *Name* > 'Ionescu')

## \* access paths

- to sum up:
  - condition  $C$ :  $attr \ op \ value$ ,  $op \in \{<, \leq, =, \geq, >\}$
  - condition  $C$  matches index  $I$  if:
    - the search key of  $I$  is  $attr$  and:
      - $I$  is a tree index or
      - $I$  is a hash index and  $op$  is  $=$

## \* access paths

- selectivity of an access path
  - the number of retrieved pages when using the access path to obtain the desired tuples
  - both data pages and index pages are counted
- example:

```
SELECT *
FROM Students
WHERE Name = 'Ionescu'
```

- there are 2 access paths for *Students*:
  - file scan – selectivity could be 1000
  - matching index I with search key <Name> – selectivity could be 3
- most selective access path
  - retrieves the fewest pages, i.e., data retrieval cost is minimized

## \* general selection conditions

- previous examples included only simple conditions of the form  $attr \ op \ value$
- in general, a selection condition can contain one or several terms of the form:
  - $attr \ op \ constant$
  - $attr1 \ op \ attr2$ ,combined with  $\wedge$  and  $\vee$

```
SELECT *
FROM Exams
WHERE SID = 7 AND EDate = '04-01-2019'
```

$$\sigma_{SID=7 \wedge EDate='04-01-2019'}(Exams)$$

## \* general selection conditions

- CNF – conjunctive normal form
  - standard form for general selection conditions
  - condition in CNF:
    - collection of conjuncts connected with the  $\wedge$  operator
    - a *conjunct* has one or more terms connected with the  $\vee$  operator
    - *term*:
      - *attr op constant*
      - *attr1 op attr2*
- example:  
condition  $(EDate < '4-1-2019' \wedge Grade = 10) \vee CID = 5 \vee SID = 3$   
is rewritten in CNF:  
 $(EDate < '4-1-2019' \vee CID = 5 \vee SID = 3) \wedge (Grade = 10 \vee CID = 5 \vee SID = 3)$

## \* general selection conditions matching an index

- index I with search key  $\langle a, b, c \rangle$ , relation  $R[a, b, c, d, e]$

Condition	B+ tree index	Hash index
$a = 10 \text{ AND } b = 5 \text{ AND } c = 2$	Yes	Yes
$a = 10 \text{ AND } b = 5$	Yes	No
$b = 5$	No	No
$b = 5 \text{ AND } c = 2$	No	No
$d = 2$	No	No
$a = 20 \text{ AND } b = 10 \text{ AND } c = 5 \text{ AND } d = 11$	Yes (partly)	Yes (partly)

*Condition – CNF selection condition*

*B+ tree index / Hash index – B+ tree / hash index I matches (Yes) / doesn't match (No) / matches a part of (Yes (partly)) the selection condition*

- for the condition in the last row ( $a = 20 \text{ AND } b = 10 \text{ AND } c = 5 \text{ AND } d = 11$ ):
  - use index I to retrieve tuples satisfying  $a = 20 \text{ AND } b = 10 \text{ AND } c = 5$ , then apply  $d = 11$  to each retrieved tuple

## \* general selection conditions matching an index

- index I1 with search key  $\langle a, b \rangle$
- B+ tree index I2 with search key  $\langle c \rangle$
- relation R[a, b, c, d]

Condition	Indexes
$c < 100 \text{ AND } a = 3 \text{ AND } b = 5$	<ul style="list-style-type: none"><li>- use I1 or I2 to retrieve tuples</li><li>- then check terms in the selection condition that do not match the index for each retrieved tuple</li><li>- e.g., use the B+ tree index to retrieve tuples where <math>c &lt; 100</math>; then apply <math>a = 3</math> AND <math>b = 5</math> to each retrieved tuple</li></ul>

## \* general selection conditions matching an index

- index  $I$ , general selection condition  $C$  (CNF)
- hash index  $I$
- condition  $C$  of the form:
  - $\bigwedge_{i=1}^n T_i$
  - term  $T_i$ :  $attr = value$   
(there are no disjunctions in  $C$ )
- $I$  matches  $C$  if  $C$  contains exactly one term for each attribute in the search key of  $I$

Condition	Hash index with search key $\langle a, b, c \rangle$
$a = 10 \text{ AND } b = 5 \text{ AND } c = 2$	Yes
$a = 10 \text{ AND } b = 5$	No
$b = 5$	No
$b = 5 \text{ AND } c = 2$	No

## \* general selection conditions matching an index

- index  $I$ , general selection condition  $C$  (CNF)
- tree index  $I$
- condition  $C$  of the form:
  - $\bigwedge_{i=1}^n T_i$
  - term  $T_i$ :  $attr \ op \ value; \ op \in \{<, \leq, =, \neq, \geq, >\}$
- $I$  matches  $C$  if  $C$  contains exactly one term for each attribute in a prefix of the search key of  $I$
- examples of prefixes for search key  $\langle a, b, c, d \rangle$ :  $\langle a \rangle$ ,  $\langle a, b, c \rangle$ ;  $\langle a, c \rangle$  and  $\langle b, c \rangle$ , on the other hand, are not prefixes for this search key

Condition	B+ tree index with search key $\langle a, b, c \rangle$
$a = 10 \text{ AND } b = 5 \text{ AND } c = 2$	Yes
$a = 10 \text{ AND } b = 5$	Yes
$b = 5$	No
$b = 5 \text{ AND } c = 2$	No

\* running example - schema

- Students (SID: integer, SName: string, Age: integer)
- Courses (CID: integer, CName: string, Description: string)
- Exams (SID: integer, CID: integer, EDate: date, Grade: integer, FacultyMember: string)
- Students
  - every record has 50 bytes
  - there are 80 records / page
  - 500 pages of Students tuples
- Courses
  - every record has 50 bytes
  - there are 80 records / page
  - 100 pages of Courses tuples

\* running example - schema

- Students (SID: integer, SName: string, Age: integer)
- Courses (CID: integer, CName: string, Description: string)
- Exams (SID: integer, CID: integer, EDate: date, Grade: integer, FacultyMember: string)
- Exams
  - every record has 40 bytes
  - there are 100 records / page
  - 1000 pages of Exams tuples

## \* joins

```
SELECT *
FROM Exams E, Students S
WHERE E.SID = S.SID
```

- algebra:  $E \otimes_{E.SID=S.SID} S$ 
  - to be carefully optimized
  - size of  $E \times S$  is large, so computing  $E \times S$  followed by selection is inefficient
- E
  - M pages
  - $p_E$  records / page
- S
  - N pages
  - $p_S$  records / page
- evaluation: number of I/O operations

## \* joins – implementation techniques

- iteration
  - Simple/Page-Oriented Nested Loops Join
  - Block Nested Loops Join
- indexing
  - Index Nested Loops Join
- partitioning
  - Sort-Merge Join
  - Hash Join
- equality join, one join column
  - join condition:  $E_i = S_j$

## Simple Nested Loops Join

```
foreach tuple e ∈ E do
    foreach tuple s ∈ S do
        if ei == sj then add <e, s> to the result
```

- for each record in the outer relation E, scan the entire inner relation S
- cost
  - $M + p_E * M * N = 1000 + 100 * 1000 * 500 \text{ I/Os} = 1000 + (5 * 10^7) \text{ I/Os}$ 
    - M I/Os – cost of scanning E
    - N I/Os – cost of scanning S
    - S is scanned  $p_E * M$  times (there are  $p_E * M$  records in the outer relation E)

\* E - M pages,  $p_E$  records / page \*

\* S - N pages,  $p_S$  records / page \*

\* 1000 pages \* \* 100 records / page \*

\* 500 pages \* \* 80 records / page \*

## Page-Oriented Nested Loops Join

```
foreach page pe ∈ E do
    foreach page ps ∈ S do
        if  $e_i == s_j$  then add  $\langle e, s \rangle$  to the result
```

- for each page in E read each page in S
- pairs of records  $\langle e, s \rangle$  that meet the join condition are added to the result (where record  $e$  is on page  $pe$ , and record  $s$  – on page  $ps$ )
- refinement of Simple Nested Loops Join

## Page-Oriented Nested Loops Join

```
foreach page pe ∈ E do
    foreach page ps ∈ S do
        if ei == sj then add <e, s> to the result
```

- cost
  - $M + M*N = 1000 + 1000*500 \text{ I/Os} = 501.000 \text{ I/Os}$ 
    - M I/Os – cost of scanning E; N I/Os – cost of scanning S
    - S is scanned M times
    - significantly lower than the cost of Simple Nested Loops Join (improvement - factor of  $p_E$ )
  - if the smaller table (S) is chosen as outer table:  
 $\Rightarrow \text{cost} = 500 + 500 * 1000 \text{ I/Os} = 500.500 \text{ I/Os}$

\* E - M pages,  $p_E$  records / page \*      \* 1000 pages \* \* 100 records / page \*

\* S - N pages,  $p_S$  records / page \*      \* 500 pages \* \* 80 records / page \*

## Block Nested Loops Join

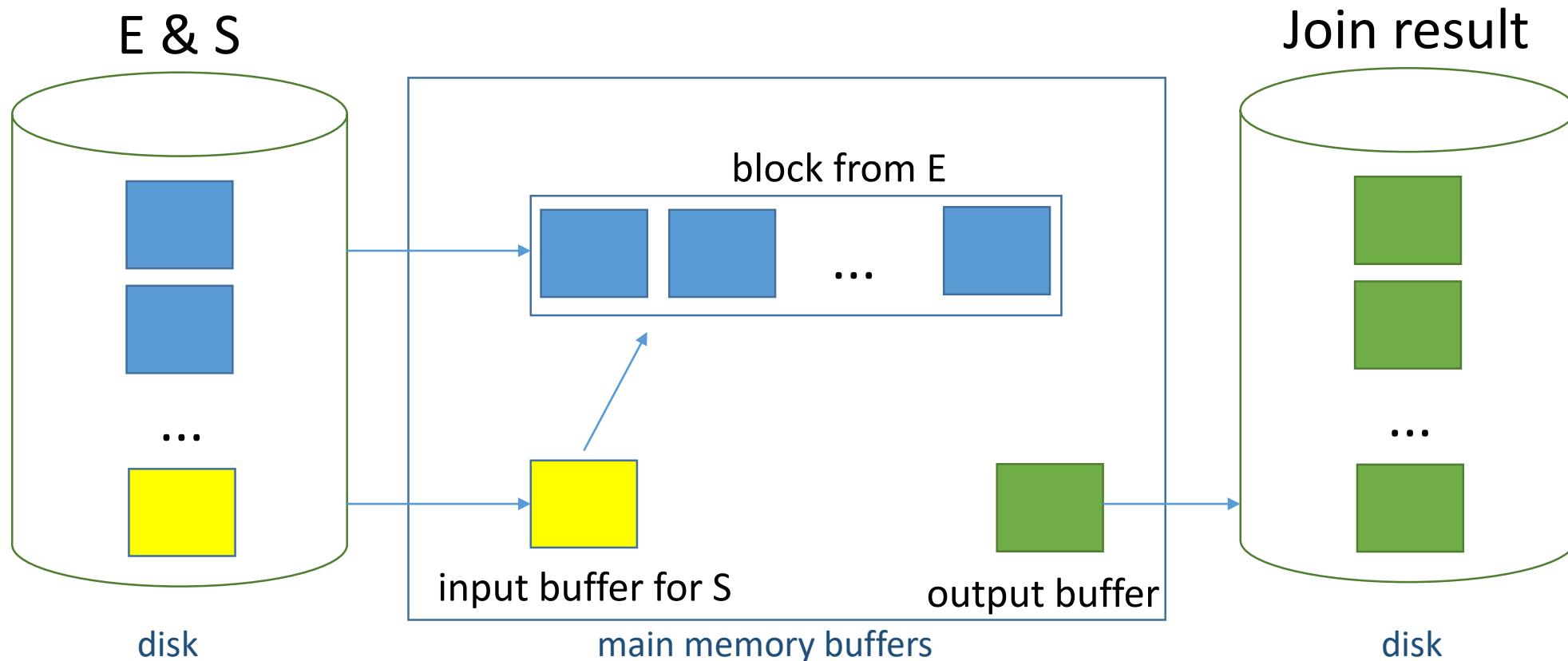
- previously presented join algorithms do not use buffer pages effectively
  - join relations R1 and R2; R1 – the smaller relation
  - assumption – the smaller relation fits in main memory
  - improvement:
    - store smaller relation R1 in memory
    - keep at least 2 extra buffer pages B1 and B2
    - use B1 to read the larger relation R2 (one page at a time)
    - use B2 as the output buffer (i.e., for tuples in the result of the join)
    - for each tuple in R2, search R1 for matching tuples
- => optimal cost: *number of pages in R1 + number of pages in R2*, since R1 is scanned only once, R2 is also scanned only once

## Block Nested Loops Join

- refinement
  - don't store the smaller relation in main memory as is, build an in-memory hash table for it instead
  - the I/O cost remains unchanged, but the CPU cost is usually much lower (since for each tuple in the larger relation, the smaller relation is examined to find matching tuples)

## Block Nested Loops Join

- if there isn't enough main memory to hold one of the input relations:
  - use one buffer page to scan the inner table (e.g., S)
  - use one page for the result
  - use all remaining pages to read a *block* from the outer table (e.g., E)
    - block – set of pages from E that fit in main memory

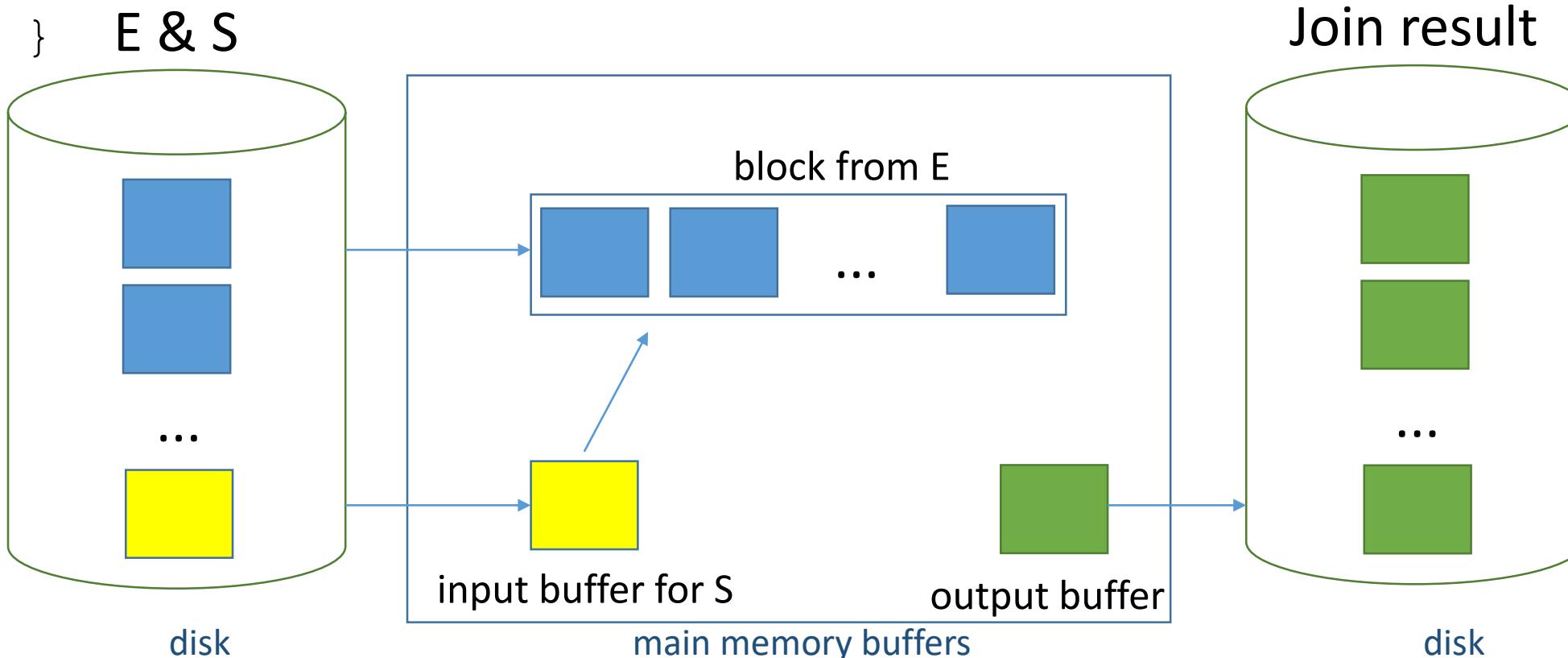


## Block Nested Loops Join

```
foreach block be ∈ E do  
  foreach page ps ∈ S do  
  {
```

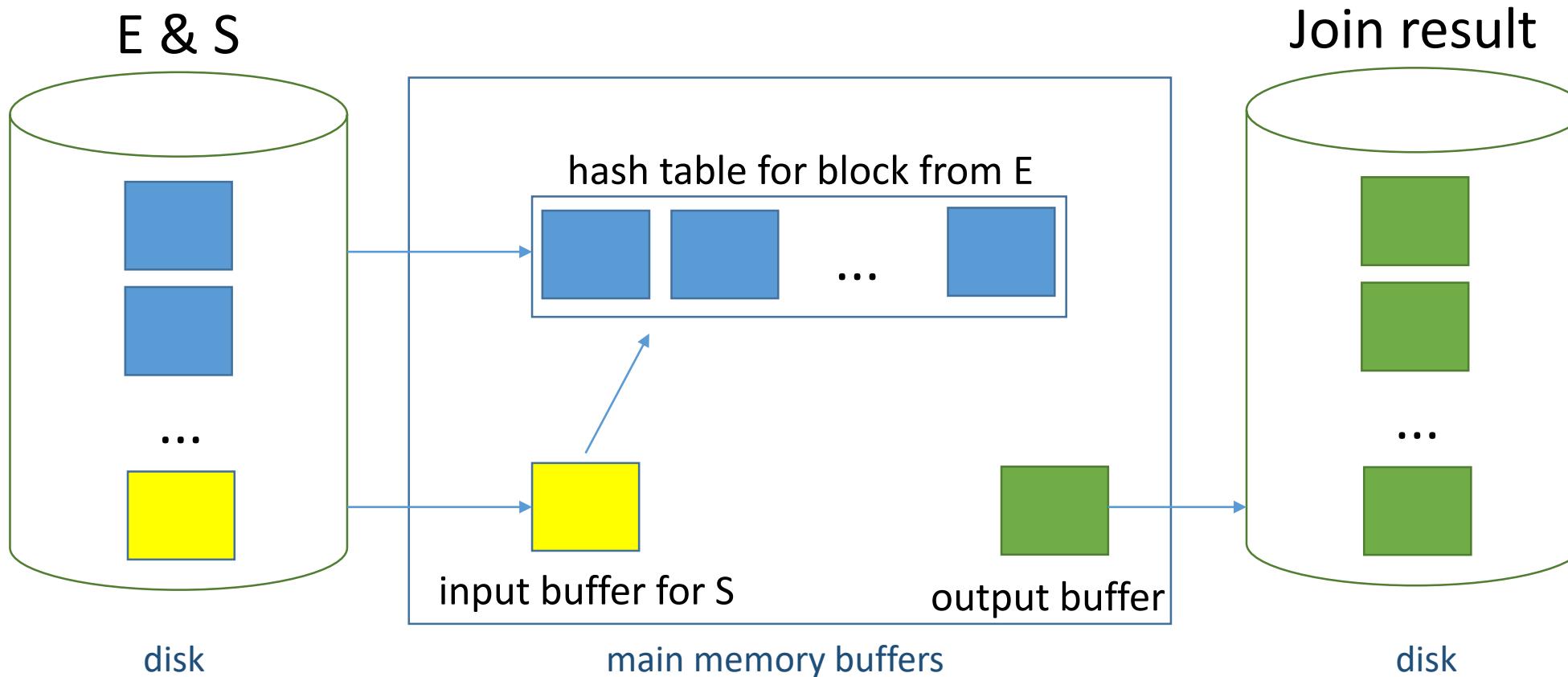
- inner relation S is scanned once for each block in outer relation E
- outer relation E is scanned once

for all pairs of tuples  $\langle e, s \rangle$  that meet the join condition, where  $e \in be$  and  $s \in ps$ ,  
add  $\langle e, s \rangle$  to the result



## Block Nested Loops Join

- refinement to efficiently find matching tuples
  - build main-memory hash table for the block of E
  - trade-off: reduce size of E block



## Block Nested Loops Join

- cost
    - scan of outer table + number of blocks in outer table \* scan of inner table
    - number of outer blocks =  $\left\lceil \frac{\text{number of pages in outer table}}{\text{size of block}} \right\rceil$
    - outer table: Exams (E), a block can hold 100 pages
      - scan cost for E: 1000 I/Os
      - number of blocks:  $\left\lceil \frac{1000}{100} \right\rceil = 10$
      - foreach block in E, scan Students (S):  $10 * 500$  I/Os
- => total cost =  $1000 + 10 * 500 = \mathbf{6000 \text{ I/Os}}$

$$\begin{array}{ll} * E - M \text{ pages}, p_E \text{ records / page} * & * 1000 \text{ pages} * * 100 \text{ records / page} * \\ * S - N \text{ pages}, p_S \text{ records / page} * & * 500 \text{ pages} * * 80 \text{ records / page} * \end{array}$$

## Block Nested Loops Join

- cost
    - scan of outer table + number of blocks in outer table \* scan of inner table
    - number of outer blocks =  $\left\lceil \frac{\text{number of pages in outer table}}{\text{size of block}} \right\rceil$
    - outer table: Exams (E)
      - suppose the buffer has 90 pages available for E, i.e., block of 90 pages  
=> number of blocks:  $\left\lceil \frac{1000}{90} \right\rceil = 12$
      - => S is scanned 12 times
        - scan cost for E: 1000 I/Os
        - foreach block in E, scan Students (S):  $12 * 500$  I/Os
- => total cost =  $1000 + 12 * 500 = \mathbf{7000 \text{ I/Os}}$

\* E - M pages,  $p_E$  records / page \*

\* S - N pages,  $p_S$  records / page \*

\* 1000 pages \* \* 100 records / page \*

\* 500 pages \* \* 80 records / page \*

## Block Nested Loops Join

- cost
    - scan of outer table + number of blocks in outer table \* scan of inner table
    - number of outer blocks =  $\left\lceil \frac{\text{number of pages in outer table}}{\text{size of block}} \right\rceil$
    - outer table: Students (S), block of 100 pages
      - scan cost for S: 500 I/Os
      - number of blocks:  $\left\lceil \frac{500}{100} \right\rceil = 5$
      - for each block in S, scan E:  $5 * 1000 \text{ I/Os}$
- => total cost =  $500 + 5 * 1000 = \mathbf{5500 \text{ I/Os}}$

$$\begin{array}{ll} * E - M \text{ pages}, p_E \text{ records / page} * & * 1000 \text{ pages} * * 100 \text{ records / page} * \\ * S - N \text{ pages}, p_S \text{ records / page} * & * 500 \text{ pages} * * 80 \text{ records / page} * \end{array}$$

## Index Nested Loops Join

```
foreach tuple e in E do
    foreach tuple s in S where ei == sj
        add <e, s> to the result
```

- if there is an index on the join column of S, S can be considered as inner table and the index can be used
- cost
  - $M + (M * p_E) * \text{cost of finding corresponding records in } S)$

\* E - M pages,  $p_E$  records / page \*

\* S - N pages,  $p_S$  records / page \*

\* 1000 pages \* \* 100 records / page \*

\* 500 pages \* \* 80 records / page \*

## Index Nested Loops Join

- for a record  $e$  in  $E$ :
  - the cost of examining the index on  $S$  is:
    - approx. 1.2 for a hash index (typical cost for hash indexes)
    - typically 2-4 for a B+-tree index
  - the cost of reading corresponding records in  $S$ :
    - for a clustered index:
      - plus one I/O for each outer tuple in  $E$  (typically)
    - for a nonclustered index:
      - up to one I/O for each corresponding record in  $S$  (worst case – there are  $n$  matching records in  $S$  located on  $n$  different pages!)

## Index Nested Loops Join

- hash index on SID in Students (Students – inner table)
- scan Exams:
  - cost = 1000 I/Os, with a total of  $100 * 1000$  records
- for each record in Exams:
  - (on average) 1.2 I/Os to obtain the page in the hash index (i.e., the page containing the rid of the matching Students tuple)  
and
  - 1 I/O to retrieve the page in Students that contains the matching tuple (exactly one! – since SID is a key in Students, i.e., there is one matching Students tuple for an exam)

=> cost to retrieve matching Students tuples:  $1000 * 100 * (1.2 + 1) = 220.000$

- total cost:  $1000 + 220.000 = 221.000$  I/Os

\* E - M pages,  $p_E$  records / page \*      \* 1000 pages \* \* 100 records / page \*

\* S - N pages,  $p_S$  records / page \*      \* 500 pages \* \* 80 records / page \*

## Index Nested Loops Join

- hash index on SID in Exams
- scan Students - 500 I/Os, 80\*500 records
- for each record in Students:
  - (on average) 1.2 I/Os to obtain the index page + the cost of reading the matching records in Exams
    - assume exams are uniformly distributed => 2.5 exams for each student (100.000 Exams tuples / 40.000 Students tuples)
    - the cost of retrieving the corresponding records:
      - if the index is clustered: 1 I/O
      - if the index is unclustered: 2.5 I/Os

=> total cost:  $500 + 80*500*(1.2 + 1) = 500 + 88.000 = 88.500$  I/Os or

total cost:  $500 + 80*500*(1.2 + 2.5) = 500 + 148.000 = 148.500$  I/Os

\* E - M pages,  $p_E$  records / page \*      \* 1000 pages \* \* 100 records / page \*

\* S - N pages,  $p_S$  records / page \*      \* 500 pages \* \* 80 records / page \*

## References

- [Ra00] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (2<sup>nd</sup> Edition), McGraw-Hill, 2000
- [Da03] DATE, C.J., An Introduction to Database Systems (8<sup>th</sup> Edition), Addison-Wesley, 2003
- [Ga08] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book, Prentice Hall Press, 2008
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007,  
<http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Si10] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, McGraw-Hill, 2010, <http://codex.cs.yale.edu/avi/db-book/>
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems,  
<http://infolab.stanford.edu/~ullman/fcdb.html>

# Database Management Systems

Lecture 7  
Evaluating Relational Operators  
Query Optimization (II)

- running example - schema
  - Students (SID: integer, SName: string, Age: integer)
  - Courses (CID: integer, CName: string, Description: string)
  - Exams (SID: integer, CID: integer, EDate: date, Grade: integer, FacultyMember: string)
- Students
  - every record has 50 bytes
  - there are 80 records / page
  - 500 pages of Students tuples
- Courses
  - every record has 50 bytes
  - there are 80 records / page
  - 100 pages of Courses tuples

- running example - schema
  - Students (SID: integer, SName: string, Age: integer)
  - Courses (CID: integer, CName: string, Description: string)
  - Exams (SID: integer, CID: integer, EDate: date, Grade: integer, FacultyMember: string)
- Exams
  - every record has 40 bytes
  - there are 100 records / page
  - 1000 pages of Exams tuples

## \* sorting

- can be explicitly required:
  - SELECT ... ORDER BY list
  - SELECT DISTINCT ...
- used by operators like:
  - join
  - union
  - intersection
  - set-difference
  - grouping

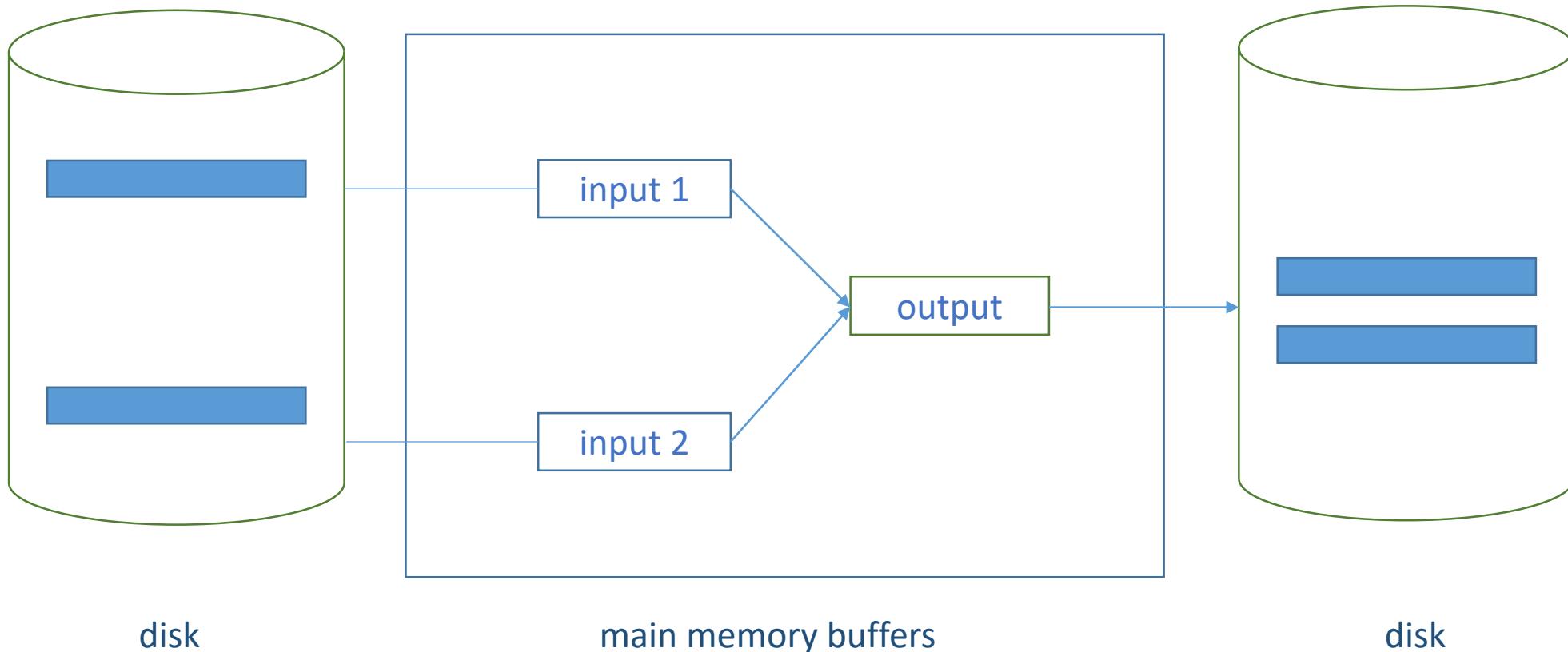
## \* sorting

e.g., the user wants to sort the collection of Courses records by name

- if the data to be sorted fits into available main memory:
  - use an internal sorting algorithm (Quick Sort or any other in-memory sorting algorithm can be used to sort a collection of records that fits into main memory)
- if the data to be sorted doesn't fit into available main memory:
  - use an external sorting algorithm
  - such an algorithm:
    - minimizes the cost of accessing the disk
    - breaks the data collection into subcollections of records
    - sorts the subcollections; a sorted subcollection of records is called a *run*
    - writes runs to disk (into temporary files)
    - merges runs

## Simple Two-Way Merge Sort

- uses 3 buffer pages
- passes over the data multiple times
- can sort large data collections using a small amount of main memory

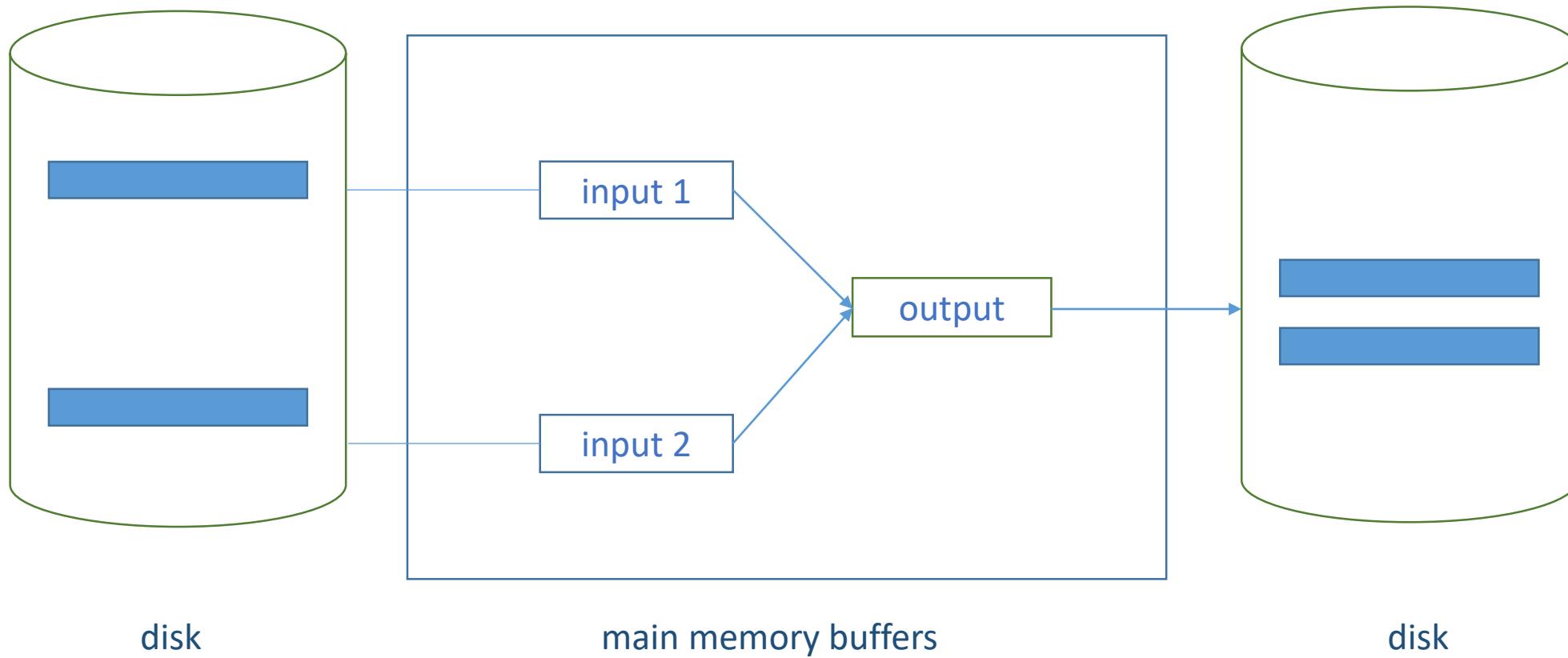


## Simple Two-Way Merge Sort

- pass 0:
  - for each page P in the data collection:
    - read in page P -> sort page P -> save page P to disk  
=> 1-page runs (runs that are 1 page long)  
example: - read in the 1<sup>st</sup> page from Courses, sort the 80 records on it by course name, write out the sorted page to disk (i.e., a *run* that is one page long);  
- read in the 2<sup>nd</sup> page from Courses, sort the 80 records on it by course name, write out sorted page to disk;  
...  
- read in the 100<sup>th</sup> page from Courses, sort the 80 records on it by course name, write out sorted page to disk  
=> 100 1-page runs saved on disk

## Simple Two-Way Merge Sort

- passes 1, 2, ... etc:
  - use 3 buffer pages
  - read and merge pairs of runs from the previous passes
  - produce runs that are twice as long

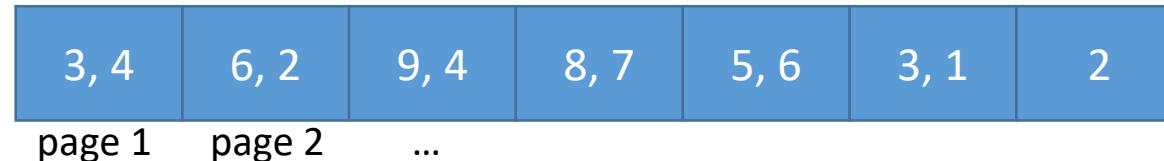


## Simple Two-Way Merge Sort

- e.g., pass 1 (remember, pass 0 produced 100 1-page runs):
  - read in 2 runs from pass 0 (i.e., two pages holding Courses records, each of them sorted in pass 0), using 2 buffer pages
    - merge these runs writing to the 3<sup>rd</sup> available buffer page (the *output* buffer); when the output buffer fills up, write it out to disk (i.e., write a page of 80 sorted records to disk)  
=> a run that is 2 pages long (it contains 160 Courses records, sorted by name)
  - read in and merge the next 2 runs from pass 0 ... => another run that is 2 pages long
  - continue while there are runs to be processed (read in and merged) from pass 0
  - at the end of pass 1 there are 50 2-page runs (each run consists of 2 pages holding 160 records sorted by course name)

## Simple Two-Way Merge Sort

- another example – sort data collection (file of records) with 7 pages:



- only the value of the key is displayed (the key on which the user wants to sort the collection, an integer number in the example)
  - simplifying assumption that allows us to focus on the idea of the algorithm: a page can hold 2 records
  - pass 0
    - read in the collection one page at a time
    - sort each page that is read in
    - write out each sorted page to disk
- => 7 sorted runs that are 1 page long:



## Simple Two-Way Merge Sort

- runs at the end of pass 0:



- pass 1

- read in & merge pairs of runs from pass 0

- produce runs that are twice as long

- read in runs 3, 4 and 2, 6 :

- merge the runs and write to the output buffer

- write the output buffer to disk one page at a time

=> run 2, 3 4, 6

- read in runs 4, 9 and 7, 8 :

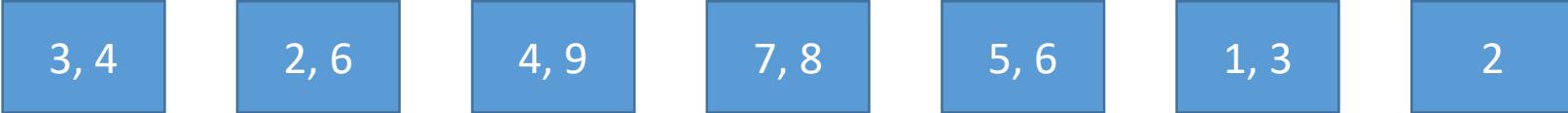
- merge the runs and write to the output buffer

- write the output buffer to disk one page at a time

=> run 4, 7 8, 9

## Simple Two-Way Merge Sort

- runs at the end of pass 0:



- pass 1

- read in runs  and  ...



- read in run  (the last run from pass 0) ...

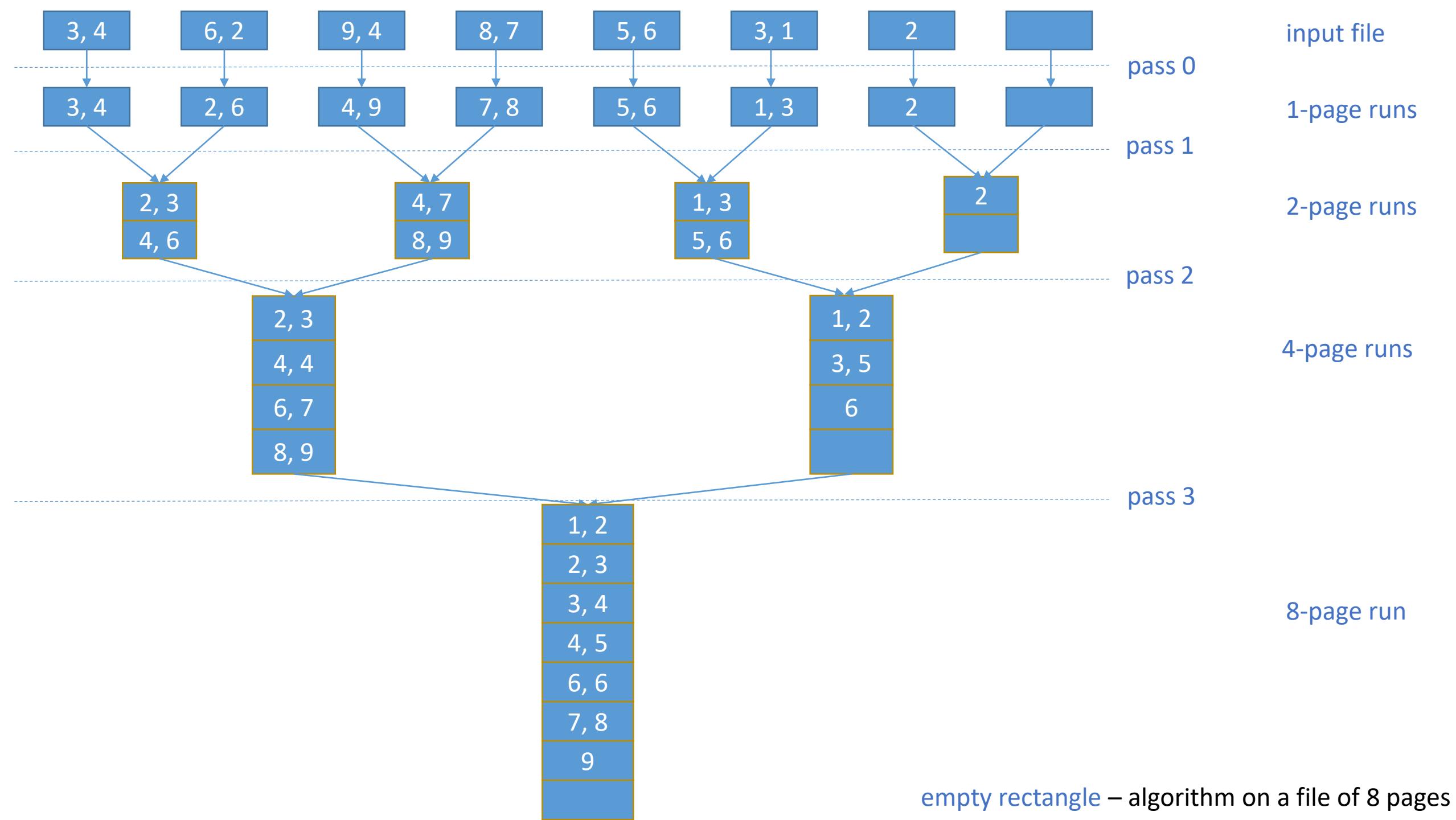


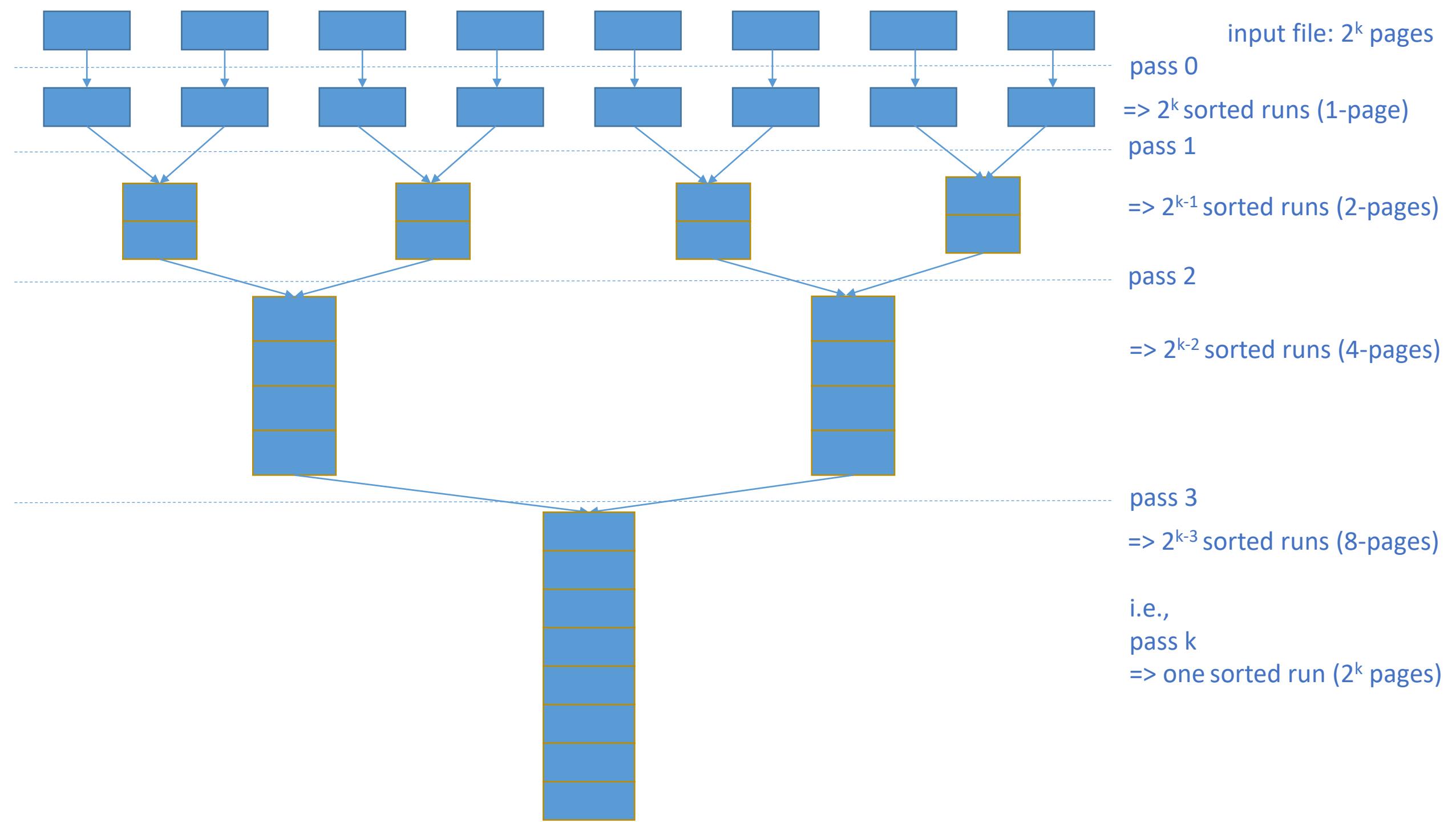
=> 4 sorted runs that are 2 pages long (except for the last run):



## Simple Two-Way Merge Sort

- pass 2
  - read in & merge pairs of runs from pass 1
  - produce runs that are twice as long
- ...
- complete example, with all passes of the algorithm, on the next page      ->





## Simple Two-Way Merge Sort

- at each pass, each page in the input file is: read in, processed, and written out; there are 2 I/O operations per page, per pass

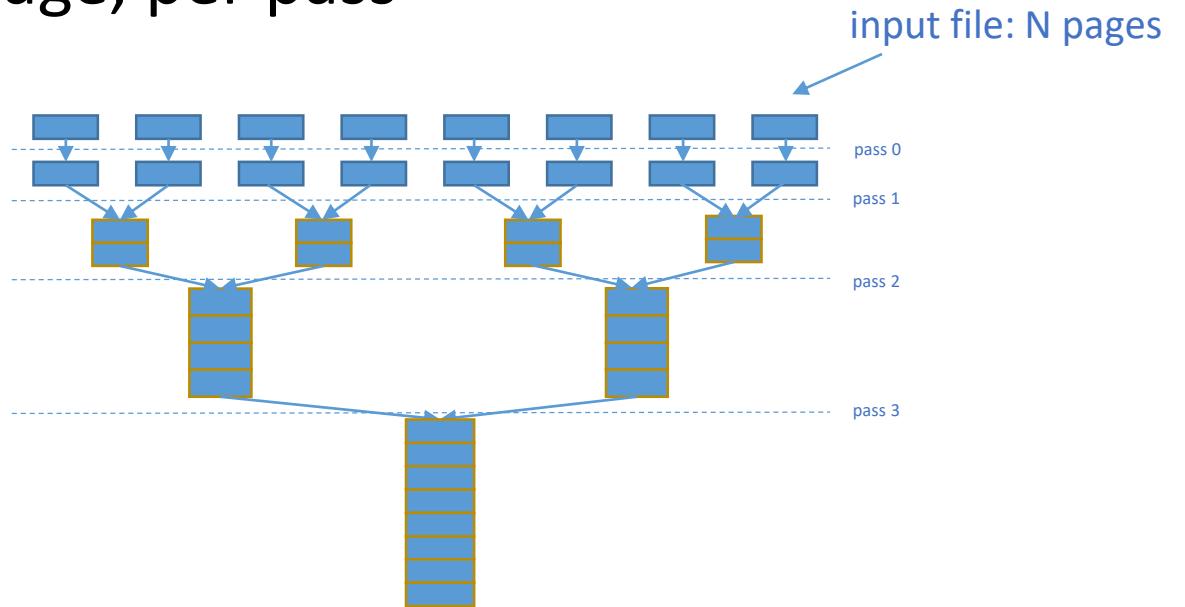
- number of passes:

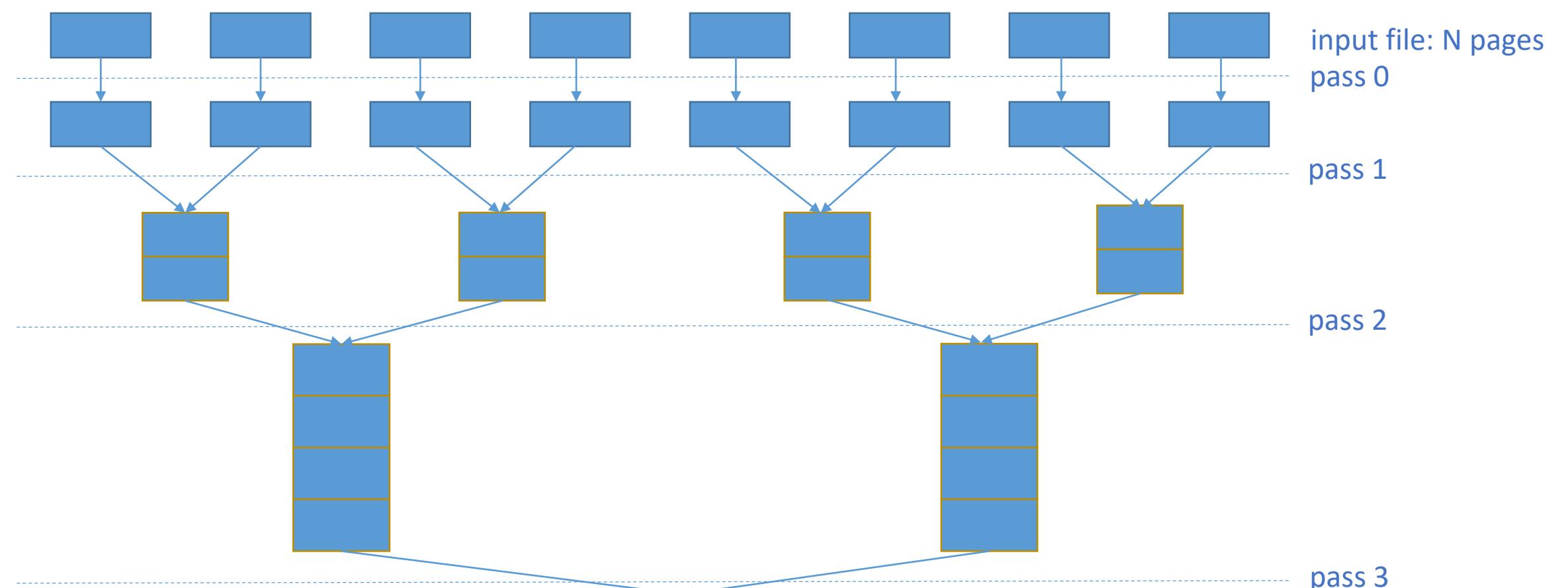
- $\lceil \log_2 N \rceil + 1$

where  $N$  is the number of pages in the file to be sorted

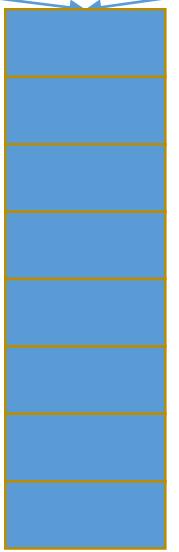
- total cost:

- $2 * \text{number of pages} * \text{number of passes}$
- $2 * N * (\lceil \log_2 N \rceil + 1) \text{ I/Os}$





- in each pass: read / process / write each page in the file
- number of passes:
  - $\lceil \log_2 N \rceil + 1$
- total cost:
  - $2 * N * (\lceil \log_2 N \rceil + 1)$  I/Os



- there are  $N = 8$  pages, 4 passes  
 $\Rightarrow 2 * 8 * 4 = 64$  I/Os
  - or:  $2 * 8 * (\lceil \log_2 8 \rceil + 1) = 2 * 8 * 4 = 64$  I/Os
- $N = 7$  pages, 4 passes  
 $\Rightarrow 2 * 7 * 4 = 56$  I/Os
  - or:  $2 * 7 * (\lceil \log_2 7 \rceil + 1) = 56$  I/Os

## External Merge Sort

- Simple Two-Way Merge Sort: buffer pages are not used effectively
  - for instance, if 200 buffer pages are available, this algorithm still uses only 2 input buffers for passes 1, 2, ...
- generalize the Two-Way Merge Sort algorithm to effectively use the available main memory and minimize the number of passes

->

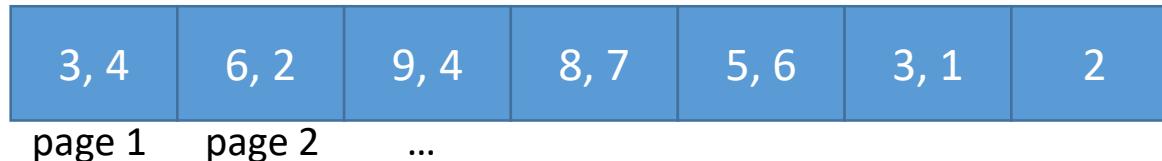
## External Merge Sort

- input file to be sorted: N pages
- B buffer pages are available
- pass 0:
  - use B buffer pages
  - read in B pages at a time and sort them in memory  
=>  $\left\lceil \frac{N}{B} \right\rceil$  runs of B pages each (except for the last one, which may be smaller)

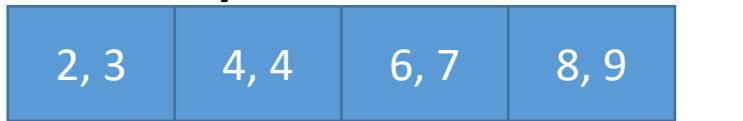
->

## External Merge Sort

- consider again the input file in the previous example:



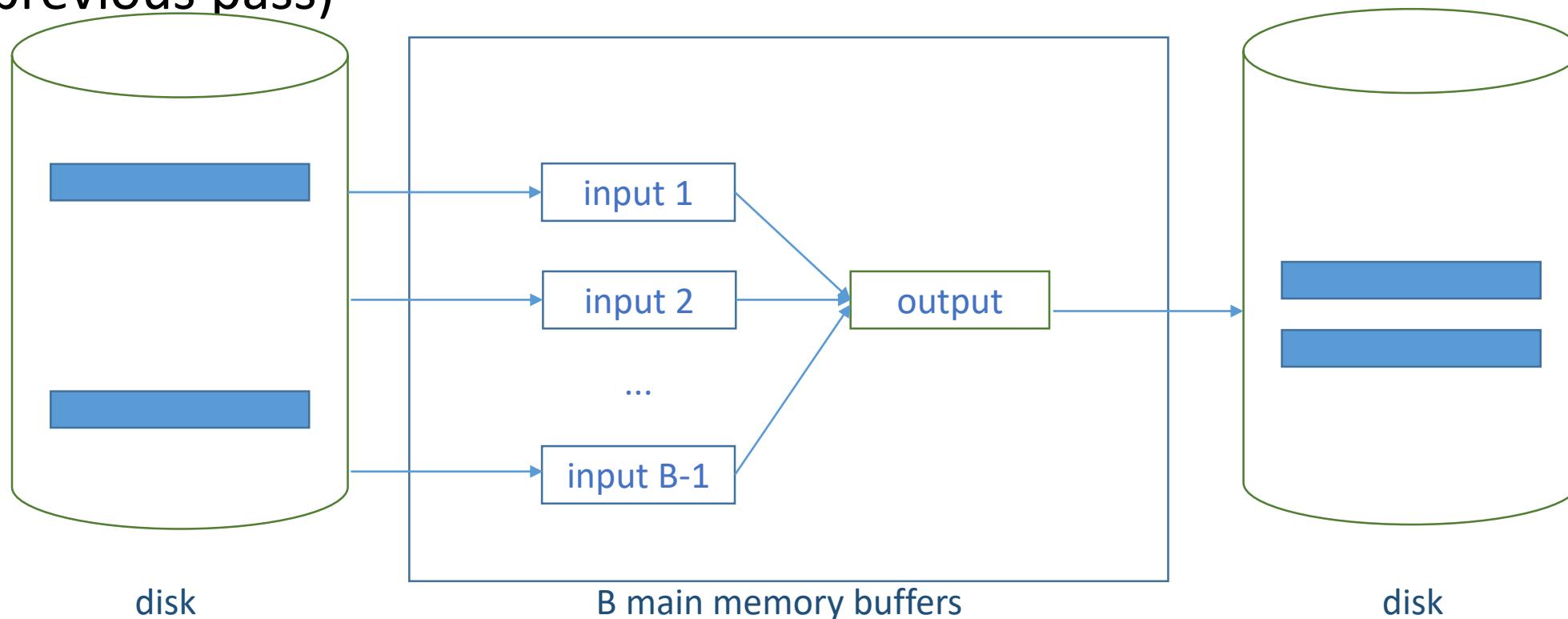
- $N = 7$  (number of pages in the file)
- $B = 4$  (there are 4 available buffer pages)
- pass 0 produces  $\left\lceil \frac{N}{B} \right\rceil = \left\lceil \frac{7}{4} \right\rceil = 2$  runs:
  - use all 4 buffer pages
  - read in 4 pages:   
3, 4      6, 2      9, 4      8, 7
  - sort the pages in memory, write to disk a run that is 4 pages long:



- read in remaining 3 pages:   
5, 6      3, 1      2
- sort pages in memory, write to disk run of 3 pages:   
1, 2      3, 5      6

## External Merge Sort

- input file to be sorted:  $N$  pages
- $B$  buffer pages are available
- pass 1, 2 ...:
  - use  $B-1$  pages for input, and one page for output
  - perform a  $(B-1)$ -way merge in each pass (i.e., merge  $B-1$  runs from the previous pass)



## External Merge Sort

- runs at the end of pass 0: 

- pass 1
  - read in & merge the first  $B - 1 = 4 - 1 = 3$  runs from pass 0
  - pass 0 produced only 2 runs in this example; read in and merge these 2 runs:

=> run



## External Merge Sort

- another example:
  - 5 buffer pages  $B = 5$
  - sort file with 108 pages  $N = 108$
- pass 0
  - use all 5 buffer pages
  - read in the first 5 pages of the file, sort them in memory, write the resulting run to disk (5 pages long)
  - read in the next 5 pages of the file, sort them in memory, write the resulting run to disk (5 pages long)
  - ...
  - read in the remaining 3 pages of the file, sort them in memory, write the resulting run to disk (3 pages long)
  - 21 runs are 5 pages long; 1 run is 3 pages long

## External Merge Sort

- another example:  $B = 5$ ,  $N = 108$
- pass 0
  - at the end of pass 0 there are  $\left\lceil \frac{N}{B} \right\rceil = \left\lceil \frac{108}{5} \right\rceil = 22$  runs
- pass 1
  - use  $B-1 = 5-1 = 4$  pages for input, and one page for output
  - do a 4-way merge: read in and merge 4 runs from the previous pass
  - read in the first 4 runs from pass 0 (each run into an input buffer)
    - merge the runs and write to the output buffer
    - write the output buffer to disk one page at a time
      - => a run that is 20 pages long (4 runs from pass 0 times 5 pages per run)
  - read in the next 4 runs from pass 0; merge the runs and write to the output buffer; write the output buffer to disk one page at a time
    - => another run (20 pages long)
- ...

## External Merge Sort

- another example:  $B = 5, N = 108$
- pass 0
  - at the end of pass 0 there are 22 runs
- pass 1
  - read in the last 2 runs from pass 0 (one has 5 pages, the other one has 3 pages)
    - merge the runs and write to the output buffer; write the output buffer to disk one page at a time  
=> the last run (8 pages long)
  - at the end of pass 1 there are  $\left\lceil \frac{22}{4} \right\rceil = 6$  runs
  - 5 runs are 20 pages long; 1 run is 8 pages long

## External Merge Sort

- another example:  $B = 5, N = 108$
- pass 1
  - at the end of pass 1 there are 6 runs
- pass 2
  - 4-way merge
  - read in the first 4 runs from pass 1
    - merge the runs and write to the output buffer; write the output buffer to disk one page at a time  
=> a run that is 80 pages long (4 runs from pass 1 times 20 pages per run)
  - read in the remaining 2 runs from pass 1 (20 and 8 pages, respectively)  
=> a run that is 28 pages long
  - at the end of pass 2 there are  $\left\lceil \frac{6}{4} \right\rceil = 2$  runs

## External Merge Sort

- another example:  $B = 5, N = 108$
- pass 2
  - at the end of pass 2 there are 2 runs
- pass 3
  - read in the 2 runs from pass 2 and merge them  
=> a run that is 108 pages long, representing the sorted file

## External Merge Sort

- cost
  - $N$  – number of pages in the input file,  $B$  – number of available pages in the buffer
  - at each pass: read / process / write each page
  - number of passes:  $\lceil \log_{B-1} [N/B] \rceil + 1$
  - total cost:  $2 * N * \left( \left\lceil \log_{B-1} \left[ \frac{N}{B} \right] \right\rceil + 1 \right)$  I/Os
- previous example:  $B = 5$  and  $N = 108$ , with 4 passes over the data
  - cost:  
$$2 * 108 * 4 = 864 \text{ I/Os}$$
  - $$2 * 108 * \left( \left\lceil \log_{5-1} \left[ \frac{108}{5} \right] \right\rceil + 1 \right) = 216 * (\lceil \log_4 22 \rceil + 1) = 216 * 4 = 864 \text{ I/Os}$$

- $B$  buffer pages
- sort file with  $N$  pages

### Simple Two-Way Merge Sort

pass 0 =>  $N$  runs

$$\text{number of passes} = \lceil \log_2 N \rceil + 1$$

### External Merge Sort

pass 0 =>  $\left\lceil \frac{N}{B} \right\rceil$  runs

$$\text{number of passes} = \left\lceil \log_{B-1} \left\lceil \frac{N}{B} \right\rceil \right\rceil + 1$$

- External Merge Sort – reduced number of:
  - runs produced by the 1<sup>st</sup> pass
  - passes over the data
- $B$  is usually large => significant performance gains

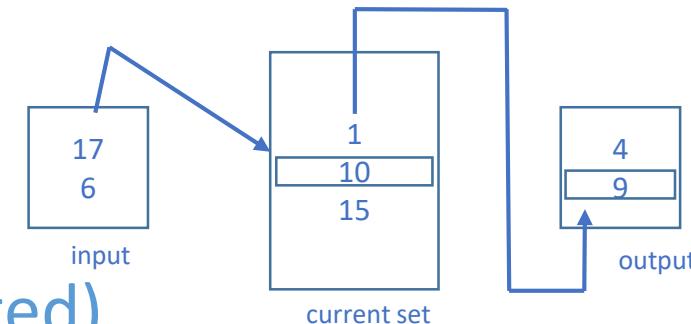
## External Merge Sort – number of passes for different values of N and B

N	B = 3	B = 5	B = 9	B = 17	B = 129	B = 257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

- \* minimize the number of runs - optional

- external merge sort
  - $N$  pages in the file,  $B$  buffer pages  $\Rightarrow \lceil N/B \rceil$  runs of  $B$  pages each
- improvement
  - algorithm known to produce sorted runs of approximately  $2*B$  pages (on average)
  - use 1 page as an input buffer, 1 page as an output buffer
  - the remaining buffer pages are collectively referred to as the *current set*

- example - sort file in ascending order on some key  $k$ :
- repeatedly pick record  $r$  with smallest  $k$  in current set such that  $k$  is  $>$  the largest  $k$  in the output buffer
- append  $r$  to output buffer (the output buffer is kept sorted)
- use the extra space in the current set to bring in the next tuple from the input buffer



\* minimize the number of runs - optional

- process all tuples in the input buffer, then read in the next page of the file
- when the output buffer fills up, write it to disk (add its content to the run that is currently being built)
- the current run is completed when every  $k$  value in the current set is  $<$  the largest  $k$  value in the output buffer; when this happens, the output buffer is written out (its content becomes the last page in the current run), and a new run is started

## Sort-Merge Join

- equality join, one join column:  $E \otimes_{i=j} S$  ( $i^{\text{th}}$  column's value in  $E = j^{\text{th}}$  column's value in  $S$ )
- sort  $E$  and  $S$  on the join column (if not already sorted):
  - for instance, by using External Merge Sort  
=> *partitions* = groups of tuples with the same value in the join column
- merge  $E$  and  $S$ ; look for tuples  $e$  in  $E$ ,  $s$  in  $S$  such that  $e_i = s_j$ :
  - while *current*  $e_i < \text{current } s_j$ 
    - advance the scan of  $E$
  - while *current*  $e_i > \text{current } s_j$ 
    - advance the scan of  $S$
  - if *current*  $e_i = \text{current } s_j$ 
    - output joined tuples  $\langle e, s \rangle$ , where  $e$  and  $s$  are in the current partition (i.e., they have the same value in the  $i^{\text{th}}$  and  $j^{\text{th}}$  column, respectively)
    - there could be multiple tuples in  $E$  with the same value in the  $i^{\text{th}}$  column as the current tuple  $e$  (same is true for  $S$ )

## Sort-Merge Join

- partitions are illustrated on tables Students and Exams below (join column SID in both tables):

SID	SName	Age
20	Ana	20
30	Dana	20
40	Dan	20
45	Daniel	20
50	Ina	20

SID	CID	EDate	Grade	FacultyMember
30	2	20/1/2018	10	Ionescu
30	1	21/1/2018	9.99	Pop
45	2	20/1/2018	9.98	Ionescu
45	1	21/1/2018	9.98	Pop
45	3	22/1/2018	10	Stan
50	2	20/1/2018	10	Ionescu

## Sort-Merge Join

- during the merging phase, E is scanned once; every partition in S is scanned as many times as there are matching tuples in the corresponding partition in E

	... i <sup>th</sup> column	...
E	1	
	2	
	2	
	2	
	3	
	...	

	j <sup>th</sup> column	...
S	0	
	2	
	2	partition P
	4	
	...	
	...	

- for instance, partition P in the above table S is scanned 3 times, once per matching tuple in the corresponding partition in E
- there are 6 output joined tuples  $\langle e, s \rangle$  for partition P
- this algorithm avoids the enumeration of the cross-product: tuples in a partition in E are compared only with the S tuples in the same partition!

## Sort-Merge Join

- cost:
  - sorting E
    - cost:  $O(M \log M)$
  - sorting S
    - cost:  $O(N \log N)$
  - cost of merging:  $M + N$  I/Os, assuming partitions in S are scanned only once
    - worst-case scenario:  $M * N$  I/Os (when all records in E and S have the same value in the join column)

\* E - M pages; S - N pages\*

## Sort-Merge Join ( $\text{Exams} \otimes_{\text{Exams.SID}=\text{Students.SID}} \text{Students}$ )

- 100 buffer pages
  - sort Exams
    - 2 passes => cost:  $2 * 2 * 1000 = 4000$  I/Os
  - sort Students
    - 2 passes => cost:  $2 * 2 * 500 = 2000$  I/Os
  - merging phase
    - cost:  $1000 + 500 = 1500$  I/Os
  - total cost:  $4000 + 2000 + 1500 = 7500$  I/Os
    - similar to the cost of Block Nested Loops Join

$$\begin{array}{ll} * E - M \text{ pages}, p_E \text{ records / page} * & * 1000 \text{ pages} * * 100 \text{ records / page} * \\ * S - N \text{ pages}, p_S \text{ records / page} * & * 500 \text{ pages} * * 80 \text{ records / page} * \end{array}$$

## Sort-Merge Join ( $\text{Exams} \otimes_{\text{Exams.SID}=\text{Students.SID}} \text{Students}$ )

- 35 buffer pages
  - sort Exams
    - 2 passes => cost:  $2 * 2 * 1000 = 4000$  I/Os
  - sort Students
    - 2 passes => cost:  $2 * 2 * 500 = 2000$  I/Os
  - merging phase
    - cost:  $1000 + 500 = 1500$  I/Os
  - total cost:  $4000 + 2000 + 1500 = 7500$  I/Os
    - ex: compute cost of BNLJ and compare

$$\begin{array}{lll} * E - M \text{ pages}, p_E \text{ records / page} * & * 1000 \text{ pages} * & * 100 \text{ records / page} * \\ * S - N \text{ pages}, p_S \text{ records / page} * & * 500 \text{ pages} * & * 80 \text{ records / page} * \end{array}$$

## Sort-Merge Join ( $\text{Exams} \otimes_{\text{Exams.SID}=\text{Students.SID}} \text{Students}$ )

- 300 buffer pages
  - sort Exams
    - 2 passes => cost:  $2 * 2 * 1000 = 4000$  I/Os
  - sort Students
    - 2 passes => cost:  $2 * 2 * 500 = 2000$  I/Os
  - merging phase
    - cost:  $1000 + 500 = 1500$  I/Os
  - total cost:  $4000 + 2000 + 1500 = 7500$  I/Os
    - ex: compute cost of BNLJ and compare

$$\begin{array}{lll} * E - M \text{ pages}, p_E \text{ records / page} * & * 1000 \text{ pages} * & * 100 \text{ records / page} * \\ * S - N \text{ pages}, p_S \text{ records / page} * & * 500 \text{ pages} * & * 80 \text{ records / page} * \end{array}$$

## References

- [Ra00] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (2<sup>nd</sup> Edition), McGraw-Hill, 2000
- [Da03] DATE, C.J., An Introduction to Database Systems (8<sup>th</sup> Edition), Addison-Wesley, 2003
- [Ga08] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book, Prentice Hall Press, 2008
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007,  
<http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Si10] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, McGraw-Hill, 2010, <http://codex.cs.yale.edu/avi/db-book/>
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems,  
<http://infolab.stanford.edu/~ullman/fcdb.html>

# Database Management Systems

Lecture 8  
Evaluating Relational Operators  
Query Optimization (III)

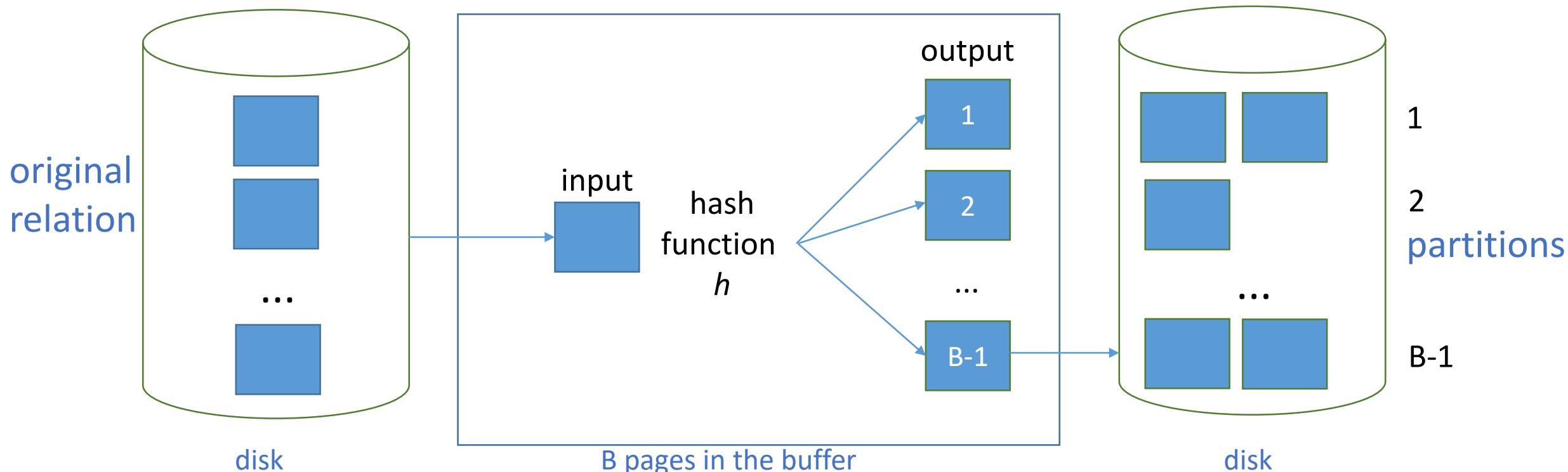
- running example - schema
  - Students (SID: integer, SName: string, Age: integer)
  - Courses (CID: integer, CName: string, Description: string)
  - Exams (SID: integer, CID: integer, EDate: date, Grade: integer, FacultyMember: string)
- Students
  - every record has 50 bytes
  - there are 80 records / page
  - 500 pages of Students tuples
- Exams
  - every record has 40 bytes
  - there are 100 records / page
  - 1000 pages of Exams tuples
- Courses
  - every record has 50 bytes
  - there are 80 records / page
  - 100 pages of Courses tuples

## Hash Join - equality join, one join column: $E \otimes_{i=j} S$

- phases: partitioning (building phase) & probing (matching phase)
- partitioning phase:
  - there are  $B$  pages available in the buffer:
    - use one page as the input buffer page
    - and the remaining  $B-1$  pages as output buffer pages
  - choose a hash function  $h$  that distributes tuples uniformly to one of  $B-1$  partitions
  - hash  $E$  and  $S$  on the join column (the  $i^{\text{th}}$  column of  $E$ , the  $j^{\text{th}}$  column of  $S$ ) with the same hash function  $h$

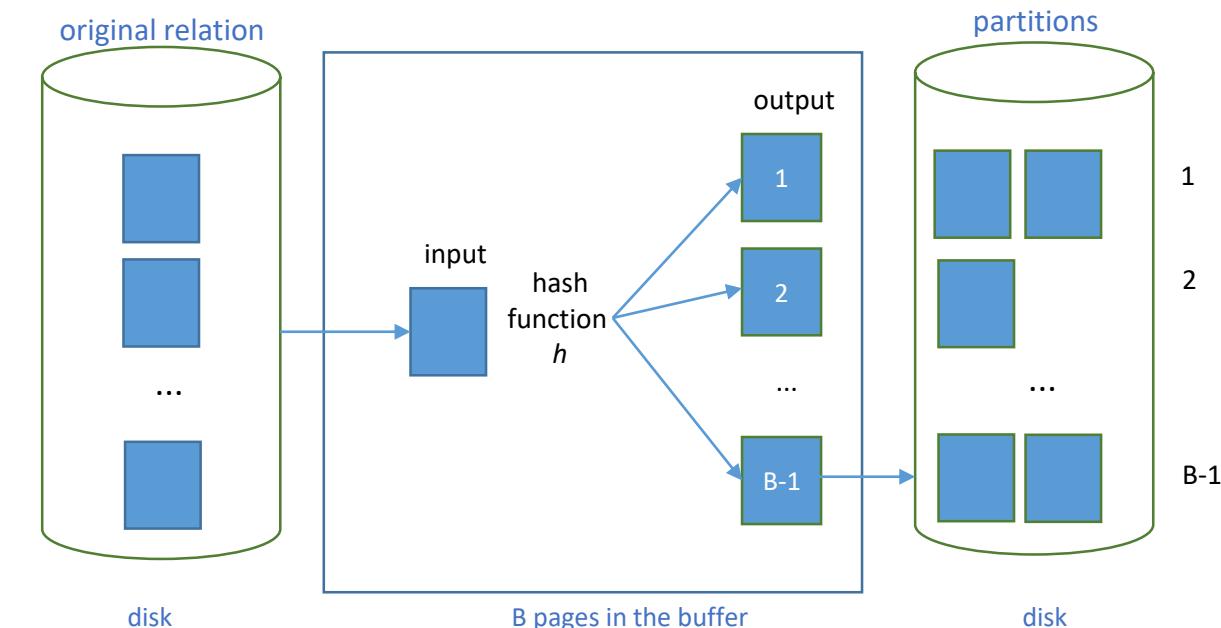
## Hash Join

- hash E on the join column with hash function  $h$  (similarly for S):
  - for each tuple  $e$  in E, compute  $h(e_i)$  ( $e_i$ : the value of the  $i^{\text{th}}$  column in tuple  $e$ )
  - add tuple  $e$  to the output buffer page that it is hashed to by  $h$  (buffer page  $h(e_i)$ )
  - when an output buffer page fills up, flush the page to disk



## Hash Join

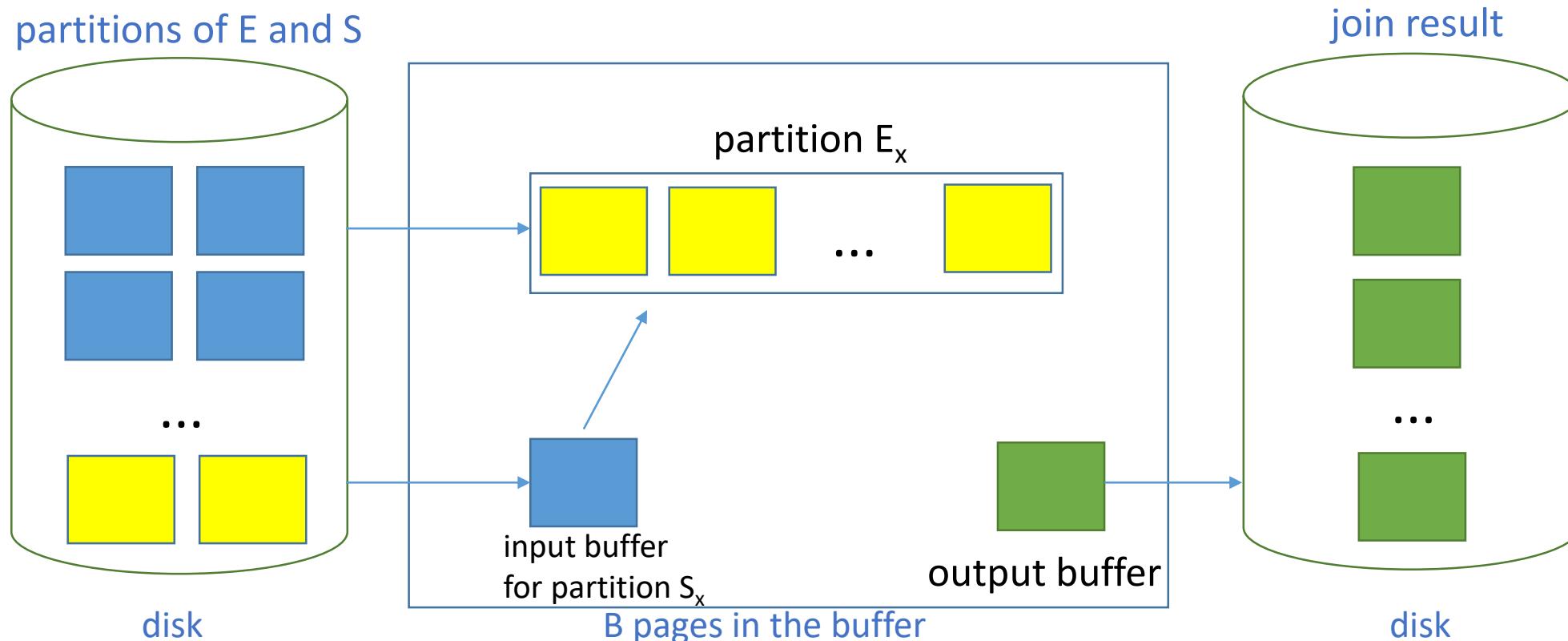
- partitioning phase => *partitions* of E ( $E_1, E_2, \text{etc}$ ) and S ( $S_1, S_2, \text{etc}$ ) on disk
- partition = collection of tuples that have the same hash value
- tuples in partition  $E_1$  can only join with tuples in partition  $S_1$  (they cannot join with tuples in partitions  $S_2$  or  $S_3$ , for instance, since these tuples have a different hash value)
- so to compute the join, we need to scan E and S only once (provided any partition of E fits in main memory)
- when reading in a partition  $E_k$  of E, we must scan only the corresponding partition  $S_k$  of S to find matching tuples (compare tuples  $e$  in  $E_k$  with tuples  $s$  in  $S_k$  to test the join condition *value of  $i^{th}$  column in E = value of  $j^{th}$  column in S*)



## Hash Join

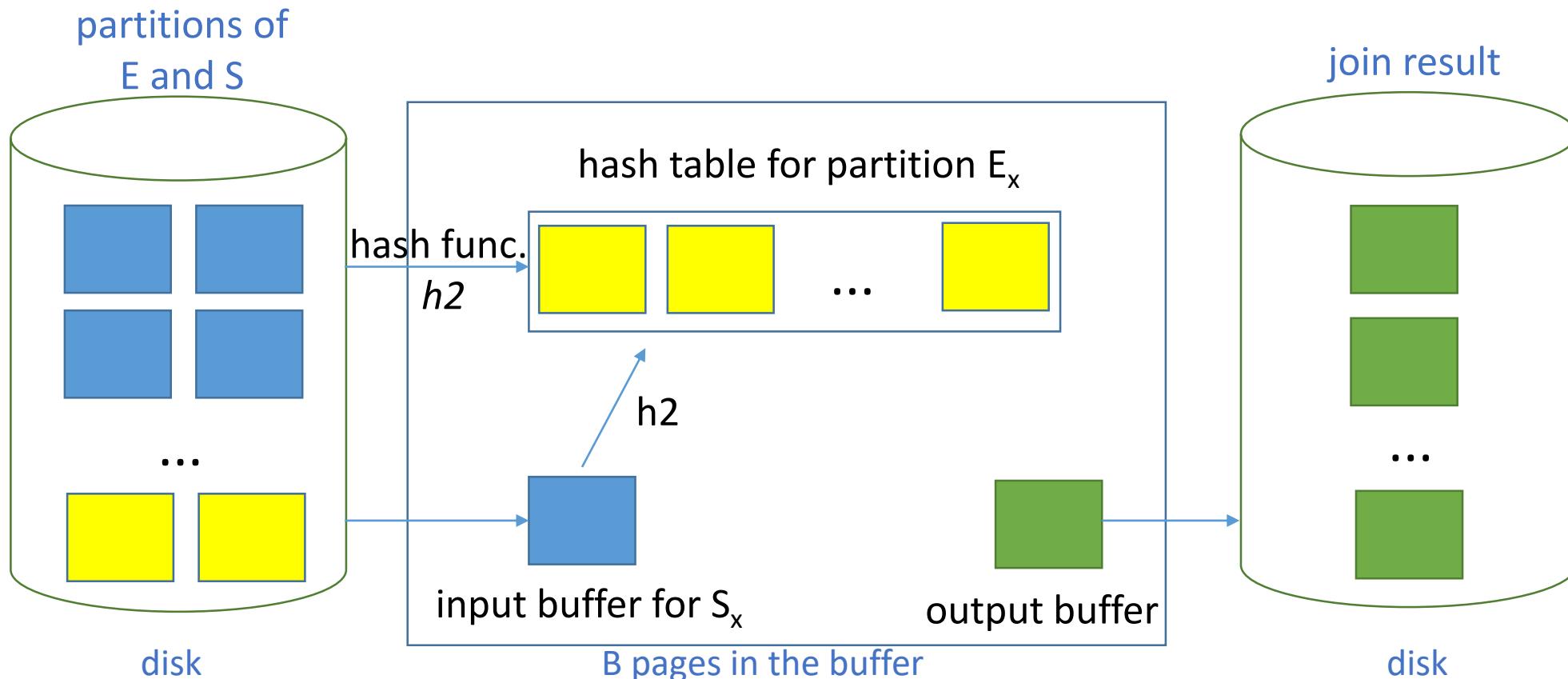
- probing phase:

- read in a partition of the smaller relation (e.g., E) and scan the corresponding partition of S for matching tuples
- use one page as the input buffer for S, one page as the output buffer, and the remaining pages to read in partitions of E



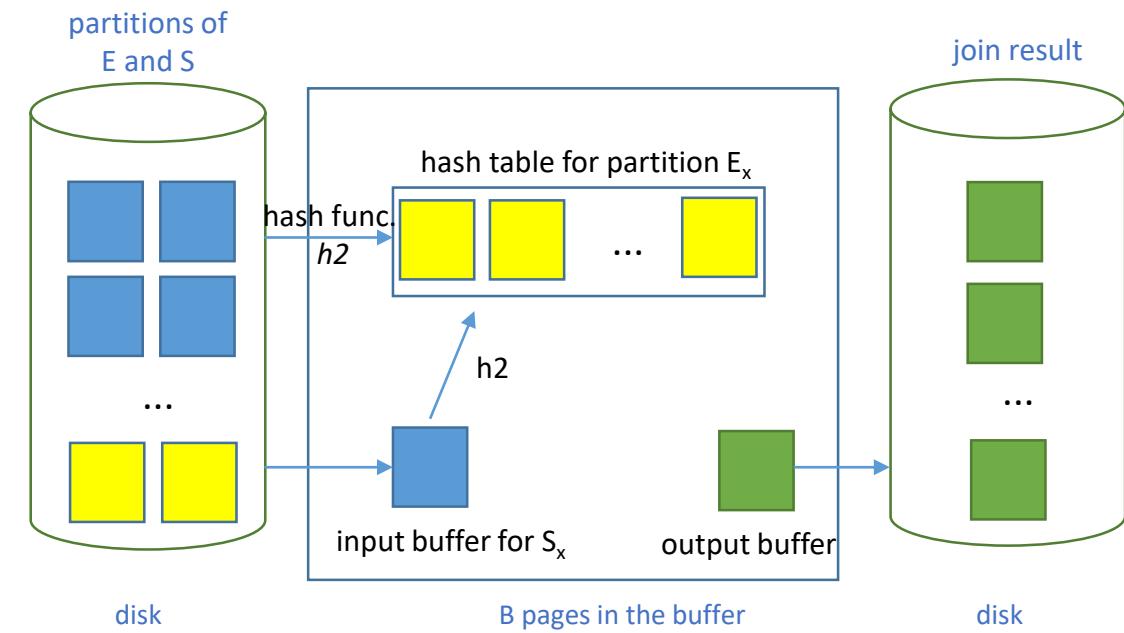
## Hash Join

- probing phase:
  - in practice, to reduce CPU costs, an in-memory hash table is built, using a different function  $h2$ , for the E partition



## Hash Join

- probing phase:
  - in practice, to reduce CPU costs, an in-memory hash table is built, using a different function  $h2$ , for the E partition
- consider a partition  $E_x$  of E
- build in-memory hash table for  $E_x$  using hash function  $h2$  (the function is applied to the join column of E)
- for each tuple  $s$  in partition  $S_x$ , find matching tuples in the hash table using the hash value  $h2(s_j)$
- result tuples  $\langle e, s \rangle$  are written to output buffer
- once partitions  $E_x$  and  $S_x$  are processed, the hash table is emptied (to prepare for the next partition)



## Hash Join

- cost:
    - partitioning:
      - both E and S are read and written once => cost:  $2*(M+N)$  I/Os
    - probing:
      - scan each partition once => cost:  $M+N$  I/Os
- => total cost:  $3*(M+N)$  I/Os
- assumption: each partition fits into memory during probing
  - $3*(1000 + 500) = 4500$  I/Os

\* E - M pages,  $p_E$  records / page \*

\* S - N pages,  $p_S$  records / page \*

\* 1000 pages \* \* 100 records / page \*

\* 500 pages \* \* 80 records / page \*

## Hash Join

- *partition overflow* – an E partition does not fit in memory during probing:  
apply hash join technique recursively:
  - divide E, S into subpartitions
  - join subpartitions pairwise
  - if subpartitions don't fit in memory, apply hash join technique recursively

## Hash Join

- memory requirements - objective: partition in E fits into main memory (S - similarly)
  - $B$  buffer pages; need one input buffer => maximum number of partitions:  $B-1$
  - size of largest partition:  $B - 2$  (need one input buffer for S, one output buffer)
  - assume uniformly sized partitions => size of each E partition:  $M/(B-1)$   
 $\Rightarrow M/(B-1) < B-2 \Rightarrow$  we need approximately  $B > \sqrt{M}$
- if an in-memory hash table is used to speed up tuple matching => need a little more memory (because the hash table for a collection of tuples will be a little larger than the collection itself)

\*  $E - M$  pages,  $p_E$  records / page \*

\* 1000 pages \* \* 100 records / page\*

## general join conditions

- equalities over several attributes
  - $E.SID = S.SID \text{ AND } E.attrE = S.attrS$ 
    - index nested loops join
      - Exams – inner relation:
        - build index on Exams with search key  $\langle SID, attrE \rangle$  (if not already created)
        - can also use index on SID or index on attrE
      - Students – inner relation (similar)
    - sort-merge join
      - sort Exams on  $\langle SID, attrE \rangle$ , sort Students on  $\langle SID, attrS \rangle$
    - hash join
      - partition Exams on  $\langle SID, attrE \rangle$ , partition Students on  $\langle SID, attrS \rangle$
    - other join algorithms
      - essentially unaffected

## general join conditions

- inequality comparison
  - $E.\text{attrE} < S.\text{attrS}$ 
    - index nested loops join
      - B+ tree index required
    - sort-merge join
      - not applicable
    - hash join
      - not applicable
    - other join algorithms
      - essentially unaffected

- \* no join algorithm is uniformly superior to others
- choice of a good algorithm depends on:
  - size(s) of:
    - joined relations
    - buffer pool
  - available access methods

## Selection

Q:

```
SELECT *
FROM Exams E
WHERE E.FacultyMember = 'Ionescu'
```

- use information in the selection condition to reduce the number of retrieved tuples
- e.g.,  $|Q| = 4$  (result set has 4 tuples), there's a B+ tree index on FacultyMember
  - it's expensive to scan E (1000 I/Os) to evaluate the query
  - should use the index instead
- selection algorithms based on the following techniques:
  - iteration, indexing

\* E - M pages,  $p_E$  records / page \*

\* 1000 pages \* \* 100 records / page\*

## Selection

- simple selections\*
  - $\sigma_{E.attr \ op \ val}(E)$
- no index on *attr*, data not sorted on *attr*
  - must scan E and test the condition for each tuple
  - access path: file scan

=> cost: M I/Os = 1000 I/Os
- no index, sorted data (E physically sorted on *attr*)
  - binary search to locate 1<sup>st</sup> tuple that satisfies condition and
  - scan E starting at this position until condition is no longer satisfied
  - access method: sorted file scan

\*Review *Relational Algebra* - lecture notes (*Databases* course)

## Selection

- simple selections
  - $\sigma_{E.attr \ op \ val}(E)$
- no index, sorted data (E physically sorted on *attr*)  
=> cost:
  - binary search:  $O(\log_2 M)$
  - scan cost: varies from 0 to M
  - binary search on E
    - $\log_2 1000 \approx 10$  I/Os

## Selection

- simple selections
  - $\sigma_{E.attr \ op \ val}(E)$
- B+ tree index on *attr*
  - \* search tree to find 1<sup>st</sup> index\* entry pointing to a qualifying E tuple
    - cost: typically 2, 3 I/Os
  - \* scan leaf pages to retrieve all qualifying entries
    - cost: depends on the number of qualifying entries
  - \* for each qualifying entry - retrieve corresponding tuple in E
    - cost: depends on the number of tuples and the nature of the index (clustered / non-clustered)

\*Review *Indexes* - lecture notes (*Databases* course)

## Selection

- simple selections
  - $\sigma_{E.attr \ op \ val}(E)$
- B+ tree index on *attr*
  - assumption
    - indexes use a2 or a3
    - a1-based index => data entry contains the data record => the cost of retrieving records = the cost of retrieving the data entries!
  - access path: B+ tree index
    - clustered index:
      - best access path when *op* is not *equality*
      - good access path when *op* is *equality*

## Selection

- simple selections:  $\sigma_{E.attr \ op \ val}(E)$
- B+ tree index on *attr*

Q

```
SELECT *
FROM Exams E
WHERE E.FacultyMember < 'C%
```

- names uniformly distributed with respect to 1<sup>st</sup> letter  
=>  $|Q| \approx 10,000$  tuples = 100 pages
- clustered B+ tree index on FacultyMember  
=> cost of retrieving tuples:  $\approx 100$  I/Os (a few I/Os to get from root to leaf)
- non-clustered B+ tree index on FacultyMember  
=> cost of retrieving tuples: up to 1 I/O per tuple (worst case) => up to 10,000 I/Os

\* E - M pages,  $p_E$  records / page \*      \* 1000 pages \* \* 100 records / page\*

## Selection

- simple selections:  $\sigma_{E.attr \ op \ val}(E)$

- B+ tree index on *attr*

```
SELECT *
```

```
FROM Exams E
```

```
WHERE E.FacultyMemger < 'C%'
```

- non-clustered B+ tree index on FacultyMember

- refinement - sort rids in qualifying data entries by page-id\*  
=> a page containing qualifying tuples is retrieved only once
    - cost of retrieving tuples: number of pages containing qualifying tuples (but such tuples are probably stored on more than 100 pages)

- range selections

- non-clustered indexes can be expensive
  - could be less costly to scan the relation (in our example: 1000 I/Os)

\*Review DB – Physical Structure - lecture notes (*Databases* course)

## Selection

- general selections
  - selections without disjunctions
- C - CNF condition without disjunctions
  - evaluation options:
    1. use the most selective access path
      - if it's an index I:
        - apply conjuncts in C that match I
        - apply rest of conjuncts to retrieved tuples
      - example
        - $c < 100 \text{ AND } a = 3 \text{ AND } b = 5$ 
          - can use a B+ tree index on  $c$  and check  $a = 3 \text{ AND } b = 5$  for each retrieved tuple
          - can use a hash index on  $a$  and  $b$  and check  $c < 100$  for each retrieved tuple

## Selection

- general selections - selections without disjunctions
  - evaluation options:
    2. use several indexes - when several conjuncts match indexes using a2 / a3
      - compute sets of rids of candidate tuples using indexes
      - intersect sets of rids, retrieve corresponding tuples
      - apply remaining conjuncts (if any)
      - example:  $c < 100 \text{ AND } a = 3 \text{ AND } b = 5$ 
        - use a B+ tree index on  $c$  to obtain rids of records that meet condition  $c < 100$  ( $R_1$ )
        - use a hash index on  $a$  to retrieve rids of records that meet condition  $a = 3$  ( $R_2$ )
        - compute  $R_1 \cap R_2 = R_{int}$
        - retrieve records with rids in  $R_{int}$  ( $R$ )
        - check  $b = 5$  for each record in  $R$

## Selection

- general selections
  - selections with disjunctions
- C - CNF condition with disjunctions, i.e., some conjunct  $J$  is a disjunction of terms
  - if some term  $T$  in  $J$  requires a file scan, testing  $J$  by itself requires a file scan
    - example:  $a < 100 \vee b = 5$ 
      - hash index on  $b$ , hash index on  $c$
  - => check both terms using a file scan (i.e., best access path: file scan)
- compare with the example below:
  - $(a < 100 \vee b = 5) \wedge c = 7$
  - hash index on  $b$ , hash index on  $c$
- => use index on  $c$ , apply  $a < 100 \vee b = 5$  to each retrieved tuple (i.e., most selective access path: index)

## Selection

- general selections
  - selections with disjunctions
- C - CNF condition with disjunctions
  - every term  $T$  in a disjunction matches an index  
=> retrieve tuples using indexes, compute union
  - example
    - $a < 100 \vee b = 5$
    - B+ tree indexes on  $a$  and  $b$
    - use index on  $a$  to retrieve records that meet condition  $a < 100$  ( $R_1$ )
    - use index on  $b$  to retrieve records that meet condition  $b = 5$  ( $R_2$ )
    - compute  $R_1 \cup R_2 = R$
    - if all matching indexes use a2 or a3 => take union of rids, retrieve corresponding tuples

## Projection

- $\Pi_{\text{SID}, \text{CID}}(\text{Exams})$

```
SELECT DISTINCT E.SID, E.CID  
FROM Exams E
```

- to implement projection:
  - eliminate:
    - unwanted columns
    - duplicates
  - projection algorithms - *partitioning* technique:
    - sorting
    - hashing

## Projection Based on Sorting

- step 1
  - scan  $E \Rightarrow$  set of tuples containing only desired attributes ( $E'$ )
  - cost:
    - scan  $E$ :  $M$  I/Os
    - write temporary relation  $E'$ :  $T$  I/Os
      - $T$  depends on: number of columns and their sizes,  $T$  is  $O(M)$
- step 2
  - sort tuples in  $E'$
  - sort key: all columns
  - cost:  $O(T \log T)$  (also  $O(M \log M)$ )
- step 3
  - scan sorted  $E'$ , compare adjacent tuples, eliminate duplicates
  - cost:  $T$
- total cost:  $O(M \log M)$

## Projection Based on Sorting

\* example

```
SELECT DISTINCT E.SID, E.CID  
FROM Exams E
```

- scan Exams: 1000 I/Os
- size of tuple in E': 10 bytes

=> cost of writing temporary relation E': 250 I/Os

- available buffer pages: 20
  - E' can be sorted in 2 passes
  - sorting cost:  $2 * 2 * 250 = 1000$  I/Os
- final scan of E' - cost: 250 I/Os

=> total cost:  $1000 + 250 + 1000 + 250 = 2500$  I/Os

\* E – record size = 40 bytes \*

\* 1000 pages \*

\* 100 records / page\*

## Projection Based on Sorting

\* example

```
SELECT DISTINCT E.SID, E.CID  
FROM Exams E
```

- scan Exams: 1000 I/Os
- size of tuple in E': 10 bytes

=> cost of writing temporary relation E': 250 I/Os

- available buffer pages: 257
  - E' can be sorted in 1 pass
  - sorting cost:  $2 * 1 * 250 = 500$  I/Os
- final scan of E' - cost: 250 I/Os

=> total cost:  $1000 + 250 + 500 + 250 = 2000$  I/Os

\* E – record size = 40 bytes \*

\* 1000 pages \*

\* 100 records / page\*

## Projection Based on Sorting

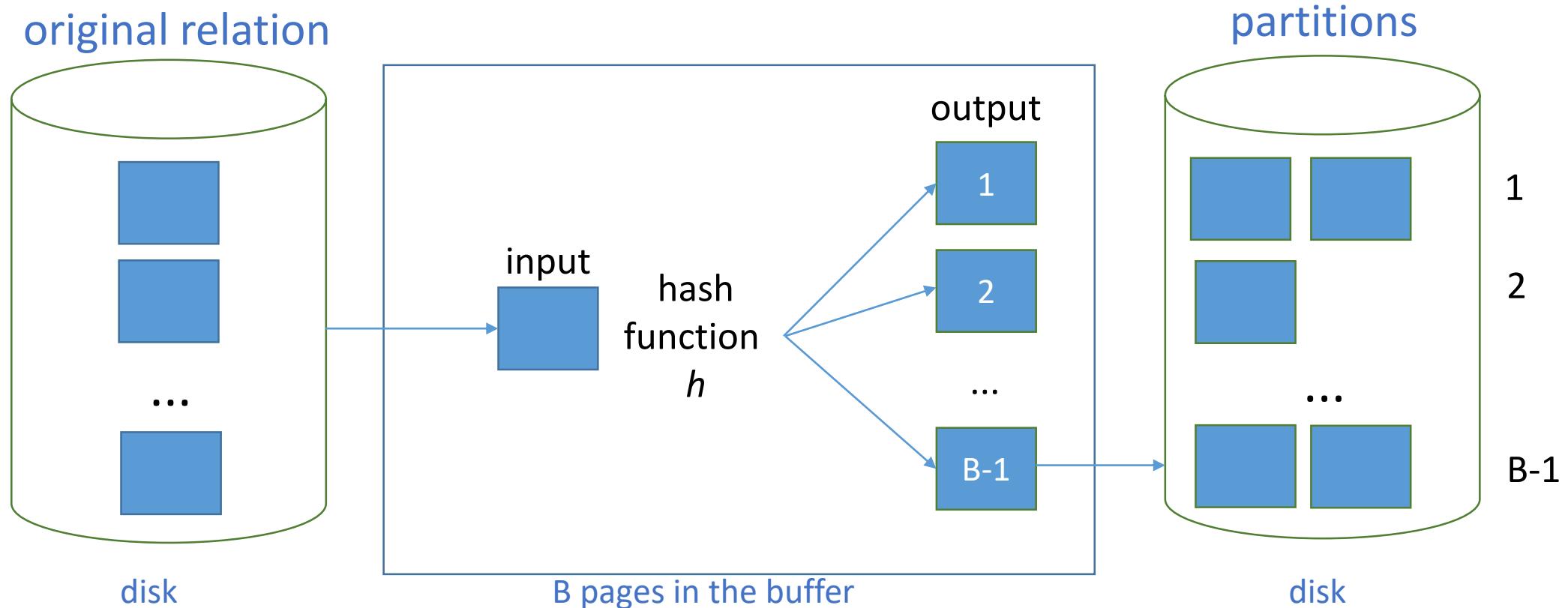
- improvement
  - adapt the sorting algorithm to do projection with duplicate elimination
    - modify pass 0 of external sort: eliminate unwanted columns
      - read in  $B$  pages from  $E$
      - write out  $(T/M) * B$  internally sorted pages of  $E'$ 
        - refinement: write out  $2*B$  internally sorted pages of  $E'$  (on average)
        - tuples in runs - smaller than input tuples
      - modify merging passes: eliminate duplicates
        - number of result tuples is smaller than number of input tuples

## Projection Based on Sorting

- improvement
  - \* example
    - pass 0:
      - scan Exams: 1000 I/Os
      - write out 250 pages:
        - 20 available buffer pages
        - 250 pages => 7 sorted runs about 40 pages long (except the last one, which is about 10 pages long)
    - pass 1:
      - read in all runs – cost: 250 I/Os
      - merge runs
    - total cost :  $1000 + 250 + 250 = 1500$  I/Os

## Projection Based on Hashing

- phases: partitioning & duplicate elimination
- partitioning phase:
  - 1 input buffer page – read in the relation one page at a time
  - hash function  $h$  – distribute tuples uniformly to one of  $B-1$  partitions
  - $B-1$  output buffer pages – one output page / partition



## Projection Based on Hashing

- partitioning phase:
  - read the relation using the input buffer page
  - for each tuple  $t$ :
    - discard unwanted fields => tuple  $t'$
    - apply hash function  $h$  to  $t'$
    - write  $t'$  to the output buffer page that it is hashed to by  $h$
- =>  $B-1$  partitions
- partition:
  - collection of tuples with:
    - common hash value
    - no unwanted fields
  - 2 tuples in different partitions are guaranteed to be distinct

## Projection Based on Hashing

- duplicate elimination phase:
  - process all partitions:
    - read in partition P, one page at a time
    - build in-memory hash table with hash function  $h_2$  ( $\neq h$ ) on all fields:
      - if a new tuple hashes to the same value as an existing tuple, compare them to check if they are distinct
      - eliminate duplicates
    - write duplicate-free hash table to result file
    - clear in-memory hash table
- partition overflow
  - apply hash-based projection technique recursively (subpartitions)

## Projection Based on Hashing

- cost
  - partitioning:
    - read E: M I/Os
    - write E': T I/Os
  - duplicate elimination:
    - read in partitions: T I/Os
- => total cost:  $M + 2*T$  I/Os
- Exams:
  - $1000 + 2*250 = 1500$  I/Os

## Set Operations

- intersection, cross-product
  - special cases of join (i.e., join condition for intersection - equality on all fields, no join condition for cross-product)
- union, set-difference
  - similar
- union:  $R \cup S$ 
  - sorting
    - sort R and S on all attributes
    - scan the sorted relations in parallel; merge them, eliminating duplicates
  - refinement
    - produce sorted runs of R and S, merge runs in parallel

## Set Operations

- union:  $R \cup S$ 
  - hashing
    - partition  $R$  and  $S$  with the same hash function  $h$
    - for each  $S$ -partition
      - build in-memory hash table (using  $h2$ ) for the  $S$ -partition
      - scan corresponding  $R$ -partition, add tuples to hash table, discard duplicates
      - write out hash table
      - clear hash table

## Aggregate Operations

- without grouping
  - scan relation
  - maintain *running information* about scanned tuples
    - COUNT - count of values retrieved
    - SUM - *total* of values retrieved
    - AVG - <*total, count*> of values retrieved
    - MIN, MAX - smallest / largest value retrieved
- with grouping
  - sort relation on the grouping attributes
  - scan relation to compute aggregate operations for each group
  - improvement: combine sorting with aggregation computation
  - alternative approach based on hashing

## Aggregate Operations

- using existing indexes
  - index with a search key that includes all the attributes required by the query
    - index-only scan
  - attribute list in the GROUP BY clause is a prefix of the index search key (tree index)
    - get data entries (and records, if necessary) in the required order
    - i.e., avoid sorting

## References

- [Ra00] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (2<sup>nd</sup> Edition), McGraw-Hill, 2000
- [Da03] DATE, C.J., An Introduction to Database Systems (8<sup>th</sup> Edition), Addison-Wesley, 2003
- [Ga08] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book, Prentice Hall Press, 2008
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007,  
<http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Si10] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, McGraw-Hill, 2010, <http://codex.cs.yale.edu/avi/db-book/>
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems,  
<http://infolab.stanford.edu/~ullman/fcdb.html>

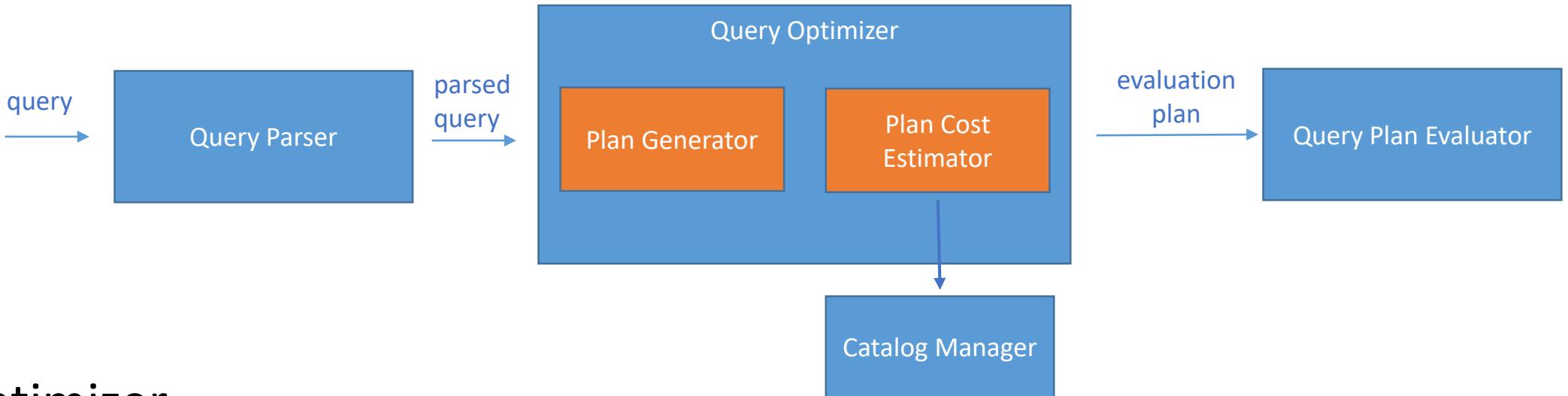
# Database Management Systems

Lecture 9

Evaluating Relational Operators

Query Optimization (IV)

# Query Optimization



- optimizer
  - objective
    - given a query  $Q$ , find a good evaluation plan for  $Q$
    - generates alternative plans for  $Q$ , estimates their costs, and chooses the one with the least estimated cost
    - uses information from the system catalogs

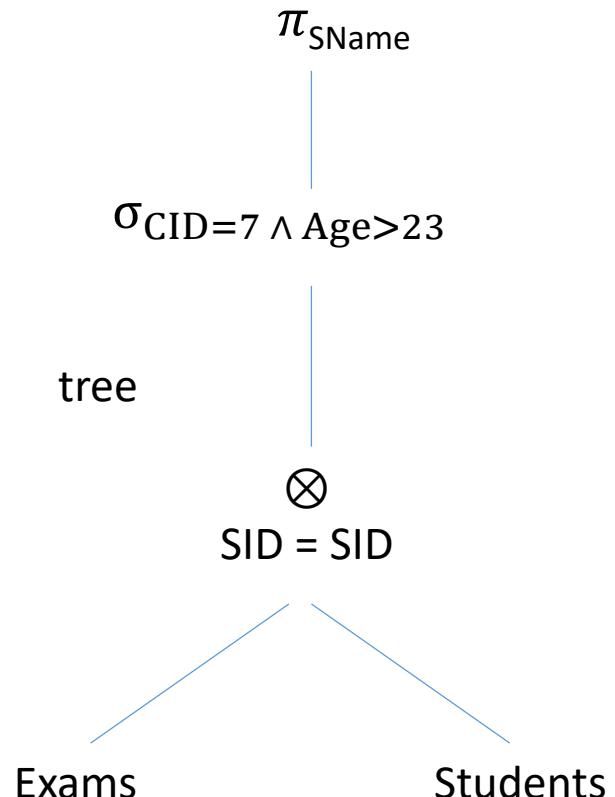
- running example - schema
  - Students (SID: integer, SName: string, Age: integer)
  - Courses (CID: integer, CName: string, Description: string)
  - Exams (SID: integer, CID: integer, EDate: date, Grade: integer)
- Students
  - every record has 50 bytes
  - there are 80 records / page
  - 500 pages
- Courses
  - every record has 40 bytes
  - there are 100 records / page
  - 1 page
- Exams
  - every record has 40 bytes
  - there are 100 records / page
  - 1000 pages

# Query Evaluation Plans

```
SELECT S.SName  
      query  
FROM Exams E, Students S  
WHERE E.SID = S.SID AND E.CID = 7  
      AND S.Age > 23
```

$$\pi_{SName}(\sigma_{CID=7 \wedge Age>23}(Exams \otimes_{SID=SID} Students))$$

relational algebra expression



# Query Evaluation Plans

```
SELECT S.SName  
FROM Exams E, Students S  
WHERE E.SID = S.SID AND E.CID = 7  
      AND S.Age > 23
```

$$\pi_{SName}(\sigma_{CID=7 \wedge Age>23}(Exams \otimes_{SID=SID} Students))$$

- query evaluation plan
  - extended relational algebra tree
  - node – annotations
    - relation
      - access method
    - relational operator
      - implementation method

$$\pi_{SName} \text{ (on-the-fly)}$$
$$\sigma_{CID=7 \wedge Age>23} \text{ (on-the-fly)}$$

fully specified evaluation plan

$$\otimes$$
  
$$SID = SID \text{ (Page-Oriented Nested Loops)}$$

(file scan) Exams

Students (file scan)

# Query Evaluation Plans

```
SELECT S.SName  
FROM Exams E, Students S  
WHERE E.SID = S.SID AND E.CID = 7  
      AND S.Age > 23
```

$$\pi_{SName}(\sigma_{CID=7 \wedge Age>23}(Exams \otimes_{SID=SID} Students))$$

- page-oriented Simpled Nested Loops Join
  - Exams – outer relation
  - selection, projection applied on-the-fly to each tuple in the join result, i.e., the result of the join (before applying selection and projection) is not stored

$$\pi_{SName} \text{ (on-the-fly)}$$
$$\sigma_{CID=7 \wedge Age>23} \text{ (on-the-fly)}$$

fully specified evaluation plan

$$\otimes$$
  
$$SID = SID \text{ (Page-Oriented Nested Loops)}$$

(file scan) Exams

Students (file scan)

## Pipelined Evaluation

```
SELECT *
FROM Exams
WHERE EDate > '1-1-2017' AND Grade > 8
      T1           T2
```

$$\sigma_{Grade>8}(\sigma_{EDate>'1-1-2017'}(Exams))$$

- index / matches  $T1$
- v1 - *materialization*
  - evaluate  $T1$
  - write out result tuples to temporary relation  $R$ , i.e., tuples are *materialized*
  - apply the 2<sup>nd</sup> selection to  $R$
  - cost: read and write  $R$

## Pipelined Evaluation

```
SELECT *
FROM Exams
WHERE EDate > '1-1-2017' AND Grade > 8
      T1           T2
```

- v2 – *pipelined evaluation*
  - apply the 2<sup>nd</sup> selection to each tuple in the result of the 1<sup>st</sup> selection as it is produced
  - i.e., 2<sup>nd</sup> selection operator is applied *on-the-fly*
  - saves the cost of writing out / reading in the temporary relation  $R$

## Query Blocks – Units of Optimization

- optimize SQL query Q
  - parse Q => collection of query *blocks*
  - optimizer:
    - optimize one block at a time
- query *block* - SQL query:
  - without nesting
  - with exactly: one SELECT clause, one FROM clause
  - with at most: one WHERE clause, one GROUP BY clause, one HAVING clause
    - WHERE condition - CNF

# Query Blocks – Units of Optimization

- query Q:

```
SELECT S.SID, MIN(E.EDate)
FROM Students S, Exams E, Courses C
WHERE S.SID = E.SID AND E.CID = C.CID AND C.Description = 'Elective' AND
      S.Age = (SELECT MAX(S2.Age)
                FROM Students S2)
GROUP BY S.SID
HAVING COUNT(*) > 2
```

nested block

- decompose query into a collection of blocks without nesting

```
SELECT S.SID, MIN(E.EDate)
FROM Students S, Exams E, Courses C
WHERE S.SID = E.SID AND E.CID = C.CID AND C.Description = 'Elective' AND
      S.Age = Reference to nested block
GROUP BY S.SID
HAVING COUNT(*) > 2
```

## Query Blocks – Units of Optimization

### \* block optimization

- express query block as a relational algebra expression

```
SELECT S.SID, MIN(E.EDate)
FROM Students S, Exams E, Courses C
WHERE S.SID = E.SID AND E.CID = C.CID AND C.Description = 'Elective' AND
      S.Age = Reference to nested block
GROUP BY S.SID
HAVING COUNT(*) > 2
```

$\pi_{S.SID, \text{MIN}(E.EDate)}$ (

$\text{HAVING}_{\text{COUNT}(* > 2)}$ (

$\text{GROUP BY}_{S.SID}$ (

$\sigma_{S.SID = E.SID \wedge E.CID = C.CID \wedge C.Description = 'Elective' \wedge S.Age = \text{value\_from\_nested\_block}}$ (

*Students × Exams × Courses ))))*

- GROUP BY, HAVING – operators in the extended algebra used for plans
- argument list of projection can include aggregate operations

## Query Blocks – Units of Optimization

- query Q treated as a  $\sigma \pi \times$  algebra expression
- the remaining operations in Q are performed on the result of the  $\sigma \pi \times$  expression

```
SELECT S.SID, MIN(E.EDate)
FROM Students S, Exams E, Courses C
WHERE S.SID = E.SID AND E.CID = C.CID AND C.Description = 'Elective' AND
      S.Age = Reference to nested block
GROUP BY S.SID
HAVING COUNT(*) > 2
```

$$\begin{aligned} &\pi_{S.SID, E.EDate}( \\ &\sigma_{S.SID = E.SID \wedge E.CID = C.CID \wedge C.Description = 'Elective' \wedge S.Age = value\_from\_nested\_block}( \\ &\quad Students \times Exams \times Courses) ) \end{aligned}$$

- attributes in GROUP BY, HAVING are added to the argument list of projection
- aggregate expressions in the argument list of projection are replaced with their argument attributes

## Query Blocks – Units of Optimization

### \* block optimization

- find best plan P for the  $\sigma \pi \times$  expression
- evaluate P => result set RS
- sort/hash RS => groups
- apply HAVING to eliminate some groups
- compute aggregate expressions in SELECT for each remaining group

$$\begin{aligned} & \pi_{S.SID, \text{MIN}(E.EDate)}( \\ & \quad \text{HAVING } \text{COUNT(*)} > 2 \\ & \quad \text{GROUP BY } S.SID \\ & \pi_{S.SID, E.EDate}( \\ & \quad \sigma_{S.SID = E.SID \wedge E.CID = C.CID \wedge C.Description = 'Elective' \wedge S.Age = \text{value\_from\_nested\_block}}( \\ & \quad \quad \textit{Students} \times \textit{Exams} \times \textit{Courses} )))) \end{aligned}$$

## IBM's System R Optimizer

- tremendous influence on subsequent relational optimizers
- design choices:
  - use statistics to estimate the costs of query evaluation plans
  - consider only plans with binary joins in which the inner relation is a base relation
  - focus optimization on SQL queries without nesting
  - don't eliminate duplicates when performing projections (unless DISTINCT is used)

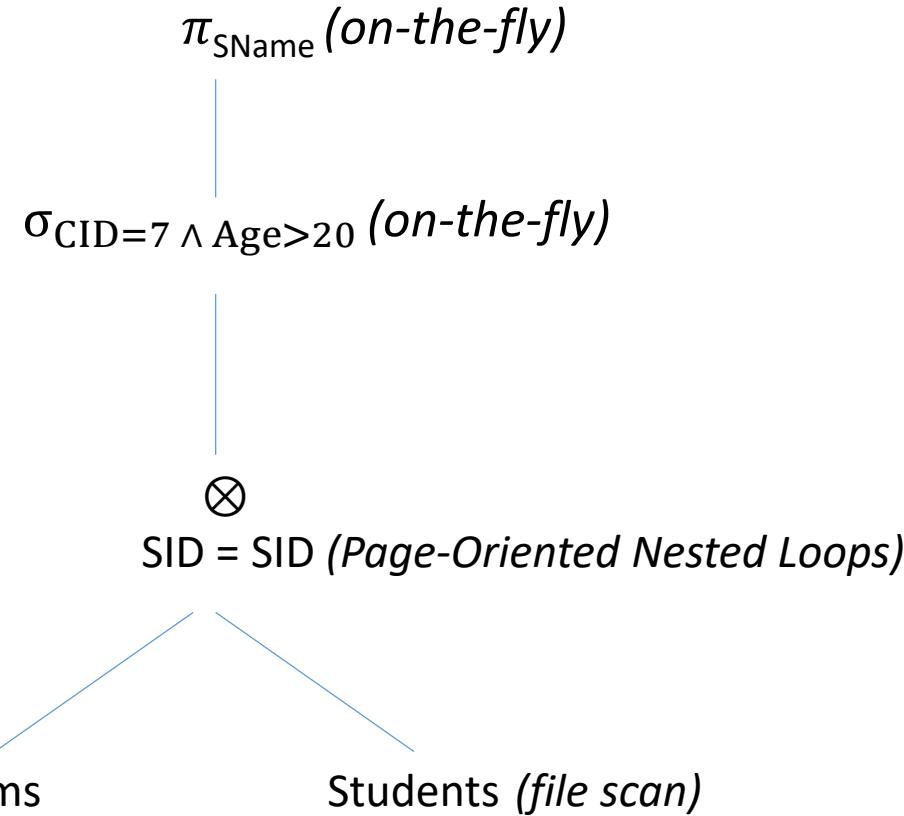
## Motivating Example

\* E - 1000 pages \*

\* S - 500 pages \*

```
SELECT S.SName  
FROM Exams E, Students S  
WHERE E.SID = S.SID AND E.CID = 7  
      AND S.Age > 20
```

- $\sigma, \pi$  – on-the-fly
- cost of plan – very high:
  - $1000 + 1000 * 500 = 501,000$  I/Os
- less efficient plan
  - join: cross-product followed by a selection



# Motivating Example

## \* optimizations

- reduce sizes of the relations to be joined

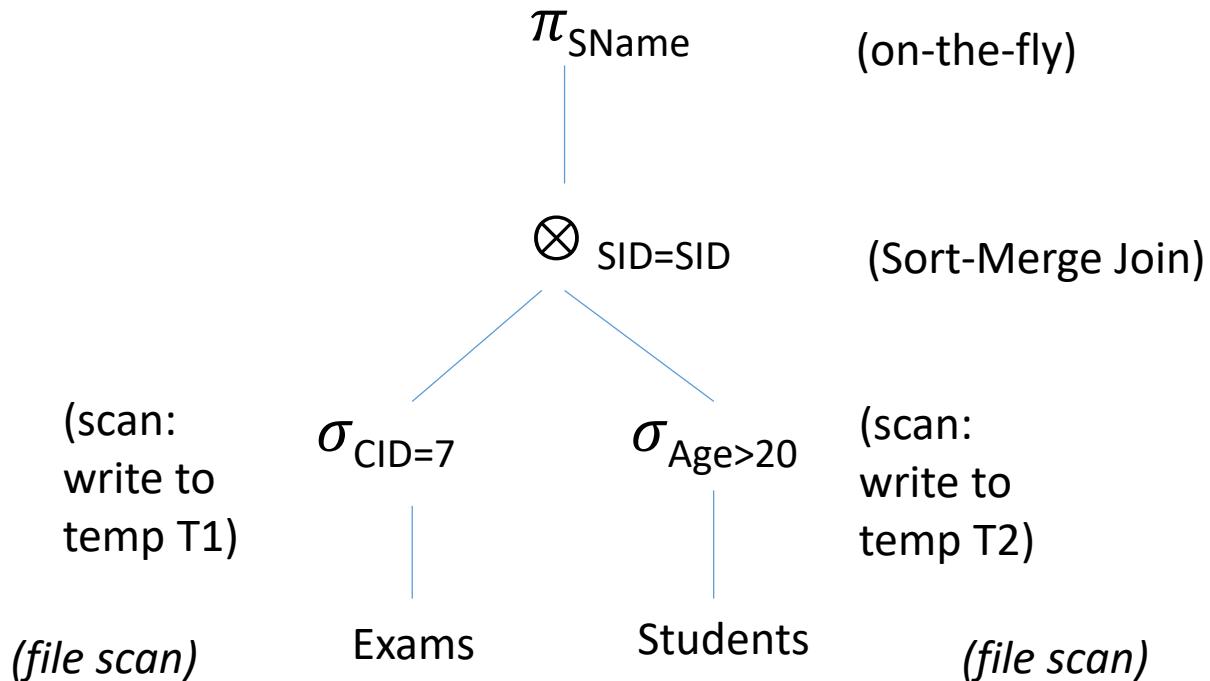
- push selections, projections ahead of the join

- alternative plans

- push selections ahead of joins

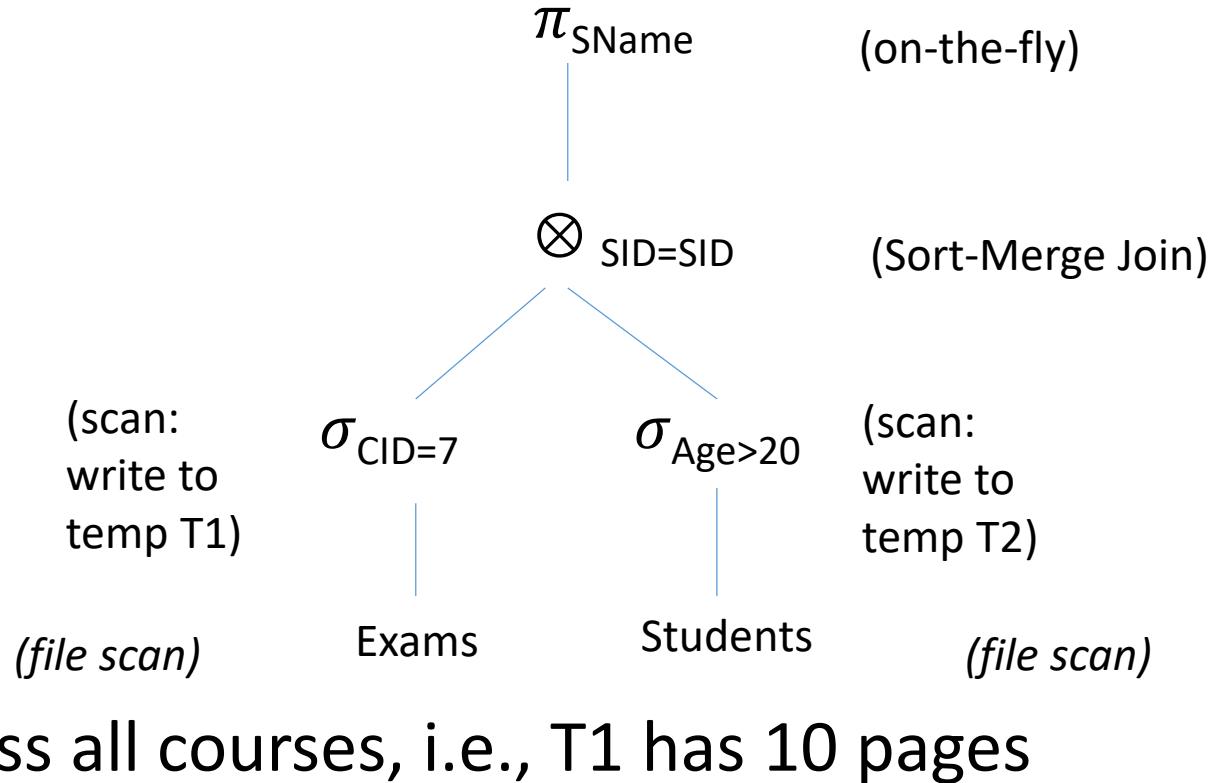
- selection

- file scan
  - write the result to a temporary relation on disk
  - join the temporary relations using Sort-Merge Join



## Motivating Example

- 5 available buffer pages
- cost
  - $\sigma_{CID=7}$ 
    - scan Exams: 1000 I/Os
    - write T1
      - assume exams are uniformly distributed across all courses, i.e., T1 has 10 pages
  - $\sigma_{Age>20}$ 
    - scan Students: 500 I/Os
    - write T2
      - assume ages are uniformly distributed over the range 19 to 22, i.e., T2 has 250 pages



## Motivating Example

- 5 available buffer pages

- cost

- Sort-Merge Join

- T1 - 10 pages

- sort T1:  $2 * 2 * 10 = 40$  I/Os

- T2 - 250 pages

- sort T2:  $2 * 4 * 250 = 2000$  I/Os

- merge sorted T1 and T2

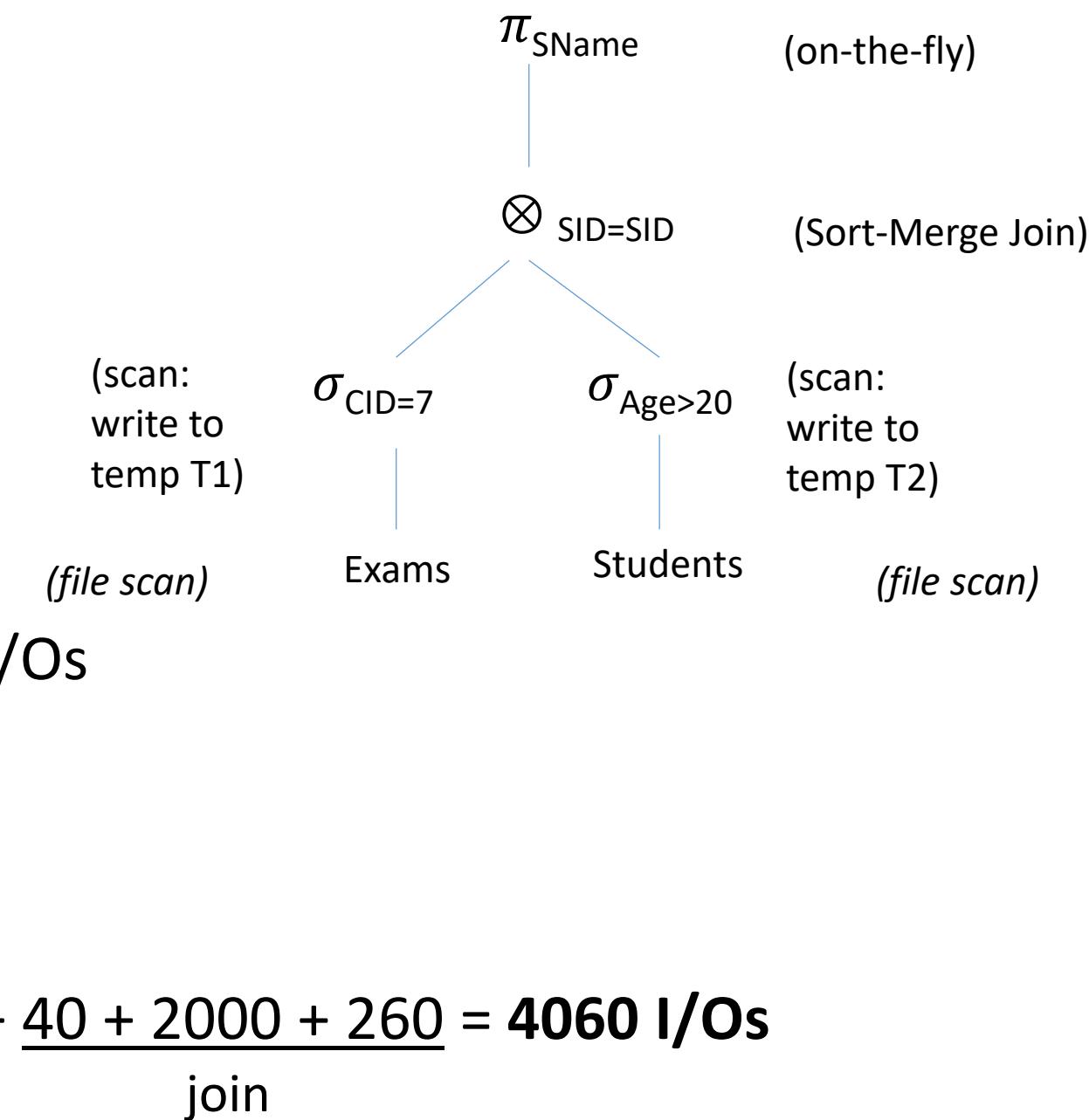
- $10 + 250 = 260$  I/Os

- $\pi$  - on the fly

=> **total cost:**  $1000 + 10 + 500 + 250 + 40 + 2000 + 260 = 4060$  I/Os

selection

join



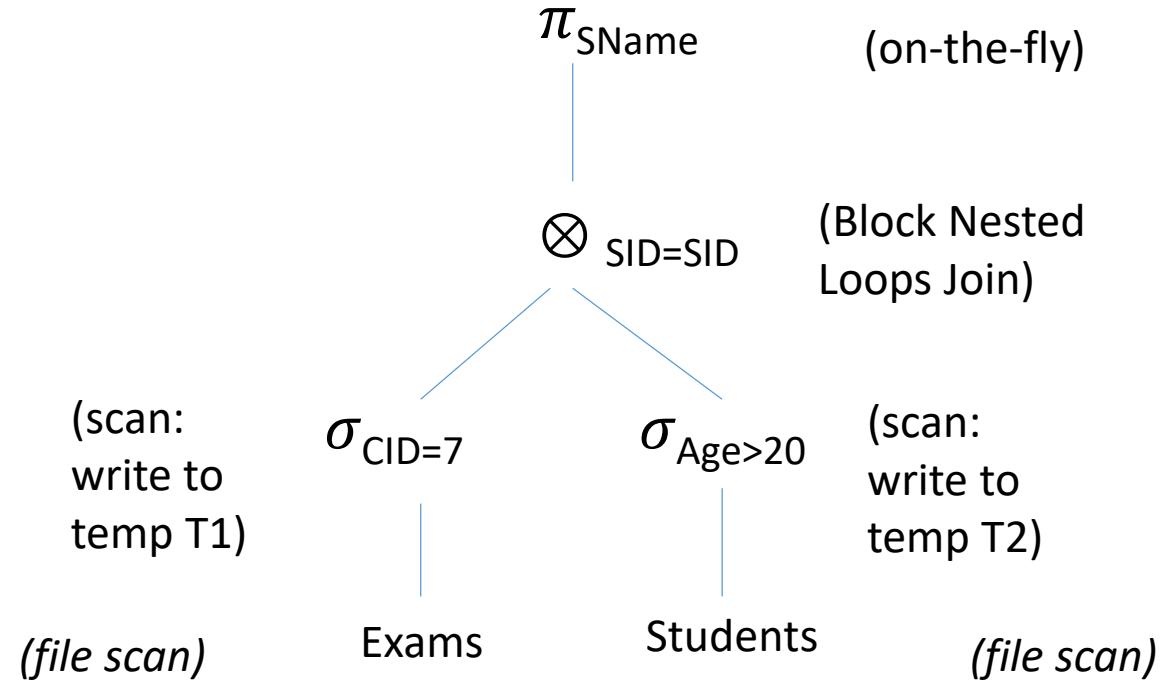
## Motivating Example

- 5 available buffer pages
- cost
  - Block Nested Loops Join
    - T1 - 10 pages, T2 - 250 pages
    - T1 - outer relation
      - => scan T1: 10 I/Os
      - $[10/3] = 4$  T1 blocks
      - => T2 scanned 4 times:  $4 * 250 = 1000$  I/Os
      - BNLJ cost:  $10 + 1000 = 1010$  I/Os
    - $\pi$  - on the fly

=> total cost: 1000 + 10 + 500 + 250 + 10 + 1000 = **2770 I/Os**

selection

join



## Motivating Example

- push projections ahead of joins
  - drop unwanted columns while scanning Exams and Students to evaluate selections => T1[SID], T2[SID, SName]
- T1 fits within 3 buffer pages  
=> T2 scanned only once  
=> **total cost:** about 2000 I/Os

## Motivating Example

\* optimizations

- investigate the use of indexes

- clustered static hash index on Exams(CID)

- hash index on Students(SID)

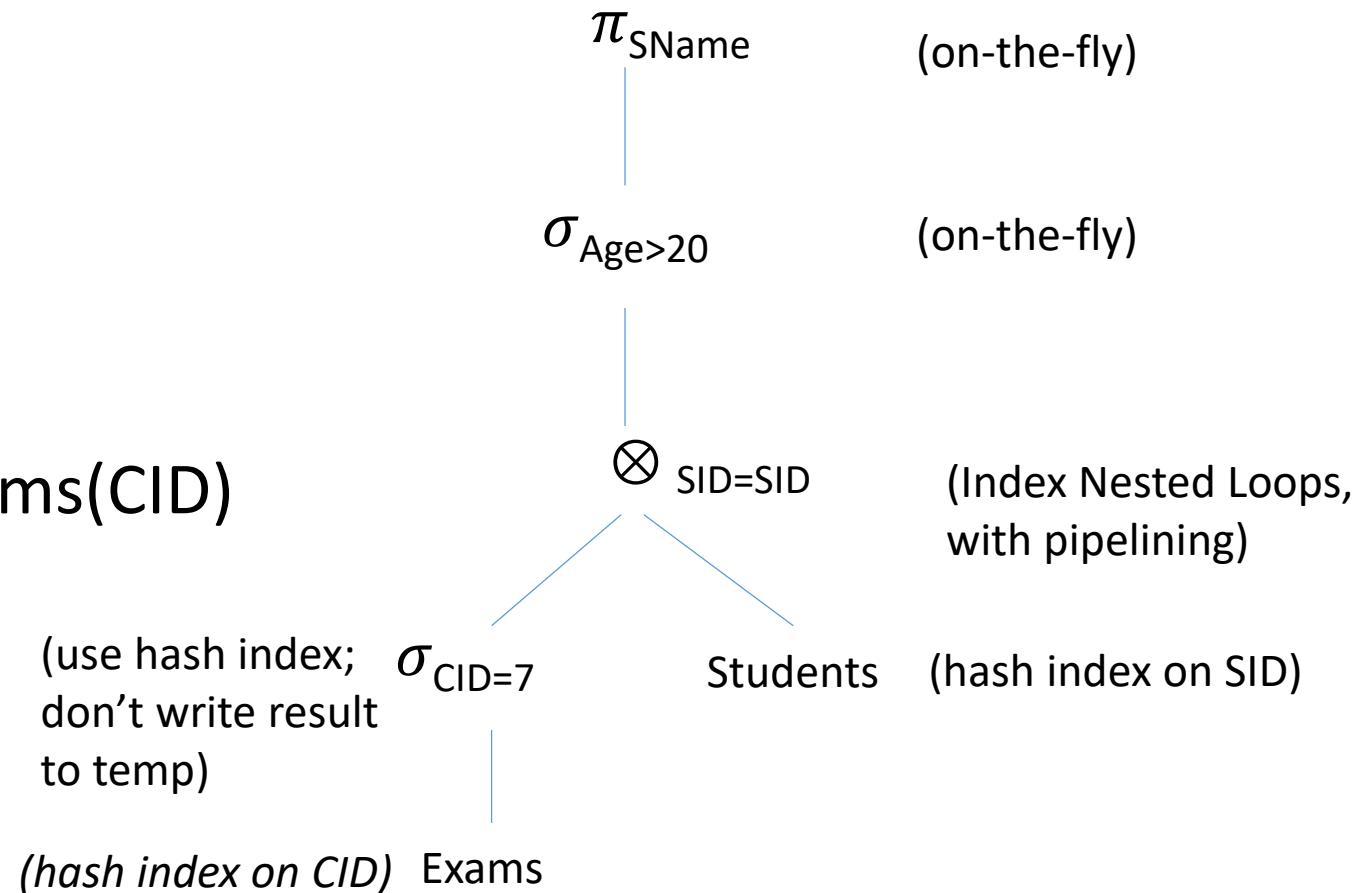
- cost

- $\sigma_{CID=7}$

- assume exams are uniformly distributed across all courses => 100,000 exams / 100 courses => 1,000 exams / course

- clustered index on CID => 1,000 tuples for course with CID=7 appear consecutively within the same bucket => cost: 10 I/Os

- the result of the selection is not materialized, the join is pipelined



## Motivating Example

- cost
    - Index Nested Loops
      - find matching Students tuple for each selected exam
      - use hash index on SID
        - assume the index uses a1 => cost of 1.2 I/Os (on avg.) per exam
    - $\sigma, \pi$  – performed on-the-fly on each tuple in the result of the join
- => total cost = 10 + 1000 \* 1.2 = **1210** I/Os
- $\pi_{SName}$  (on-the-fly)  
 $\sigma_{Age > 20}$  (on-the-fly)  
 $\otimes_{SID=SID}$  (Index Nested Loops, with pipelining)  
 Students (hash index on SID)  
 Exams  
 (use hash index; don't write result to temp)  
 (hash index on CID)
- 

\* can we push the selection  $Age > 20$  ahead of the join?

## Estimating the Cost of a Plan

\* estimating the cost of an evaluation plan for a query block

- for each node N in the tree:
  - estimate the cost of the corresponding operation (pipelining versus temporary relations)
  - estimate the size of N's result and whether it is sorted
    - N's result is the input of N's parent node
    - these estimates affect the estimation of cost, size, and sort order for N's parent

## Estimating the Cost of a Plan

- estimating costs
    - use data about the input relations (such statistics are stored in the DBMS's system catalogs)
    - number of pages, existing indexes, etc.
  - obtained estimates are at best approximations to actual sizes and costs
- => one shouldn't expect the optimizer to find the best possible plan
- optimizer - goals:
    - avoid the worst plans
    - find a good plan

## Statistics Maintained by the DBMS

- updated periodically, not every time the data is changed
  - relation R
    - cardinality -  $NTuples(R)$ 
      - the number of tuples in R
    - size -  $NPages(R)$ 
      - the number of pages in R
  - index I
    - cardinality -  $NKeys(I)$ 
      - the number of distinct key values for I
    - size -  $INPages(I)$ 
      - the number of pages for I
      - B+ tree index
        - number of leaf pages
    - height -  $IHeight(I)$ 
      - maintained for tree indexes
      - the number of nonleaf levels in I
    - range -  $ILow(I), IHight(I)$ 
      - the minimum / maximum key value in I

## Estimating Result Sizes

- query Q
  - SELECT attribute list
  - FROM relation list
  - WHERE  $\text{term}_1 \wedge \dots \wedge \text{term}_k$
- the maximum number of tuples in Q's result:
  - $\prod |R_i|, R_i \in \text{relation list}$
  - every  $\text{term}_j$  in the WHERE clause eliminates some candidate tuples
    - associate a reduction factor  $RF_j$  with each term  $\text{term}_j$
    - $RF_j$  models the impact  $\text{term}_j$  has on the result size
  - estimate the actual size of the result:  $\prod |R_i| * \prod RF_j$ 
    - i.e., the maximum result size times the product of the reduction factors for the terms in the WHERE clause
  - assumption: the conditions tested by the terms in the WHERE clause are statistically independent

## Estimating Result Sizes

- compute reduction factors for terms in the WHERE clause
- assumptions:
  - uniform distribution of values
  - independent distribution of values in different columns

SELECT attribute list

FROM relation list

WHERE term<sub>1</sub> AND ... AND term<sub>k</sub>

- *column = value*
  - index I on *column* => RF approximated by  $1/\text{NKeys}(I)$
  - no index on *column* => RF: 1/10
    - maintain statistics on *column* (e.g., number of distinct values in *column*) to obtain a better value
- *column1 = column2*
  - indexes *I1* on *column1*, *I2* on *column2*  
=> RF:  $1/\text{MAX}(\text{NKeys}(I1), \text{NKeys}(I2))$

## Estimating Result Sizes

- only one index I (on one of the 2 columns) => RF:  $1/N\text{Keys}(I)$
- no indexes => RF:  $1/10$
- $column > value$ 
  - index I on  $column$  => RF:  $(I\text{High}(I) - value) / (I\text{High}(I) - I\text{Low}(I))$
  - no index on  $column$  or  $column$  not of an arithmetic type  
=> a value less than 0.5 is arbitrarily chosen
  - similar formulas can be obtained for other range selections
- $column \text{ IN } (list \text{ of } values)$   
=> RF: (RF for  $column = value$ ) \* number of items in list (but at most 0.5)
- $\text{NOT condition}$   
=> RF:  $1 - \text{RF for condition}$
- obtain better estimates
  - use more detailed statistics (e.g., histograms of the values in a column)

## Relational Algebra Equivalences

- central role in generating alternative plans
- different join orders can be considered
- selections, projections can be pushed ahead of joins
- cross-products can be converted to joins
- selections
  - *cascading selections*
    - $\sigma_{c1 \wedge \dots \wedge cn}(R) \equiv \sigma_{c1}(\dots(\sigma_{cn}(R)))$
    - *commutativity*
      - $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$
  - projections
    - *cascading projections*
      - $\pi_{a1}(R) \equiv \pi_{a1}(\dots(\pi_{an}(R)))$
      - $a_i$  – set of attributes in R
      - $a_i \subseteq a_{i+1}$ , for  $i = 1..n-1$

## Relational Algebra Equivalences

- joins and cross-products
  - assumption
    - fields are identified by their name, not by their position
  - *associativity*
    - $R \times (S \times T) \equiv (R \times S) \times T$
    - $R * (S * T) \equiv (R * S) * T$
  - *commutativity*
    - $R \times S \equiv S \times R$
    - $R * S \equiv S * R$
    - can choose the inner / outer relation in a join
- e.g., check that  $R * (S * T) \equiv (T * R) * S$ 
  - commutativity
    - $R * (S * T) \equiv R * (T * S)$
  - associativity
    - $R * (T * S) \equiv (R * T) * S$
  - commutativity
    - $(R * T) * S \equiv (T * R) * S$

## Relational Algebra Equivalences

- can commute  $\sigma$  with  $\pi$  if  $\sigma$  uses only attributes retained by  $\pi$ 
  - $\pi_a(\sigma_c(R)) \equiv \sigma_c(\pi_a(R))$
- can combine  $\sigma$  with  $\times$  to form a join
  - $R \otimes_c S \equiv \sigma_c(R \times S)$
- can commute  $\sigma$  with  $\times$  or a join when the selection condition includes only fields of one of the arguments (to the cross-product or join)
  - for instance:
    - $\sigma_c(R * S) \equiv \sigma_c(R) * S$
    - $\sigma_c(R \times S) \equiv \sigma_c(R) \times S$ 
      - condition c must include only fields from R
- in general:  $\sigma_c(R \times S) \equiv \sigma_{c1}(\sigma_{c2}(R) \times \sigma_{c3}(S))$ 
  - c1 – attributes of both R and S
  - c2 – only attributes of R
  - c3 – only attributes of S

## Relational Algebra Equivalences

- can commute  $\pi$  with  $\times$ 
  - $\pi_a(R \times S) \equiv \pi_{a1}(R) \times \pi_{a2}(S)$
  - $a1$  – attributes in  $a$  that appear in  $R$
  - $a2$  – attributes in  $a$  that appear in  $S$
- can commute  $\pi$  with join
  - $\pi_a(R \otimes_c S) \equiv \pi_{a1}(R) \otimes_c \pi_{a2}(S)$ 
    - every attribute in  $c$  must appear in  $a$
    - $a1$  – attributes in  $a$  that appear in  $R$
    - $a2$  – attributes in  $a$  that appear in  $S$
  - $a$  doesn't contain all the attributes in  $c$  – generalization
    - eliminate unwanted fields, compute join, eliminate fields not in  $a$ 
      - $\pi_a(R \otimes_c S) \equiv \pi_a(\pi_{a1}(R) \otimes_c \pi_{a2}(S))$
      - $a1$  – attributes of  $R$  that appear in either  $a$  or  $c$
      - $a2$  – attributes of  $S$  that appear in either  $a$  or  $c$

## Enumeration of Alternative Plans

- query Q
  - consider a certain set of plans
  - choose the plan with the least estimated cost
    - algebraic equivalences
    - implementations techniques for Q's operators
- not all algebraically equivalent plans are enumerated (optimization costs would be too high)
- two main cases:
  - queries with one relation in the FROM clause
  - queries with two or more relations in the FROM clause

## Enumeration of Alternative Plans

- queries with one relation in the FROM clause (i.e., no joins; only  $\sigma$ ,  $\pi$ , grouping, aggregate operations):
  - if there is only one  $\sigma$  or  $\pi$  or aggregate operation: consider implementation techniques and cost estimates discussed in previous lectures
  - if there is a combination of several  $\sigma$ ,  $\pi$ , aggregate operations:
    - plans with / without indexes
    - example query:

```
SELECT S.RoundedGPA, COUNT(*)
FROM Students S
WHERE S.RoundedGPA > 5 AND S.Age = 20
GROUP BY S.RoundedGPA
HAVING COUNT(DISTINCT S.SName) > 5
```

$$\begin{aligned} & \pi_{S.RoundedGPA, COUNT(*)}( \\ & \text{HAVING } COUNT(DISTINCT S.SName) > 5( \\ & \text{GROUP BY } S.RoundedGPA( \\ & \pi_{S.RoundedGPA, S.SName}( \\ & \sigma_{S.RoundedGPA > 5 \wedge S.Age = 20}( \\ & \quad \text{Students})))) \end{aligned}$$

## Enumeration of Alternative Plans

\* plans without indexes:

- apply  $\sigma, \pi$  while scanning Students
  - file scan: NPages(Students): 500 I/Os
  - write out tuples to a temporary relation T:
    - $NPages(Students) * RF(RoundedGPA > 5) * RF(Age = 20) * (\text{size of a pair } <\text{RoundedGPA}, \text{SName}> / \text{size of a Students tuple})$
    - RF for *RoundedGPA* > 5: 0.5
    - RF for *Age* = 20: 0.1
    - size of <*RoundedGPA*, *SName*>: about 0.8 \* size of a Students tuple
  - =>  $500 * 0.5 * 0.1 * 0.8 = 20$  I/Os (temporary relation T)
- GROUP BY:
  - sort T in 2 passes:  $4 * 20 = 80$  I/Os
  - HAVING, aggregations: no additional I/O
- **total cost:  $500 + 20 + 80 = 600$  I/Os**

## Enumeration of Alternative Plans

\* plans that use an index:

- available indexes on Students - a2
  - hash index on  $\langle \text{Age} \rangle$
  - B+ tree index on  $\langle \text{RoundedGPA} \rangle$
  - B+ tree index on  $\langle \text{RoundedGPA}, \text{SName}, \text{Age} \rangle$
- single-index access path:
  - choose the index that provides the most selective access path
  - apply  $\pi$ , nonprimary selection terms
  - compute grouping and aggregation operations
  - example:
    - use the hash index on Age to retrieve Students with Age = 20
      - cost: retrieve index entries and corresponding Students tuples
      - apply condition  $\text{RoundedGPA} > 5$  to each retrieved tuple
      - retain RoundedGPA and SName

## Enumeration of Alternative Plans

\* plans that use an index:

- single-index access path – example:
  - write out tuples to a temporary relation
  - sort the temporary relation by RoundedGPA to identify groups
  - apply the HAVING condition (to eliminate some groups)
- multiple-index access path:
  - several indexes using a2 / a3 match the selection condition, e.g., I1, I2
  - retrieve Rids<sub>I1</sub>, Rids<sub>I2</sub> using I1, I2
  - get tuples with rids in Rids<sub>I1</sub> ∩ Rids<sub>I2</sub> (tuples satisfying the primary selection terms of I1 and I2)
  - apply  $\pi$ , nonprimary selection terms
  - compute grouping and aggregation operations
  - example:
    - use the index on Age => rids of tuples with Age = 20 (R1)

## Enumeration of Alternative Plans

\* plans that use an index:

- multiple-index access path – example:
  - index on RoundedGPA => rids of tuples with RoundedGPA > 5 (R2)
  - retrieve tuples with rids in  $R1 \cap R2$
  - keep only RoundedGPA and SName
  - write out tuples to a temporary relation
  - sort the temporary relation by RoundedGPA to identify groups
  - apply the HAVING condition (to eliminate some groups)
- sorted index access path:
  - works well when the index is clustered
  - B+ tree index I with search key K
  - GROUP BY attributes – prefix of K
  - use the index to retrieve tuples in the order required by the GROUP BY clause

## Enumeration of Alternative Plans

\* plans that use an index:

- apply  $\sigma$ ,  $\pi$
- compute aggregation operations
- sorted index access path – example:
  - use the B+ tree index on RoundedGPA to retrieve Students tuples with RoundedGPA > 5, ordered by RoundedGPA
  - aggregations in HAVING, SELECT - computed on-the-fly
- index-only access path:
  - dense index I with search key K
  - all the attributes in the query are included in K

=> index-only scan, don't need to retrieve tuples from the relation

  - data entries: apply  $\sigma$ , perform  $\pi$ , sort the result (to identify groups), compute aggregate operations
    - \* obs. index I doesn't need to match the selections in the WHERE clause

## Enumeration of Alternative Plans

\* plans that use an index:

- index-only access path

\* obs. I – tree index, GROUP BY attributes – prefix of K

=> can avoid sorting

- example:

- use the B+ tree index on <RoundedGPA, SName, Age> to retrieve entries with RoundedGPA > 5, ordered by RoundedGPA

- select entries with Age = 20

- aggregation operations in the HAVING and SELECT clauses - computed on-the-fly

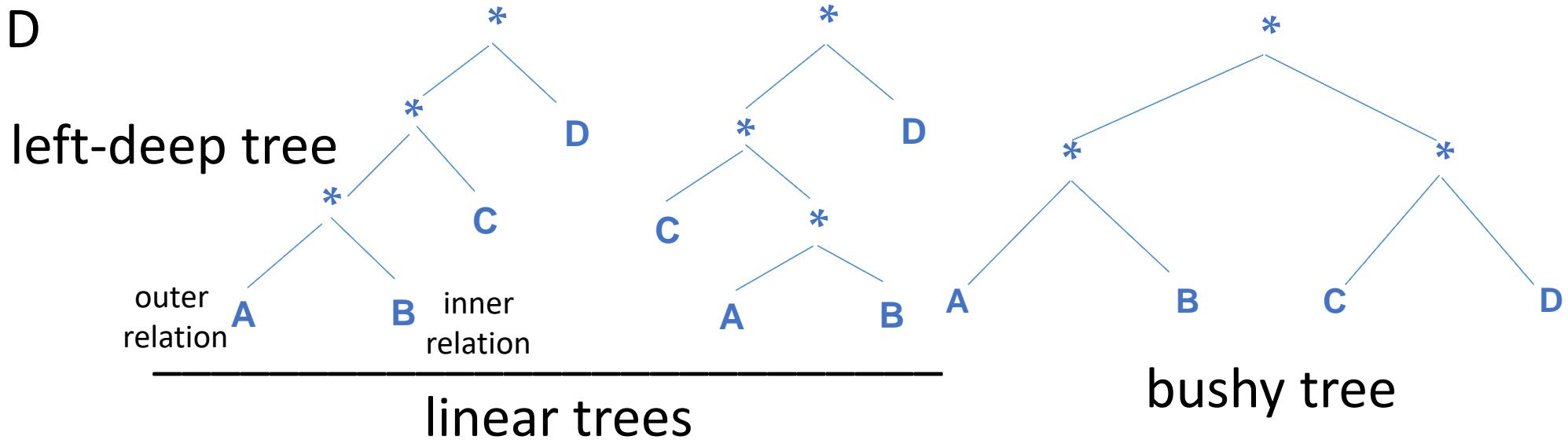
## Enumeration of Alternative Plans

- queries with several relations in the FROM clause:
  - joins, cross-products => queries can be quite expensive
  - different join orders => intermediate relations of widely varying sizes => plans with very different costs
- class of plans considered by the optimizer
- plan enumeration

## Enumeration of Alternative Plans

- queries with several relations in the FROM clause

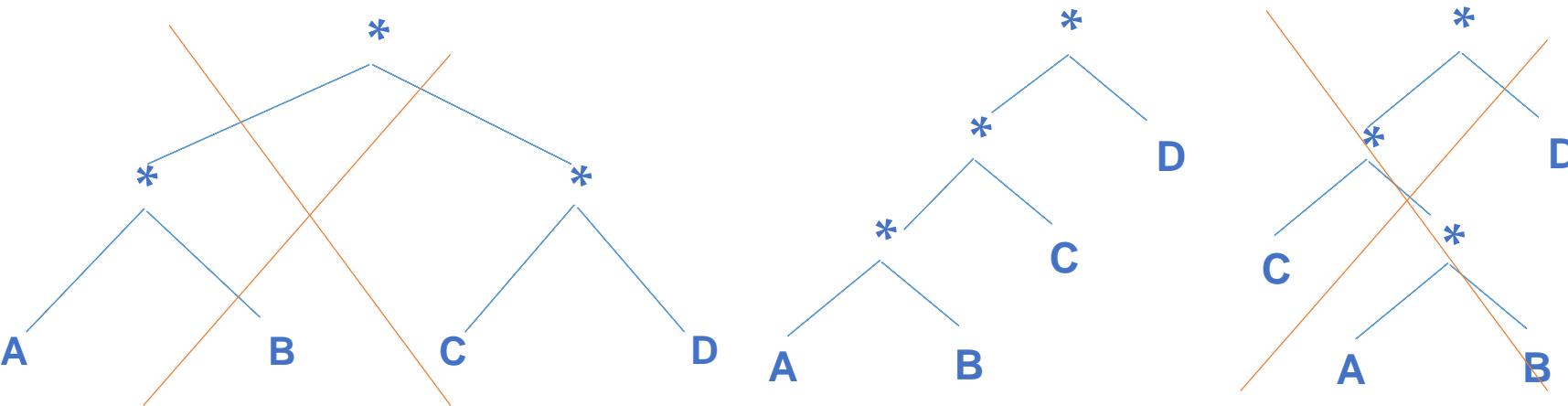
$A * B * C * D$



- linear trees:
  - at least one child of a join node is a base relation
- left-deep trees:
  - the right child of each join node is a base relation
- bushy tree: not linear

## Enumeration of Alternative Plans

- queries with several relations in the FROM clause
- fundamental decision in System R:
  - only *left-deep trees* are considered



- motivation:
  - number of joins increases => number of alternative plans increases quickly => must prune the search space
  - left-deep trees generate all fully pipelined plans (all joins are evaluated using pipelining)

## Enumeration of Alternative Plans

- queries with several relations in the FROM clause
  - \* obs: not all plans based on left-deep trees are fully pipelined
    - e.g., sort-merge join: outer tuples may have to be retrieved in a particular order, so the outer relation should be materialized
- multiple passes:
  - pass 1:
    - find best 1-relation plan for each relation
  - pass 2:
    - find best way to join the result of each 1-relation plan (as the outer argument) with another relation (all 2-relation plans)
  - ...
  - repeat until the obtained plans contain all the relations in the query
- for each subset of relations, retain: the cheapest overall plan & the cheapest plan for each interesting ordering of tuples

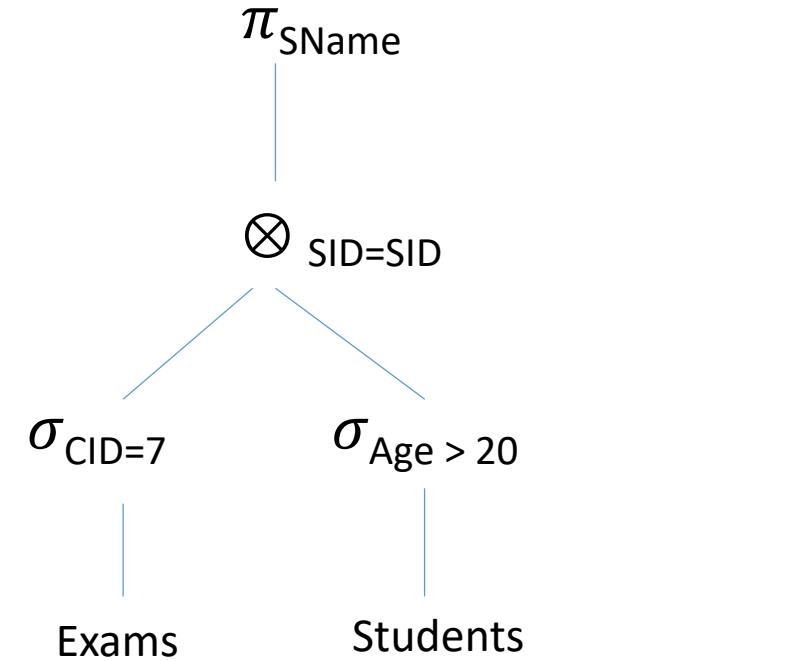
## Enumeration of Alternative Plans

- queries with several relations in the FROM clause
  - GROUP BY, aggregates are handled as a final step:
    - use a plan with an interesting ordering of tuples
    - use an additional sorting operator
- obs. avoid cross-products if possible

## Example

- unclustered indexes using a2
- Students: B+ tree index on Age, hash index on SID
- Exams: B+ tree index on CID

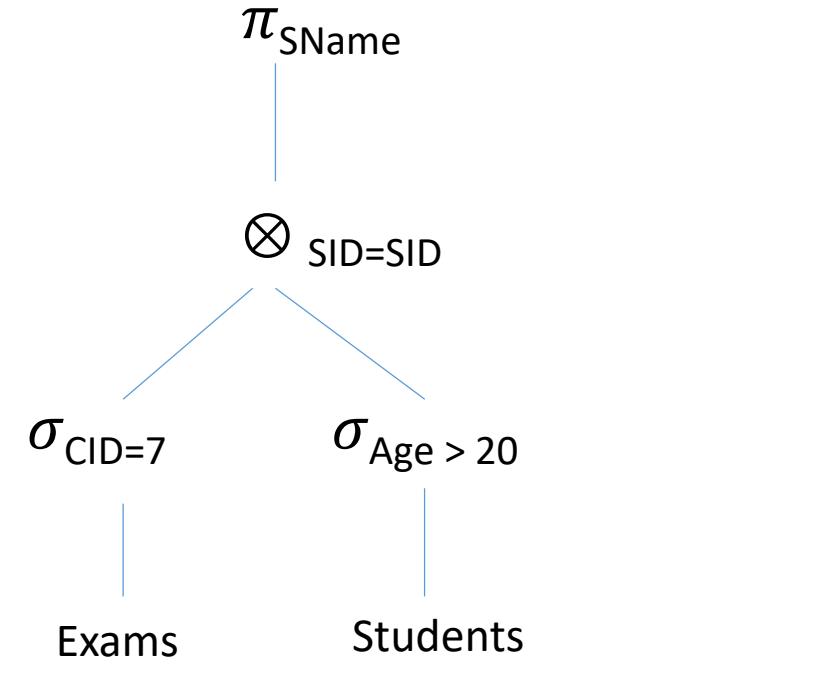
- pass 1:
  - Students – 3 possible access paths:  
B+ tree index, hash index, file scan
    - *Age > 20* matches the B+ tree index  
on Age
    - hash index / file scan => probably much higher cost
    - keep the plan using the B+ tree index => retrieved tuples are ordered by Age
  - Exams – 2 possible access paths
    - *CID = 7* matches the B+ tree index on CID
    - better than file scan => keep the plan using the B+ tree index



## Example

- Students: B+ tree index on Age, hash index on SID
- Exams: B+ tree index on CID
- pass 2:

- consider (the result of) each plan from pass 1 as the outer argument and analyze how to join it with the other relation
- e.g., Exams – outer argument
  - examine alternative access methods / join methods
  - access methods:
    - need Students tuples s.t. SID = *value* from outer tuple and Age > 20
    - hash index => Students tuples s.t. SID = *value* from outer tuple
    - B+ tree index => Students tuples s.t. Age > 20
  - join methods:
    - consider all available methods



## Example

- Students: B+ tree index on Age, hash index on SID
- Exams: B+ tree index on CID
- pass 2:
  - consider (the result of) each plan from pass 1 as the outer argument and analyze how to join it with the other relation
  - e.g., Students – outer argument
    - access / join methods
    - access methods:
      - need Exams tuples s.t. SID = *value* from outer tuple and CID = 7
    - join methods:
      - consider all available methods
  - retain cheapest plan overall!

## Nested Queries - optional

- usually handled using some form of nested loops evaluation
- correlated query - typical evaluation strategy:
  - the inner block is evaluated for each tuple of the outer block
- some strategies are not considered
  - e.g., index on SID in Students
  - best plan could be INLJ with Exams as the outer argument, and Students as the inner one; such a plan is never considered by the optimizer
- the unnested version of the query is typically optimized better

```
SELECT S.SName  
FROM Students S  
WHERE EXISTS  
(SELECT *  
FROM Exams E  
WHERE E.CID=7  
AND E.SID=S.SID)
```

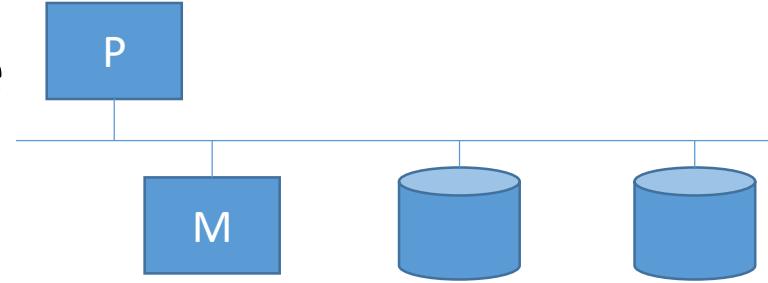
Equivalent unnested query:  
SELECT S.SName  
FROM Students S, Exams E  
WHERE E.SID=S.SID AND E.CID = 7

# Database Management Systems

Lecture 10  
Distributed Databases

## \* centralized DB systems

- all data at a single site, with one processor and one memory; the data is stored on one or several hard disk drives
- assumed each transaction is processed sequentially
- centralized lock management
- if the processor / memory fails => entire system fails



## \* distributed systems

- the data is stored at several sites, i.e., multiple processors (+ memories)
- each site is managed by a DBMS that can run independently; these autonomous components can also be heterogeneous

=> impact on: query processing, query optimization, concurrency control, recovery

# Distributed Database Systems

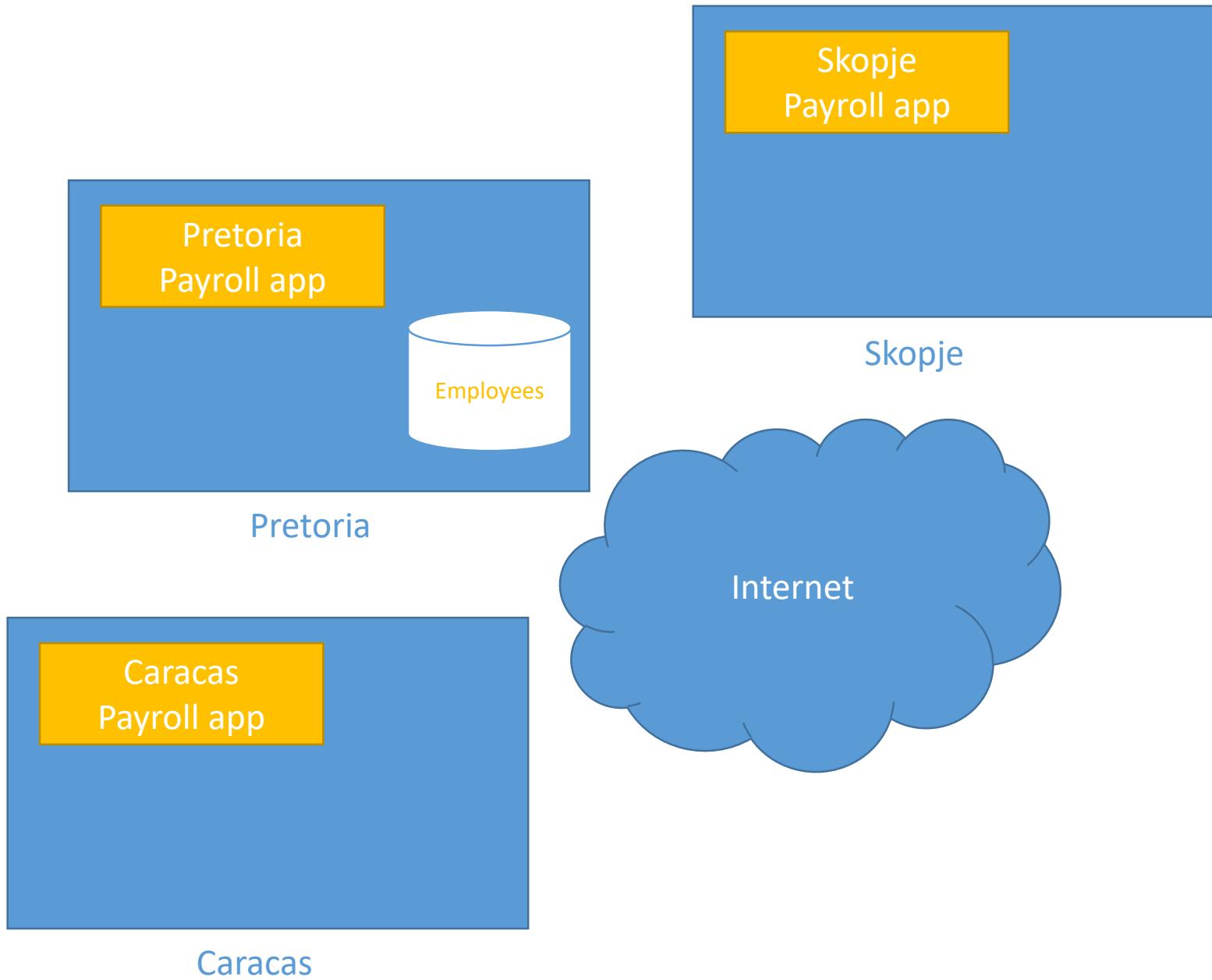
\* properties:

- distributed data independence (extension of the physical and logical data independence principles):
  - users can write queries without knowing / specifying the actual location of the data
  - cost-based query optimization that takes into account communication costs & differences in computation costs across sites
- distributed transaction atomicity
  - users can write transactions accessing multiple sites just as they would write local transactions
  - transactions are still atomic (if the transaction commits, all its changes persist; if it aborts, none of its changes persist)

## Distributed Databases - Motivating Example

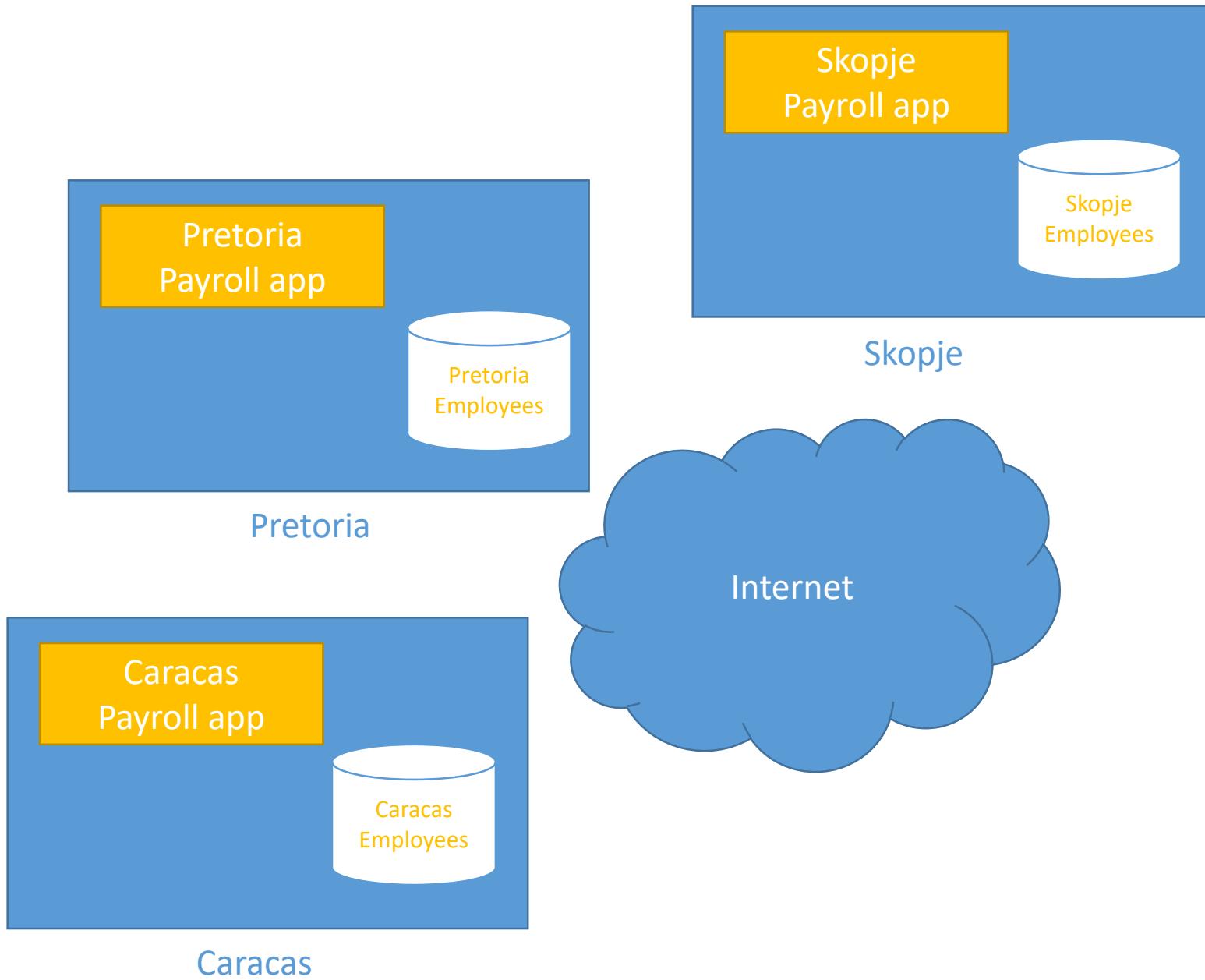
- company with offices in Pretoria, Skopje, Caracas
- in general, an employee's data is managed from the office where the employee works (payroll, benefits, hiring data, etc)
  - for instance, data about Skopje employees is managed from the Skopje office
- periodically, the company needs access to all employees' data, e.g.:
  - compute the total payroll expenses
  - compute the annual bonus, which depends on the global net profit
- where should we store employee data?

# Distributed Databases - Motivating Example



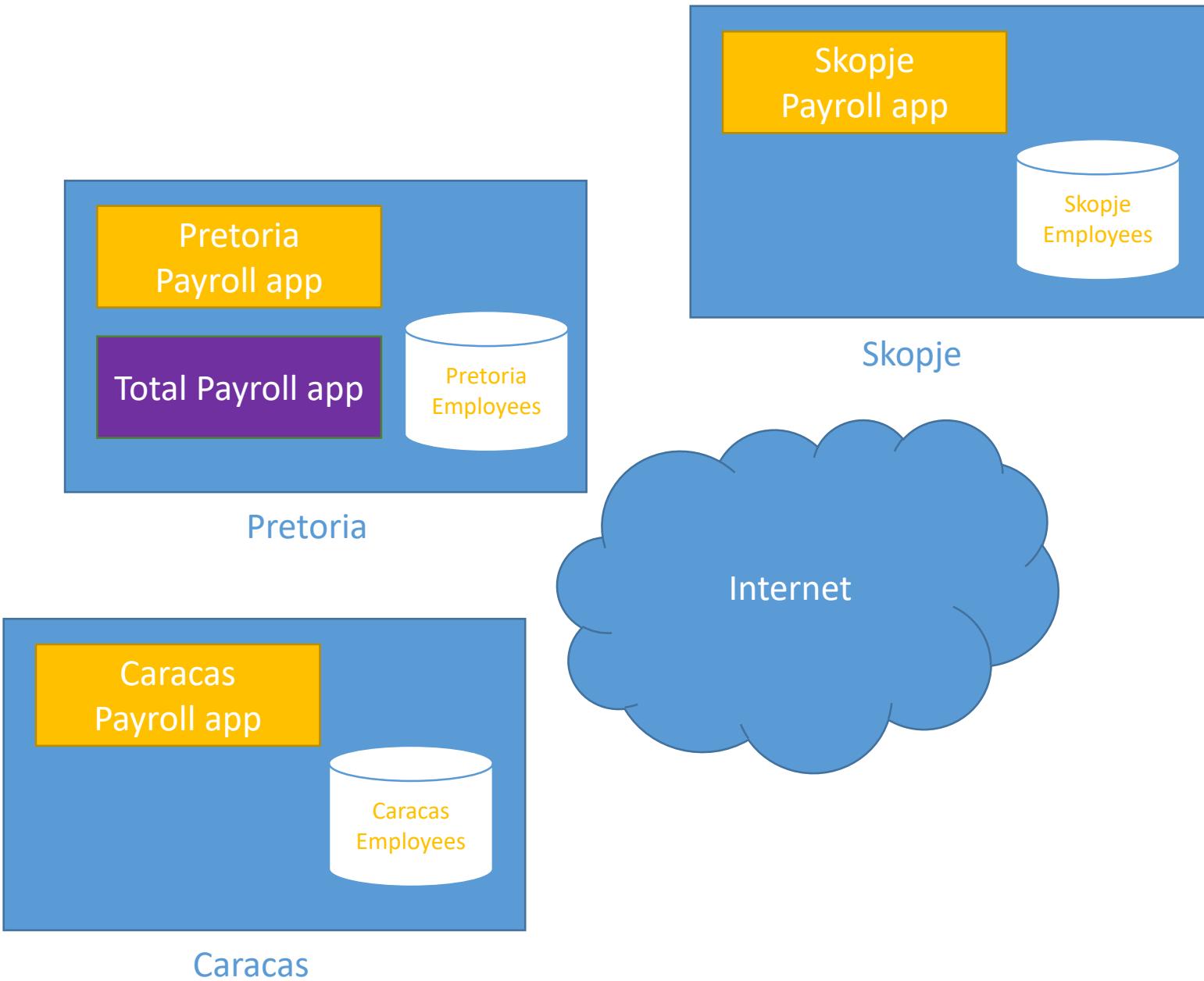
- *SiteX* payroll app – computing payrolls for *SiteX* employees
- suppose the DB is stored at the company's headquarters in Pretoria  
=> slow-running payroll apps in Caracas and Skopje
- moreover, if the Pretoria site becomes unavailable, payroll apps in Caracas and Skopje cannot access the required data

# Distributed Databases - Motivating Example



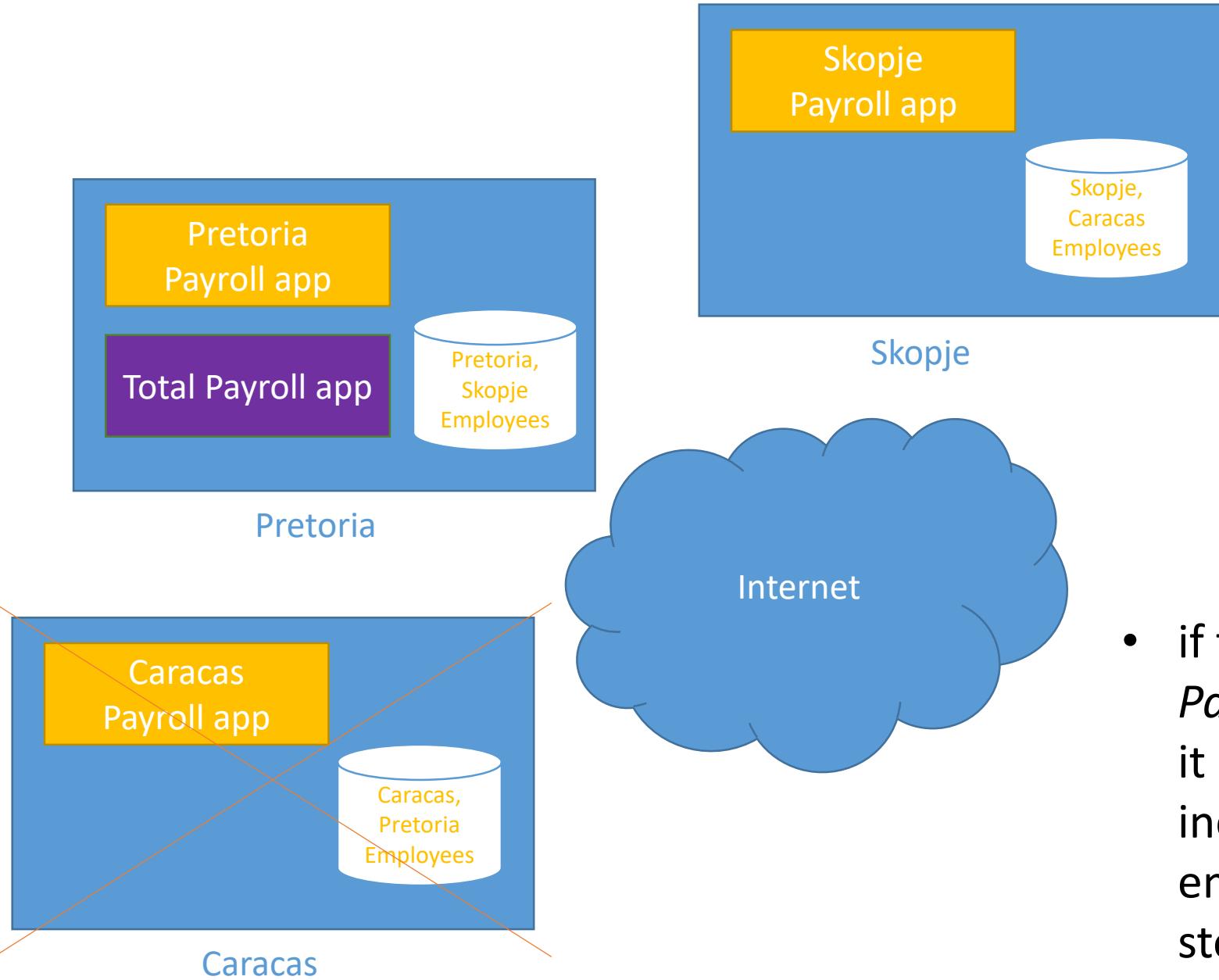
- store data about Pretoria employees in Pretoria, about Skopje employees in Skopje, etc  
=> improved performance for payroll apps in Caracas and Skopje
- if the Pretoria site becomes unavailable, payroll apps in Caracas and Skopje are still operational (as these apps only need data about Caracas employees and about Skopje employees, respectively)

# Distributed Databases - Motivating Example



- *Total Payroll app* (at the company's headquarters in Pretoria) computing the total payroll expenses
- this app needs to access employee data at all 3 sites, so generating the final results will last a little longer
- if the Caracas site crashes, the *Total Payroll app* cannot access all required data
- opportunities for parallel execution

# Distributed Databases - Motivating Example



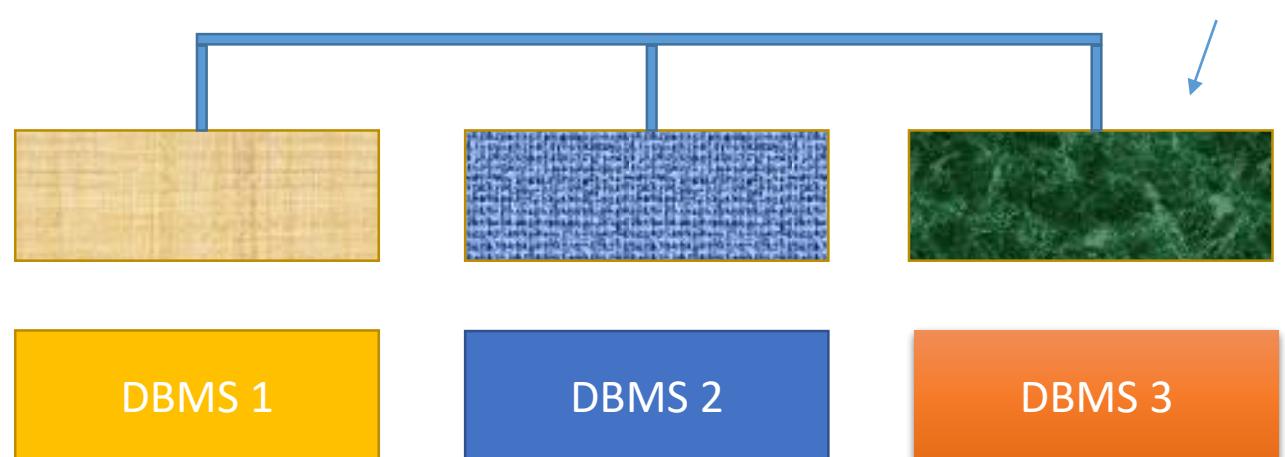
- data replication: at *Site X*, store data about *Site X* employees and data about employees from a different site, e.g., store data about Pretoria employees and Skopje employees in Pretoria (a copy of the Skopje employees data, which is stored in Skopje)
- if the Caracas site crashes, the *Total Payroll app* can continue to work, as it can access all required data, including data about the Caracas employees (a copy of this data being stored in Skopje)

# Types of Distributed Databases

- homogeneous:
  - the same DBMS software (e.g., SQL Server) at every site
- heterogeneous (multidatabase system):
  - different DBMSs at different sites (e.g., SQL Server at one site, Oracle at another site); different relational DBMSs or even non-relational DBMSs

## \* *gateway*:

- a software component that accepts requests (in some subset of SQL), submits them to local DBMSs, and returns the answers to the requestors (in some standard format)
- such components insulate users from differences among various DBMSs



## Distributed Databases - Challenges

### \* distributed database design:

- deciding where to store the data
- depends on the data access patterns of the most important applications (how are they accessing the data)
- two sub-problems: *fragmentation* and *allocation*

### \* distributed query processing:

- centralized query plan (treated in previous lectures)
  - objective: minimize the number of disk I/Os; execute the query as fast as possible
- distributed context – there are additional factors to consider:
  - communication costs
  - opportunity for parallelism

=> the space of possible query plans for a given query is much larger!

## Distributed Databases - Challenges

### \* distributed concurrency control:

- transaction schedules must be globally serializable (on all sites)
- distributed deadlock management (e.g., deadlock involving transactions T1 and T2, with T1 executing at site S1, and T2 executing at site S2)

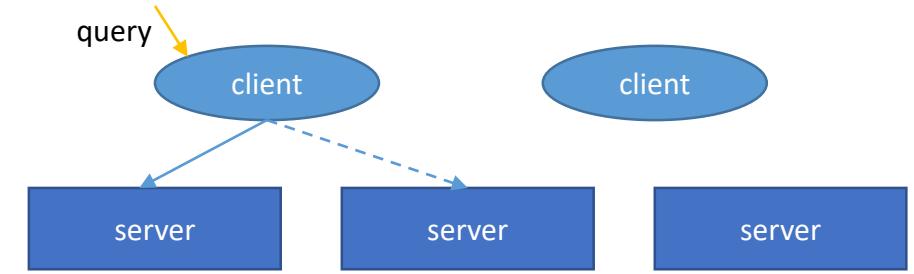
### \* reliability of distributed databases:

- transaction failures
  - one or more processors may fail
  - the network may fail
- data must be synchronized

# Distributed DBMS Architectures

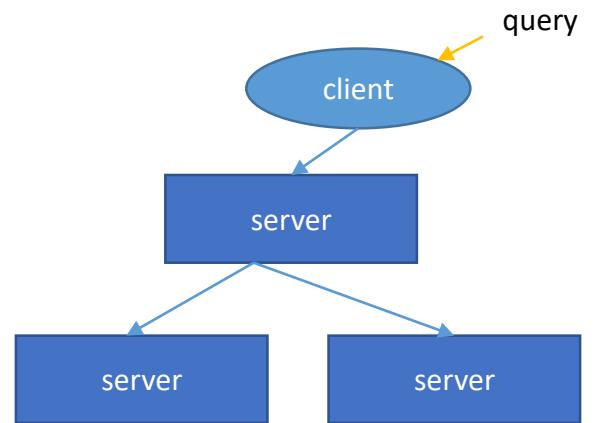
## \* Client-Server Systems

- one or several client processes, one or several server processes
- the client submits the query to a single site
- query processing is done at the server



## \* Collaborating Server Systems

- queries can span several sites, e.g.:
- server S receives query Q: *select all employees*; it generates subqueries *select all Caracas employees*, *select all Skopje employees*, ...
- servers in Caracas, Skopje, ... execute these subqueries
- server S assembles the subqueries' answer sets => the answer set of the initial query Q



## Storing Data in a Distributed DBMS

- accessing relations at remote sites => communication costs
  - \* example:
    - Pretoria: the site stores the *Employees* relation, holding data about all employees in the company
    - Skopje: a local manager wants to obtain the average salary for Skopje employees; she must access the relation stored in Pretoria
  - reduce such costs: *fragmentation / replication*
    - a relation can be partitioned into *fragments*, which are stored across several sites (a fragment is kept where it's most often accessed)
- \* example:
  - partition the *Employees* relation into fragments *PretoriaEmployees*, *SkopjeEmployees*, etc
  - store fragment *PretoriaEmployees* in Pretoria, fragment *SkopjeEmployees* in Skopje, etc

## Storing Data in a Distributed DBMS

- accessing relations at remote sites => communication costs
- reduce such costs: *fragmentation / replication*
  - a relation can be *replicated* at each site where it's needed the most
    - \* example:
      - suppose the *Employees* relation is frequently needed in Beijing, New York, and Bucharest
      - *Employees* can be replicated at the Bucharest site, the New York one, and in Beijing

# Storing Data in a Distributed DBMS

\* *fragmentation*: break a relation into smaller relations (fragments); store the fragments instead of the relation itself

- *horizontal / vertical / hybrid*

\* example – relation Accounts(accnum, name, balance, branch):

- horizontal fragmentation

- fragment: subset of rows
- n selection predicates => n fragments (n record sets)
- horizontal fragments should be disjoint
- reconstruct the original relation: take the union of the horizontal fragments
- $\sigma_{\text{branch}=\text{'Eroilor'}}(\text{Accounts})$ ,  
 $\sigma_{\text{branch}=\text{'Napoca'}}(\text{Accounts})$ ,  
 $\sigma_{\text{branch}=\text{'Motilor'}}(\text{Accounts}) \Rightarrow$

R
1, Radu, 250, Eroilor
2, Ana, 200, Napoca
3, Ionel, 150, Motilor
4, Maria, 400, Eroilor
5, Andi, 600, Napoca
6, Calin, 250, Eroilor
7, Iulia, 350, Motilor

R1	1, Radu, 250, Eroilor 4, Maria, 400, Eroilor 6, Calin, 250, Eroilor
R2	2, Ana, 200, Napoca 5, Andi, 600, Napoca
R3	3, Ionel, 150, Motilor 7, Iulia, 350, Motilor

# Storing Data in a Distributed DBMS

- vertical fragmentation
  - fragment: subset of columns
  - performed using projection operators
    - must obtain a good decomposition\*
    - reconstruction operator: natural join
      - $\pi_{\{\text{accnum}, \text{name}\}}(\text{Accounts})$
      - $\pi_{\{\text{accnum}, \text{balance}, \text{branch}\}}(\text{Accounts})$

R1	R2
1, Radu	1, 250, Eroilor
2, Ana	2, 200, Napoca
3, Ionel	3, 150, Motilor
4, Maria	4, 400, Eroilor
5, Andi	5, 600, Napoca
6, Calin	6, 250, Eroilor
7, Iulia	7, 350, Motilor

- hybrid fragmentation
  - horizontal fragmentation + vertical fragmentation

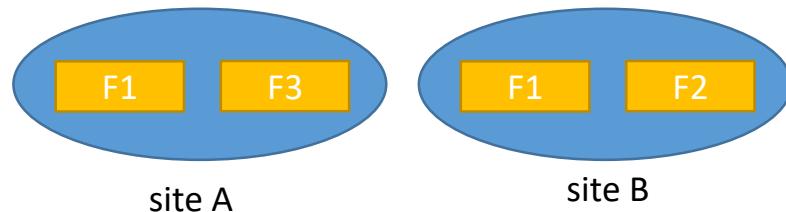
R1	R2
1, Radu	1, 250, Eroilor
2, Ana	2, 200, Napoca
3, Ionel	3, 150, Motilor
6, Calin	6, 250, Eroilor
R3	R4
4, Maria	4, 400, Eroilor
5, Andi	5, 600, Napoca
7, Iulia	7, 350, Motilor

\* see *Databases – Normal Forms*

## Storing Data in a Distributed DBMS

\* *replication*: store multiple copies of a relation or of a relation fragment; an entire relation or one or several fragments of a relation can be replicated at one or several sites

\* example: relation R is partitioned into fragments F1, F2, F3; fragment F1 is stored at site A and at site B:



- motivation:
  - increased availability of data: query Q uses fragment F1 from site A; if site A goes down or a communication link fails, the query can use another active server (e.g., site B)
  - faster query evaluation: can use a local copy of the data to avoid communication costs, e.g., query Q at site A can use the local copy of F1, it doesn't need to access the copy of F1 from site B
- types of replication: synchronous versus asynchronous
  - how are the copies of the data kept current when the relation is changed

## Updating Distributed Data

- synchronous replication:
  - suppose relation R has 3 copies: R1, R2, R3
  - transaction T modifies relation R
  - before T commits, it synchronizes all copies of R (R1, R2, R3 and R all contain the same data)
- => data distribution is transparent to the user: when reading relation R, the user doesn't need to know which copy of R it accesses, as all copies store correct, current data
- asynchronous replication:
  - transaction T modifies relation R
  - R's copies (R1, R2, R3) are synchronized periodically, i.e., it's possible that some of R's copies are outdated for brief periods of time
  - a transaction T2 reading 2 different copies of R (say R1 and R3) may see different data; but in the end, all copies of R will be synchronized

## Updating Distributed Data

- asynchronous replication:

=> users must be aware of the fact that the data is distributed, i.e., distributed data independence is compromised

- a lot of current systems are using this approach

## Updating Distributed Data – Synchronous Replication

- transaction T, object O (with several copies)
- objective:
  - no matter which copy of O it accesses, T should see the same value
  - 2 basic techniques: *voting* and *read-any write-all*

### \* *voting*

- to modify O, a transaction T1 must write a majority of its copies
- when reading O, a transaction T2 must read enough copies to make sure it's seeing at least one current copy
- e.g., O has 10 copies; T1 changes O: suppose T1 writes 7 copies of O; T2 reads O: it should read at least 4 copies to make sure at least one of them is current
- each copy has a version number (the copy that is current has the highest version number)

## Updating Distributed Data – Synchronous Replication

### \* *voting*

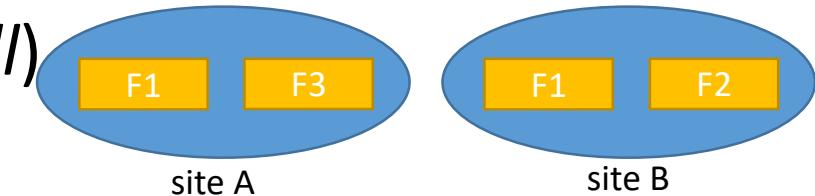
- not an attractive approach in most cases, because reads are usually much more common than writes (and reads are expensive in this approach)

### \* *read-any write-all*

- transaction T1 modifies O: T1 must write all copies of O
- transaction T2 reads O: T2 can read any copy of O (no matter which copy T2 reads, it will see current data, as T1 wrote all copies of O)
- fast reads (only one copy is read), slower writes (compared with the voting technique)
- most common approach to synchronous replication

## Updating Distributed Data – Synchronous Replication Costs

- before an update transaction T can commit, it must lock all copies of the modified relation / fragment (with *read-any write-all*)
  - suppose relation R is partitioned into fragments F1, F2, F3, stored at sites A and B (with F1 being stored at both sites)
  - if transaction T changes fragment F1 at site A, it must also lock the copy of F1 stored at site B!
  - such a transaction sends lock requests to remote sites; while waiting for the response, the transaction holds on to its other locks
- if there's a site or link failure, transaction T cannot commit until the network / involved sites are back up (if site B becomes unavailable in the example above, transaction T cannot commit until site B comes back up)
- even if there are no failures and locks are immediately obtained, T must follow an expensive commit protocol when committing (with several messages being exchanged) => asynchronous replication is more used



## Updating Distributed Data – Asynchronous Replication

- transaction T1 modifies object O; T1 is allowed to commit before all copies of O have been changed
  - => users must know:
    - which copy they are reading
    - that copies may be outdated for brief periods of time
- two approaches:
  - *primary site replication*
  - *peer-to-peer replication*
- \* difference: number of *updatable copies (master copies)*

# Updating Distributed Data – Asynchronous Replication

## \* *peer-to-peer* replication

- several copies of an object O can be *master copies* (i.e., updatable)
- changes to a master copy are propagated to the other copies of object O
- need a conflict resolution strategy for cases when two master copies are changed in a conflicting manner:
  - e.g., master copies at site 1 and site 2; at site 1: Dana's city is changed to Cluj-Napoca; at site 2: Dana's city is changed to Timișoara; which value is correct?
  - in general - ad hoc approaches to conflict resolution

# Updating Distributed Data – Asynchronous Replication

\* *peer-to-peer* replication

- best utilized when conflicts do not arise:
  - each master site owns a fragment (usually a horizontal fragment) and any 2 fragments that can be updated by different master sites are disjoint
    - \* example: store relation *Employees* at Skopje and Caracas
      - data about Skopje employees can only be updated in Skopje
      - data about Caracas employees can only be updated in Caracas
- updating rights are held by only one master site at a time

## Updating Distributed Data – Asynchronous Replication

### \* *primary site* replication

- exactly one copy of an object O is chosen as the *primary (master)* copy; this copy is published at the *primary site*
- secondary copies of the object (copies of the relation or copies of relation fragments) can be created at other sites (*secondary sites*)
- secondary sites can subscribe to the primary copy or to fragments of the primary copy
- changes to the primary copy are propagated to the secondary copies in 2 steps:
  - *capture* the changes made by committed transactions
  - *apply* these changes to secondary copies

## Updating Distributed Data – Asynchronous Replication

### \* *primary site* replication

- capture: *log-based capture / procedural capture*
  - log-based capture:
    - the log (kept for recovery purposes) is used to generate the *Change Data Table* (CDT) structure: write log tail to stable storage => write all log records affecting replicated relations to the CDT
    - changes of aborted transactions must be removed from the CDT
    - in the end, CDT contains only update log records of committed transactions
    - suppose committed transaction T1 has executed 3 UPDATE operations on (the replicated) relation *Employees*; then the CDT will include 3 update log records, describing the UPDATE operations

# Updating Distributed Data – Asynchronous Replication

\* *primary site* replication

- capture: *log-based capture / procedural capture*
  - procedural capture
    - capture is performed through a procedure that is automatically invoked (e.g., a trigger)
    - the procedure takes a snapshot of the primary copy
    - *snapshot*: a copy of the relation
  - log-based capture:
    - smaller overhead and smaller delay, but it depends on proprietary log details

## Updating Distributed Data – Asynchronous Replication

### \* *primary site* replication

- apply
  - applies changes collected in the Capture step (from the Change Data Table or snapshot) to the secondary copies:
    - the primary site can continuously send the CDT
    - or the secondary sites can periodically request a snapshot or (the latest portion of) the CDT from the primary site
    - each secondary site runs a copy of the Apply process

# Updating Distributed Data – Asynchronous Replication

\* *primary site replication* - optional

- the replica could be a view over the modified relation
  - replication: incrementally updating the view as the relation changes
- log-based capture + continuous apply
  - minimizes delay in propagating changes
- procedural capture + application-driven apply
  - most flexible way to process changes

# Distributed Query Processing

Researchers(RID: integer, Name: string, ImpactF: integer, Age: real)

AuthorContribution(RID: integer, PID: integer, Year: integer, Coord: string)

- Researchers
  - 1 tuple - 50 bytes
  - 1 page - 80 tuples
  - 500 pages
- AuthorContribution
  - 1 tuple - 40 bytes
  - 1 page - 100 tuples
  - 1000 pages
- estimate the cost of evaluation strategies:
  - number of I/O operations and number of pages shipped among sites, i.e., take into account communication costs
  - use  $t_d$  to denote the time to read / write a page from / to disk
  - use  $t_s$  to denote the time to ship a page from one site to another (e.g., from Skopje to Caracas)

# Distributed Query Processing

## \* nonjoin queries in a distributed DBMS

- impact of fragmentation / replication on simple operations
- scanning a relation, selection, projection
- horizontal fragmentation
  - all Researchers tuples with  $\text{ImpactF} < 6$  - stored at New York
  - all Researchers tuples with  $\text{ImpactF} \geq 6$  - stored at Lisbon

Q1 .

```
SELECT R.Age  
FROM Researchers R  
WHERE R.ImpactF > 4 AND R.ImpactF < 10
```

- the DBMS evaluates the query at New York and Lisbon, then takes the union of the obtained results to produce the result set for query Q1

## Distributed Query Processing

\* nonjoin queries in a distributed DBMS

- horizontal fragmentation

Q2.

```
SELECT AVG(R.Age)
FROM Researchers R
WHERE R.ImpactF > 4 AND R.ImpactF < 10
```

- the DBMS computes  $\text{SUM}(\text{Age})$  and number of Age values at New York and Lisbon; then based on this information, it computes the average age of all researchers with ImpactF in the specified range

Q3.

```
SELECT ...
FROM Researchers R
WHERE R.ImpactF > 7
```

- in this case, the DBMS evaluates the query only at Lisbon

## Distributed Query Processing

\* nonjoin queries in a distributed DBMS

- vertical fragmentation
  - RID, ImpactF - stored at New York (for all researchers)
  - Name, Age - stored at Lisbon (for all researchers)
  - the DBMS adds a field that contains the id of the corresponding tuple from Researchers to both fragments and rebuilds the Researchers relation by joining the 2 fragments on the common field (the tuple-id field)
  - the DBMS then evaluates the query over the reconstructed relation

# Distributed Query Processing

\* nonjoin queries in a distributed DBMS

- replication
  - Researchers relation stored at both New York and Lisbon
  - then Q1, Q2, Q3 can be executed at either New York or Lisbon
  - choosing the execution site - factors to consider:
    - the cost of shipping the result to the query site (e.g., the query site could be New York, Lisbon, or a 3<sup>rd</sup>, distinct site)
    - local processing costs:
      - can vary from one site to another
      - e.g., check the available indexes on Researchers at New York and Lisbon

# Distributed Query Processing

## \* join queries in a distributed DBMS

- can be quite expensive if the relations are stored at different sites
- Researchers stored at New York
- AuthorContribution stored at Lisbon
- evaluate *Researchers join AuthorContribution*

->

# Distributed Query Processing

## \* join queries in a distributed DBMS

- Researchers R - New York, AuthorContribution A - Lisbon, R join A

## \* fetch as needed

- page-oriented nested loops in New York
  - Researchers - outer relation
  - for every page in Researchers, bring in all the AuthorContribution pages from Lisbon
  - cost
    - scan Researchers:  $500t_d$
    - scan AuthorContribution + ship all AuthorContribution pages (for every Researchers page):  $1000(t_d + t_s)$

=> total cost:  $500t_d + 500,000(t_d + t_s)$

## Distributed Query Processing

### \* join queries in a distributed DBMS

- Researchers R - New York, AuthorContribution A - Lisbon, R join A

### \* fetch as needed

- page-oriented nested loops in New York
  - obs. bring in all AuthorContribution pages for each Researchers tuple => much higher cost
  - optimization
    - bring in AuthorContribution pages only once from Lisbon to New York
    - cache AuthorContribution pages at New York until the join is complete

## Distributed Query Processing

- join queries in a distributed DBMS
  - Researchers R - New York, AuthorContribution A - Lisbon, R join A

### \* fetch as needed

- page-oriented nested loops in New York
  - query not submitted at New York  
=> add the cost of shipping the result to the query site
  - RID - key in Researchers  
=> the result has 100,000 tuples (the number of tuples in AuthorContribution)
  - the size of a tuple in the result
    - $40 + 50 = 90$  bytes
    - the number of result tuples / page
      - $4000 / 90 = 44$

## Distributed Query Processing

- join queries in a distributed DBMS
  - Researchers R - New York, AuthorContribution A - Lisbon, R join A

### \* fetch as needed

- page-oriented nested loops in New York
  - query not submitted at New York
  - number of pages necessary to hold all the result tuples
    - $100,000/44 = 2273$  pages
  - the cost of shipping the result to another site (if necessary)
    - $2273 t_s$
    - higher than the cost of shipping both Researchers and AuthorContribution to the site ( $1500 t_s$ )

## References

- [Ra00] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (2<sup>nd</sup> Edition), McGraw-Hill, 2000
- [Da03] DATE, C.J., An Introduction to Database Systems (8<sup>th</sup> Edition), Addison-Wesley, 2003
- [Ga08] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book, Prentice Hall Press, 2008
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007,  
<http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Si10] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, McGraw-Hill, 2010, <http://codex.cs.yale.edu/avi/db-book/>
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems,  
<http://infolab.stanford.edu/~ullman/fcdb.html>

# Database Management Systems

Lecture 11  
Distributed Databases (II)\*

\* videos with further explanations on this topic will follow on YouTube

# Distributed Query Processing

Researchers(RID: integer, Name: string, ImpactF: integer, Age: real)

AuthorContribution(RID: integer, PID: integer, Year: integer, Coord: string)

- Researchers
  - 1 tuple - 50 bytes
  - 1 page - 80 tuples
  - 500 pages
- AuthorContribution
  - 1 tuple - 40 bytes
  - 1 page - 100 tuples
  - 1000 pages

# Distributed Query Processing

## \* join queries in a distributed DBMS

- Researchers R - New York, AuthorContribution A - Lisbon, R join A

## \* fetch as needed

- index nested loops join in New York
  - AuthorContribution - unclustered hash index on RID
  - 100,000 AuthorContribution tuples, 40,000 Researchers tuples
  - on average, a researcher has 2.5 corresponding tuples in AuthorContribution
  - for each Researchers tuple, retrieve the 2.5 corresponding tuples in AuthorContribution:
    - obtain the index page:  $1.2 t_d$  (on average)
      - +
        - read the matching records in AuthorContribution:  $2.5 t_d$

## Distributed Query Processing

### \* join queries in a distributed DBMS

- Researchers R - New York, AuthorContribution A - Lisbon, R join A

### \* fetch as needed

- index nested loops join in New York
    - for each Researchers tuple, retrieve the 2.5 corresponding tuples in AuthorContribution:  
=> cost per Researchers tuple:  $(1.2 + 2.5)t_d$ 
      - the pages containing these 2.5 tuples must also be shipped from Lisbon to New York
- => total cost:  $500t_d + 40.000(3.7t_d + 2.5t_s)$  (there are 40.000 records in Researchers)

# Distributed Query Processing

## \* join queries in a distributed DBMS

- Researchers R - New York, AuthorContribution A - Lisbon, R join A

## \* ship to one site

- ship Researchers to Lisbon, compute the join at Lisbon
    - scan Researchers, ship it to Lisbon, save Researchers at Lisbon:
      - cost:  $500(2t_d + t_s)$
      - compute *Researchers join AuthorContribution* at Lisbon
        - example: use improved version of Sort-Merge Join
          - combine the merging phase of sorting with the merging phase of the join => SMJ cost:  $3(\text{number of R pages} + \text{number of A pages})$
          - SMJ cost:  $3(500 + 1000) = 4500t_d$
- => total cost:  $500(2t_d + t_s) + 4500t_d$

## Distributed Query Processing

### \* join queries in a distributed DBMS

- Researchers R - New York, AuthorContribution A - Lisbon, R join A

### \* ship to one site

- ship Researchers to Lisbon, compute the join at Lisbon
  - total cost:  $500(2t_d + t_s) + 4500t_d$
- ship AuthorContribution to New York, compute the join at New York
  - total cost:  $1000(2t_d + t_s) + 4500t_d$

# Distributed Query Processing

## \* join queries in a distributed DBMS

- Researchers R - New York, AuthorContribution A - Lisbon, R join A

## \* semijoin

- at New York:
  - project Researchers onto the join columns (RID)
  - ship the projection to Lisbon
- at Lisbon:
  - join the Researchers projection with AuthorContribution
  - => the so-called *reduction of AuthorContribution with respect to Researchers*
  - ship the reduction of AuthorContribution to New York
- at New York:
  - join Researchers with the reduction of AuthorContribution

# Distributed Query Processing

## \* join queries in a distributed DBMS

- Researchers R - New York, AuthorContribution A - Lisbon, R join A

## \* semijoin

- tradeoff:
  - the cost of computing and shipping the projection  
+  
• the cost of computing and shipping the reduction versus
    - the cost of shipping the entire AuthorContribution relation
  - very useful if there is a selection on one of the relations

# Distributed Query Processing

## \* join queries in a distributed DBMS

- Researchers R - New York, AuthorContribution A - Lisbon, R join A

## \* bloomjoin

- at New York:
  - compute a bit-vector of some size k
    - hash Researchers tuples (using the join column RID) into the range 0 to k-1
    - if some tuple hashes to  $i$ , set bit  $i$  to 1 ( $i$  from 0 to k-1)
      - otherwise (no tuple hashes to  $i$ ), set bit  $i$  to 0
    - ship the bit-vector to Lisbon
- at Lisbon:
  - hash each AuthorContribution tuple (using the join column RID) into the range 0 to k-1, with the same hash function

# Distributed Query Processing

## \* join queries in a distributed DBMS

- Researchers R - New York, AuthorContribution A - Lisbon, R join A

## \* bloomjoin

- at Lisbon:
  - discard tuples with a hash value  $i$  that corresponds to a 0 bit in the Researchers bit-vector
    - => *reduction of AuthorContribution with respect to Researchers*
    - ship the reduction to New York
- at New York:
  - join Researchers with the reduction

## Distributed Catalog Management

- keeping track of data distribution across sites
- one should be able to identify each replica of each fragment for a relation that is fragmented and replicated
- local autonomy should not be compromised
  - solution - names containing several fields:
    - global relation name:
      - $\langle \text{local-name}, \text{birth-site} \rangle$
    - global replica name:
      - $\langle \text{local-name}, \text{birth-site}, \text{replica\_id} \rangle$

# Distributed Catalog Management

- centralized system catalog
  - stored at a single site
  - contains data about all the relations, fragments, replicas
  - vulnerable to single-site failures
  - can overload the server

## Distributed Catalog Management

- global system catalog maintained at each site
  - every copy of the catalog describes all the data
  - not vulnerable to single-site failures (the data can be obtained from a different site)
  - local autonomy is compromised:
    - changes to a local catalog must be propagated to all the other sites

## Distributed Catalog Management

- local catalog maintained at each site
  - each site keeps a catalog that describes local data, i.e., copies of data stored at the site
  - the catalog at the birth-site for a relation keeps track of all the fragments / replicas of the relation
  - create a new replica / move a replica to another site:
    - must update the catalog at the birth-site
  - not vulnerable to single-site failures & doesn't compromise local autonomy

## Distributed Transaction Management

- a transaction submitted at a site S could ask for data stored at several other sites
- *subtransaction* - the activity of a transaction at a given site
- context: Strict 2PL with deadlock detection
- problems:
  - distributed concurrency control
    - lock management when objects are stored across several sites
    - deadlock detection
  - distributed recovery
    - transaction atomicity
      - all the effects of a committed transaction (across all the sites it executes at) are permanent
      - none of the actions of an aborted transaction are allowed to persist

# Distributed Transaction Management

- distributed concurrency control
  - lock management
    - techniques – synchronous / asynchronous replication
      - which objects will be locked
    - concurrency control protocols
      - when are locks acquired / released
    - lock management
      - *centralized*
      - *primary copy*
      - *fully distributed*

# Distributed Transaction Management

- distributed concurrency control
  - lock management
    - centralized:
      - one site does all the locking for all the objects
      - vulnerable to single-site failures
    - primary copy:
      - object O, primary copy PC of O stored at site S with lock manager L
      - all requests to lock / unlock some copy of O are handled by L
      - not vulnerable to single-site failures
      - read some copy C of O stored at site S2:  
=> communicate with both S and S2

# Distributed Transaction Management

- distributed concurrency control
  - lock management
    - fully distributed:
      - object O, some copy C of O stored at site S with Lock Manager L
        - requests to lock / unlock C are handled by L (the site where the copy is stored)
      - one doesn't need to access 2 sites when reading some copy of O

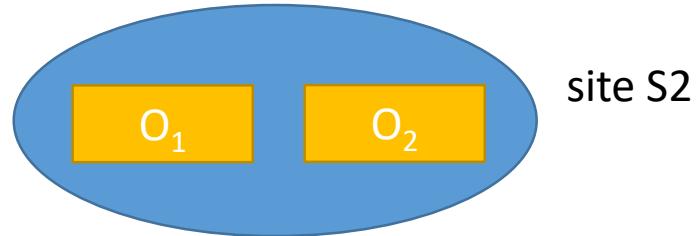
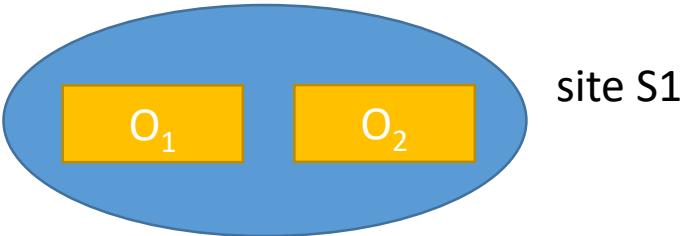
## Distributed Transaction Management

- distributed concurrency control
  - detect and resolve deadlocks
  - each site maintains a local waits-for graph
  - a cycle in such a graph indicates a deadlock
  - but a global deadlock can exist even if none of the local graphs contains a cycle

->

# Distributed Transaction Management

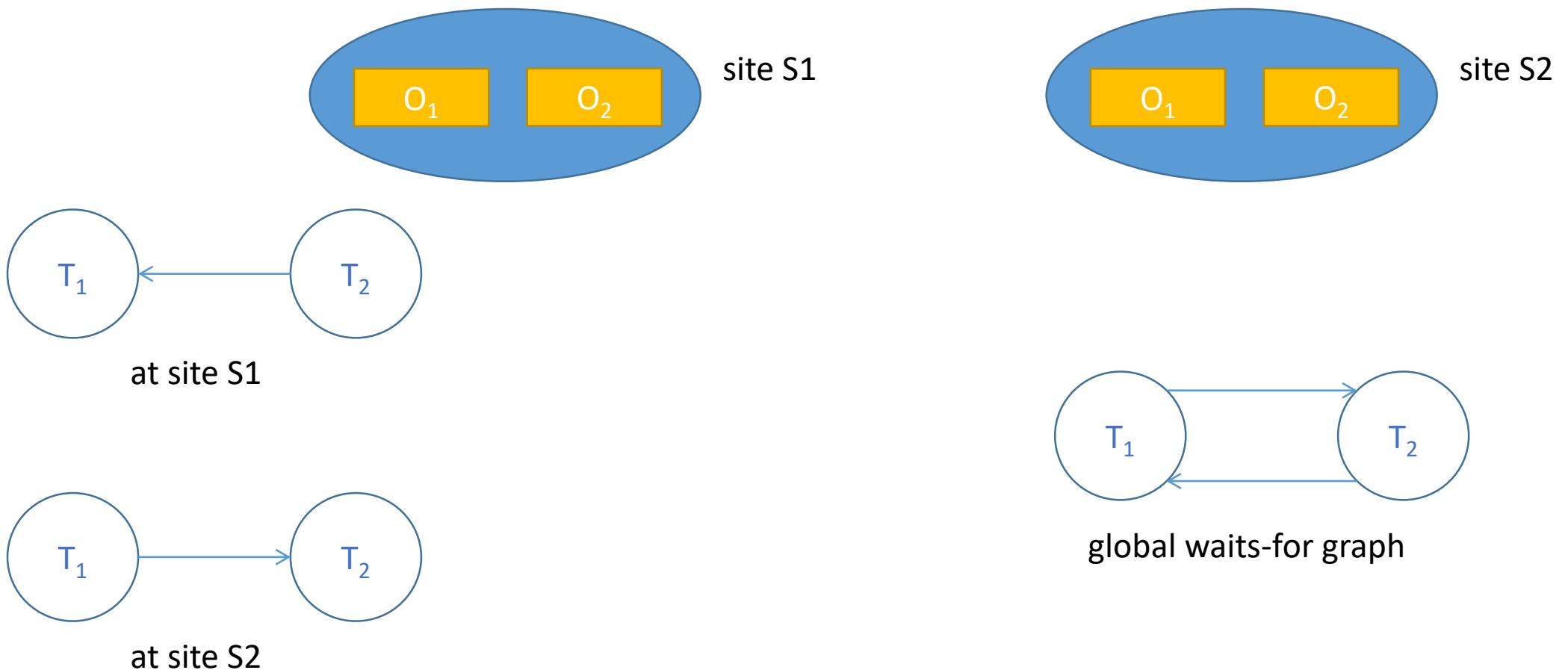
- distributed concurrency control – distributed deadlock
  - e.g., using *read-any write-all*



- T<sub>1</sub> wants to read O<sub>1</sub> and write O<sub>2</sub>
- T<sub>2</sub> wants to read O<sub>2</sub> and write O<sub>1</sub>
- T<sub>1</sub> acquires a S lock on O<sub>1</sub> and an X lock on O<sub>2</sub> at site S1
- T<sub>2</sub> obtains a S lock on O<sub>2</sub> and an X lock on O<sub>1</sub> at site S2
- T<sub>1</sub> asks for an X lock on O<sub>2</sub> at site S2
- T<sub>2</sub> asks for an X lock on O<sub>1</sub> at site S1

# Distributed Transaction Management

- distributed concurrency control – distributed deadlock
  - e.g., using *read-any write-all*



## Distributed Transaction Management

- distributed concurrency control – distributed deadlock
  - distributed deadlock detection algorithms
    - *centralized*
    - *hierarchical*
    - based on a *timeout* mechanism

## Distributed Transaction Management

- distributed concurrency control – distributed deadlock
  - distributed deadlock detection algorithms
    - centralized:
      - all the local waits-for graphs are periodically sent to a single site S
      - S - responsible for global deadlock detection
      - the global waits-for graph is generated at site S
        - nodes
          - the union of the nodes in the local graphs
        - edges
          - there is an edge from node N1 to node N2 if such an edge exists in one of the local graphs

## Distributed Transaction Management

- distributed concurrency control – distributed deadlock
  - distributed deadlock detection algorithms
    - hierarchical:
      - sites are organized into a hierarchy, e.g., grouped by city, county, country, etc
      - each site periodically sends its local waits-for graph to its parent site
      - assumption: more deadlocks are likely across related sites
      - all the deadlocks are detected in the end

# Distributed Transaction Management

- distributed concurrency control – distributed deadlock
  - distributed deadlock detection algorithms
    - hierarchical:
      - example:  
RO ( CJ ( Cluj-Napoca, Dej, Turda ), BN ( Bistrita, Beclean ) )

Cluj-Napoca : T1 -> T2

Dej: T2 -> T3

Turda: T3 -> T4 <- T7

Bistrita: T5 -> T6

Beclean: T4 -> T7 -> T6 -> T5

CJ: T1 -> T2 -> T3 -> T4 <- T7

BN: T5 <-> T6 <- T7 <- T4 (\*)

RO: T1 -> T2 -> T3 -> T4 <-> T7 -> T6 <-> T5

Obs RO: T5 or T6 has been aborted at (\*)

## Distributed Transaction Management

- distributed concurrency control – distributed deadlock
  - distributed deadlock detection algorithms
    - based on a timeout mechanism:
      - a transaction is aborted if it lasts longer than a specified interval
      - can lead to unnecessary restarts
      - however, the deadlock detection overhead is low
      - could be the only available option in a heterogeneous system (if the participating sites cannot cooperate, i.e., they cannot share their local waits-for graphs)

# Distributed Transaction Management

- distributed concurrency control – distributed deadlock
- phantom deadlocks
  - "deadlocks" that don't exist, but are detected due to delays in propagating local information
  - lead to unnecessary aborts
  - example:



at site S1



global waits-for graph



at site S2

- generate local waits-for graphs, send them to the site responsible for global deadlock detection
- T2 aborts (not because of the deadlock) => local waits-for graphs are changed, there is no cycle in the "real" global waits-for graph
- but the built waits-for graph does have a cycle, T1 could be chosen as a victim

# Distributed Transaction Management

- distributed recovery
  - more complex than in a centralized DBMS
  - new types of failure
    - network failure
    - site failure
  - commit protocol
    - either all the subtransactions of a transaction commit, or none of them does
  - normal execution
    - ensure all the necessary information is provided to recover from failures
  - a log is maintained at each site; it contains:
    - data logged in a centralized DBMS
    - actions carried out as part of the commit protocol

# Distributed Transaction Management

- distributed recovery
  - transaction T
    - coordinator
      - the Transaction Manager at the site where T originated
    - subordinates
      - the Transaction Managers at the sites where T's subtransactions execute

# Distributed Transaction Management

- distributed recovery
  - two-phase commit protocol (2PC)
    - exchanged messages + records written in the log
    - 2 rounds of messages, both initiated by the coordinator
      - *voting phase*
      - *termination phase*
    - any Transaction Manager can abort a transaction
    - however, for a transaction to commit, all Transaction Managers must decide to commit

# Distributed Transaction Management

- distributed recovery
  - two-phase commit protocol
    - the user decides to commit transaction T  
=> the commit command is sent to T's coordinator, initiating 2PC
      1. the coordinator sends a *prepare* message to each subordinate
      2. upon receiving a *prepare* message, a subordinate decides whether to commit / abort its subtransaction
        - the subordinate force-writes an *abort* or a *prepare\** log record
        - then it sends a *no* or *yes* message to the coordinator

\* *prepare* log records are specific to the commit protocol, they are not used in centralized DBMSs

# Distributed Transaction Management

- distributed recovery
  - two-phase commit protocol
    3.
      - if the coordinator receives *yes* messages from all subordinates:
        - it force-writes a *commit* log record
        - it then sends *commit* messages to all subordinates
      - otherwise (i.e., if it receives at least one *no* message or if it doesn't receive any message from a subordinate for a predetermined timeout interval)
        - it force-writes an *abort* log record
        - it then sends an *abort* message to each subordinate

# Distributed Transaction Management

- distributed recovery
  - two-phase commit protocol
    4.
      - upon receiving an *abort* message, a subordinate:
        - force-writes an *abort* log record
        - sends an *ack* message to the coordinator
        - aborts the subtransaction
      - upon receiving a *commit* message, a subordinate:
        - force-writes a *commit* log record
        - sends an *ack* message to the coordinator
        - commits the subtransaction

## Distributed Transaction Management

- distributed recovery
  - two-phase commit protocol
    5. after it receives *ack* messages from all subordinates, the coordinator writes an *end* log record for the transaction
  - sending a message – the sender has made a decision
  - the message is sent only after the corresponding log record has been forced to stable storage (to ensure the corresponding decision can survive a crash)
  - T is a committed transaction if the commit log record of T's coordinator has been forced to stable storage

# Distributed Transaction Management

- distributed recovery
  - two-phase commit protocol
    - log records for the commit protocol
      - record type
      - transaction id
      - coordinator's identity
    - the commit / abort log record for the coordinator also contains the identities of the subordinates

## Distributed Transaction Management

- distributed recovery – optional\*
  - restart after a failure – site S comes back up after a crash
    - if there is a *commit* or an *abort* log record for transaction T:
      - must redo / undo T
      - if S is T's coordinator:
        - periodically send *commit* / *abort* messages to subordinates until *ack* messages are received
        - write an *end* log record after receiving all *ack* messages
      - if there is a *prepare* log record for transaction T, but no *commit* / *abort*, S is one of T's subordinates
        - contact T's coordinator repeatedly until T's status is obtained
        - write a *commit* / an *abort* log record
        - redo / undo T

## Distributed Transaction Management

- distributed recovery – optional\*
  - restart after a failure – site S
    - if there are no *commit / abort / prepare* log records for T:
      - abort T, undo T
      - if S is T's coordinator, T's subordinates may subsequently contact S
  - blocking
    - if T's coordinator site fails, T's subordinates who have voted yes cannot decide whether to commit or abort T until the coordinator recovers, i.e., T is *blocked*
    - even if all the subordinates know each other (overhead - *prepare* messages), they are still blocked (unless one of them voted *no*)

# Distributed Transaction Management

- distributed recovery
  - link and remote site failures
    - current site S, remote site R, transaction T
    - if R doesn't respond during the commit protocol for T, either because R failed or the link failed:
      - if S is T's coordinator:
        - S should abort T
        - if S is one of T's subordinates, and has not voted yet:
          - S should abort T
        - if S is one of T's subordinates and has voted yes:
          - S is blocked until T's coordinator responds

# Distributed Transaction Management

- distributed recovery
  - 2PC – observations
    - *ack* messages
      - used to determine when can a coordinator C “forget” about a transaction T
      - C must keep T in the transaction table until it receives all *ack* messages
    - C fails after sending *prepare* messages, but before writing a *commit* / an *abort* log record
      - when C comes back up it aborts T
      - i.e., absence of information => T is presumed to have aborted
      - if a subtransaction doesn't change any data, its commit / abort status is irrelevant

# Distributed Transaction Management

- distributed recovery
  - 2PC with Presumed Abort
    - coordinator C, transaction T, some subordinate S, some subtransaction t
    - C aborts T
      - T is undone
      - C immediately removes T from the Transaction Table, i.e., it doesn't wait for *ack* messages
    - subordinates' names need not be recorded in C's abort log record
    - S doesn't send an *ack* message when it receives an *abort* message

# Distributed Transaction Management

- distributed recovery
  - 2PC with Presumed Abort
    - coordinator C, transaction T, some subordinate S, some subtransaction t
    - t doesn't change any data
      - t responds to *prepare* messages with a *reader* message, instead of *yes / no*
    - C subsequently ignores readers
    - if all subtransactions are readers, the 2nd phase of the protocol is not needed

## References

- [Ra00] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (2<sup>nd</sup> Edition), McGraw-Hill, 2000
- [Da03] DATE, C.J., An Introduction to Database Systems (8<sup>th</sup> Edition), Addison-Wesley, 2003
- [Ga08] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book, Prentice Hall Press, 2008
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007,  
<http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Si10] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, McGraw-Hill, 2010, <http://codex.cs.yale.edu/avi/db-book/>
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems,  
<http://infolab.stanford.edu/~ullman/fcdb.html>