CSC553 Advanced Database Concepts

Tanu Malik

School of Computing

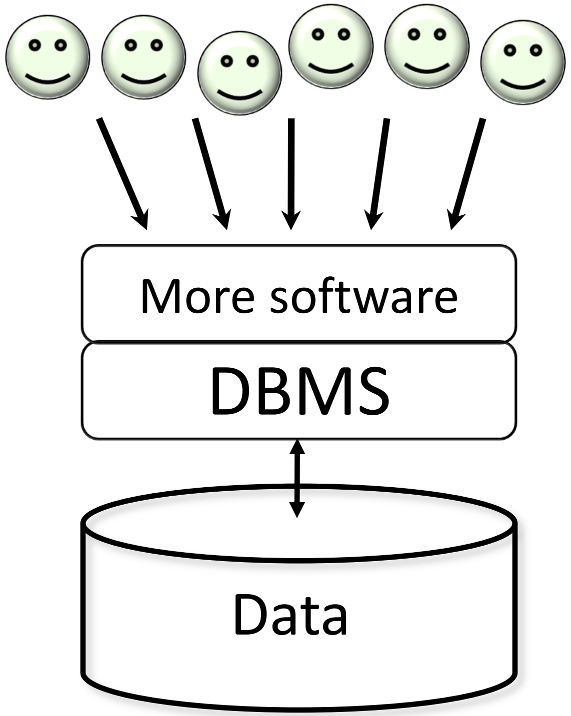
DePaul University

Transactions

Motivated by two independent requirements

- Concurrent database access
- Resilience to system failures

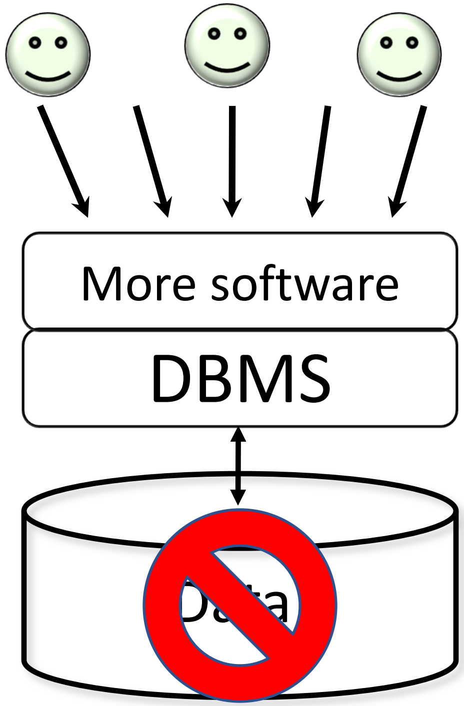
R1: Concurrent database access


More software

DBMS

Data

R2: Resilience to system failures



Example Transaction

- Withdraw $100 from an ATM machine

- ATM Transaction

```
balance ← read(account)
balance ← balance – 100
write(account, balance)
```

- The account is properly updated to reflect the new balance.

Lost Update Transaction

- Concurrent access of an account
- Two clients accessing the same account at the same time.

```
balance ← read(account)
balance ← balance – withdraw (100)
write(account, balance)

balance ← read(account)
balance ← balance – withdraw (200)
write(account, balance)
```

Before

Account Type	**Amount**
Checking	1000

After

Account Type	**Amount**
Checking	700

Lost Update Transaction

Before

Account Type	Amount
Checking	1000

- Concurrent access of an account
- Two clients accessing the same account at the same time.

1000
balance ← read(account)

900
balance ← balance – withdraw (100)

900
write(account, balance)

1000
balance ← read(account)

800
balance ← balance – withdraw (200)

800
write(account, balance)

Money is created!!!

Dirty Read Transaction

- Reading uncommitted data of transactions.

```
balance ← read(account)
balance ← balance – withdraw (100)
write(account, balance)

balance ← read(account)
balance ← balance – withdraw (200)
write(account, balance)
```

Before

Account Type	Amount
Checking	1000

After

Account Type	Amount
Checking	700

Dirty Read Transaction

Before

Account Type	Amount
Checking	1000

- Concurrent access of an account
- Two clients accessing the same account at the same time.

1000 balance ← read(account)	
900 balance ← balance – withdraw (100)	
900 write(account, balance)	
	900 balance ← read(account)
	700 balance ← balance – withdraw (200)
abort	
	700 write(account, balance)

Money is not dispensed but deducted!!

Inconsistent Read Transaction

- Concurrent access of accounts
- Two clients accessing the same accounts at the same time.

```
balance ← read(checking_account)
balance ← balance – withdraw (100)
write(checking_account, balance)
```

Thinks before
transfering 100

```
balance ← read(savings_account)
balance ← balance + withdraw (100)
write(savings_account, balance)
```

```
balance ← read(sum(accounts))
```

Before sum= 2500

Account Type	**Amount**
Checking	1000
Savings	1500

inBetween sum= 2400

Account Type	**Amount**
Checking	900
Savings	1500

After sum= 2500

Account Type	**Amount**
Checking	900
Savings	1600

Incorrect total sum is reported

Inconsistent Read Transaction

- Reconsider the transfer from checking to saving account

- Transaction 1

Update Accounts
Set balance = balance – 500
Where customer = 1904 and account_type = ‘Checking’

Update Accounts
Set balance = balance + 500
Where customer = 1904 and account_type = ‘Saving’

- Transaction 2

Select sum(balance)
From Accounts
Where customer = 1904

Transaction 2 sees a temporary, inconsistent database state

Interrupted Transaction

- Money transfer from Checking to Savings

Step1: subtract money from checkings account

Step2: add money to savings

```
1. checking_balance ← read(checking_account)
2. checking_ balance ← checking_ balance – transfer (500)
3. write(checking_account, checking_balance)
4. savings_balance ← read(savings_account)
5. savings_ balance ← savings_ balance + transfer (500)
6. write(savings_account, savings_balance)
```

Before

Account Type	Amount
Checking	1000
Savings	1500

After

Account Type	Amount
Checking	500
Savings	2000

Interrupted **Transaction**

- Money transfer from Checking to Savings

Step1: subtract money from checkings account

Step2: add money to savings

```
1. checking_balance ← read(checking_account)
2. checking_ balance ← checking_ balance – transfer (500)
3. write(checking_account, checking_balance)
4. savings_balance ← read(savings_account)
5. savings_ balance ← savings_ balance + transfer (500)
6. write(savings_account, savings_balance)
```

- System crash (power outage, network failure)

prevents the final write to happen. Money is lost!

Before

Account Type	Amount
Checking	1000
Savings	1500

After

Account Type	Amount
Checking	500
Savings	2000

Unrepeatable Read Transaction

Before sum= 2500

Account Type	Amount
Checking	1000
Savings	1500

- Concurrent access of accounts
- Two clients accessing the same accounts at the same time.

```
                                      balance ← read(sum(accounts))
balance ← read(checking_account)
balance ← balance – withdraw (100)
write(checking_account, balance)
                                      balance ← read(sum(accounts))
```

After sum= 2400

Account Type	Amount
Checking	900
Savings	1500

Different sum of monies during a txn!!! Money was lost during a transaction

Unrepeatable Read Transaction

- Reconsider the transfer from checking to saving account

- Transaction 1

Update Accounts

Set balance = balance – 500

Where customer = 1904 and account_type = ‘Checking’

- Transaction 2

Select sum(balance)

From Accounts

Where customer = 1904

Select sum(balance)

From Accounts

Where customer = 1904

Transaction 2
reads
two different
balances

Summary

- Lost Update Anomaly
  - The effect of one transaction are lost due to an uncontrolled overwrite performed by a second transaction
- Inconsistent Read
  - A transaction reads the partial result of another transaction
- Dirty Reads
  - A transaction reads partial information of a transaction that potentially aborts.
- Interrupted Transactions
  - System state may not reflect the set of user actions.
- Unrepeatable Read
  - A transaction reads a value which is afterwards changed by another transaction (before the former transaction is finished). So, the first transaction operates on stale data.

Solution for both concurrency and failures: Transactions

- A transaction is a sequence of one or more SQL operations treated as a unit
    - Transactions appear to run in isolation
    - If the system fails, each transaction's changes are reflected either entirely or not at all

Transactions

- Transaction begins with a "`Begin Transaction`" statement
  - In Oracle a transaction implicitly begins with a Begin statement
- On "`commit`" transaction ends and new one begins
- Current transaction ends on session termination
- "`Autocommit`" turns each statement into transaction
  - CREATE TABLE statements are autocommitted

Transaction resilience to system failures

- Transaction Rollback (= Abort)
  - Undoes partial effects of transaction
- Can be system- or client-initiated

Each transaction is
“all-or-nothing,”
never left half done

```
Begin Transaction;
<get input from user>
SQL commands based on input
<confirm results with user>
If ans=‘ok’ Then Commit; Else Rollback;
```

Transaction: **Terms**

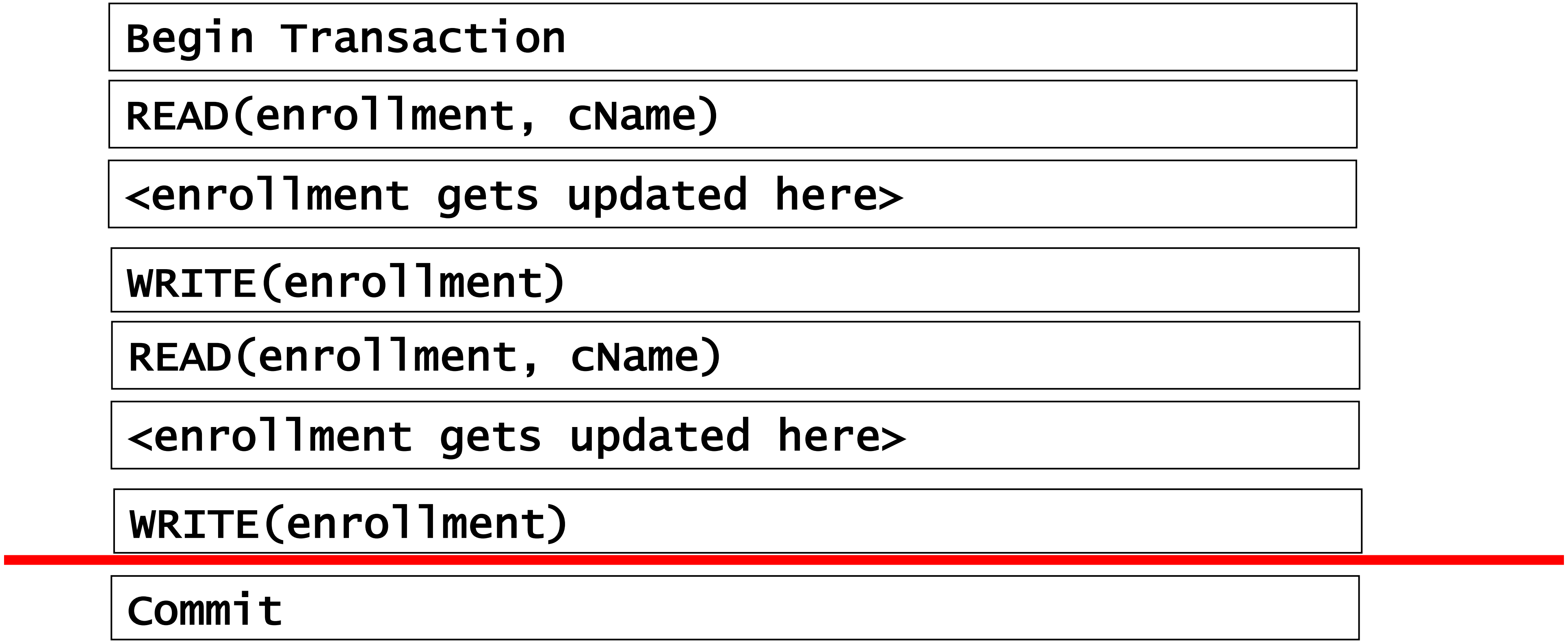
- Begin transaction: A transaction that is in progress/active.
- Commit: A transaction that completes its execution successfully.
- Abort: A transaction that does not complete its execution successfully.
- Rollback: Changes caused by an aborted transaction are undone.

➢A transaction that is successfully committed cannot undo its effects by aborting it.

Transaction Schedular must satisfy ACID Properties

Properties	Meaning
A Atomicity	all operations are reflected in the DB or none (all-or-nothing propert)y
C Consistency	an isolated transaction will preserve a consistent DB state
I Isolation	concurrent transactions “appear” to act in isolation
D Durability	commits are persistent and are reflected even if there are system failures

Transactions: **Atomicity (All-or-nothing)**



```
Begin Transaction
READ(enrollment, cName)
<enrollment gets updated here>
WRITE(enrollment)
READ(enrollment, cName)
<enrollment gets updated here>
WRITE(enrollment)
Commit
```

Transactions: **Isolation**

- Allow concurrent database access (i.e. operations may be interleaved) but execution must be equivalent to some sequential (serial) order of all transactions.
- Avoid Lost update, inconsistent reads, and repeatable read

Transactions: **Consistency**

```
Create Table Test
( x int,
  y int ,
  check (y = x)
)
```

X	Y
5	5
6	6

```
Update Test set x = x*2
Update Test set y = y*2
```

Transactions: Consistency

```
Create Table Test
( x int,
  y int ,
  check (y >= x)
)
```

```
Update Test set y = x+y;
Update Test set x = x+y;
```

X	Y
5	5
6	6

Transactions: Durability (persistence)

```
Begin Transaction
READ(enrollment, cName)
enrollment gets updated here
WRITE(enrollment)
Commit
```

Transactions: Simplified Model

- Transactions is a list of actions on database object O.

- The actions are:
  - Reads: R(O): reads object O and transfers its value to a variable O in a buffer in main memory belonging to the transaction that executed the read.
  - Writes: W(O): transfers the value of variable O in the main memory buffer of the transaction that executed the write to the data item O in the database.

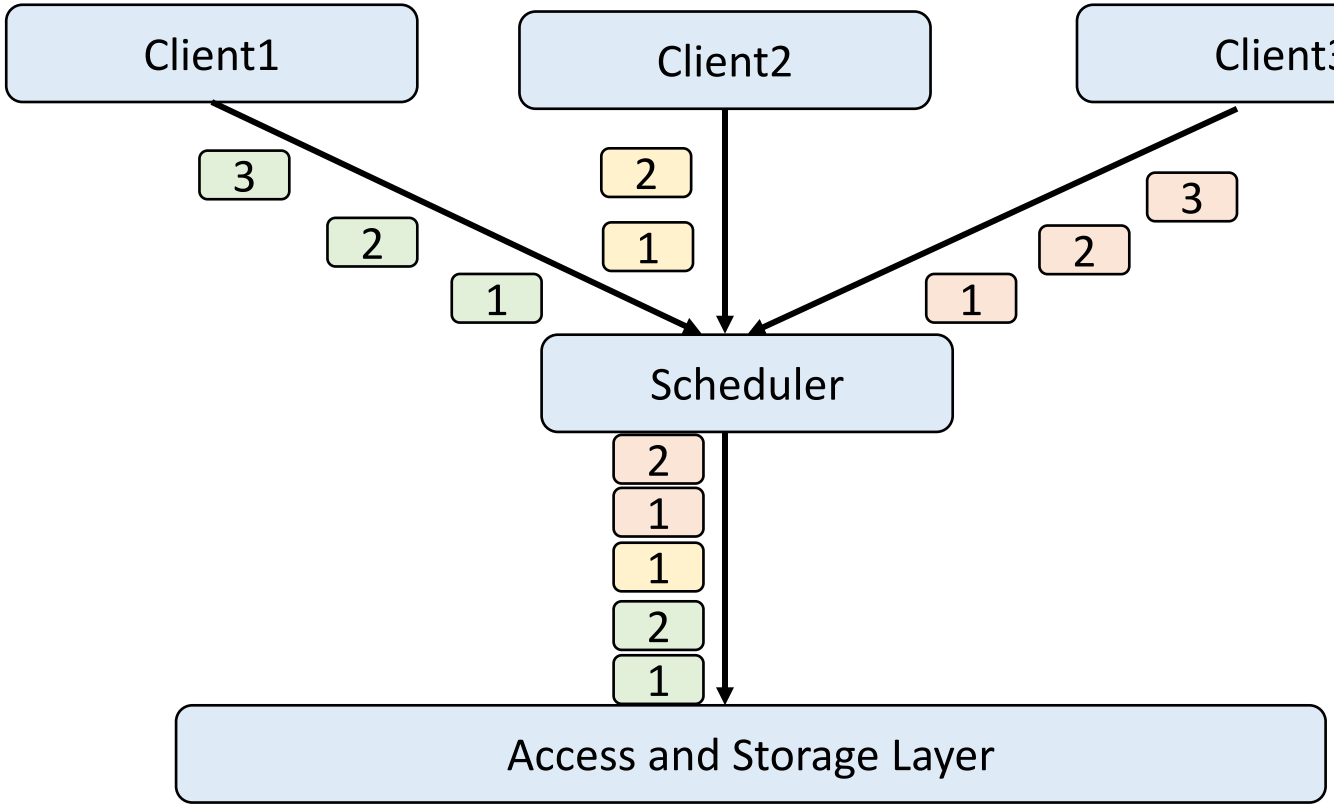
➢Writes are not immediately reflected on disk.

- Transactions end with **Commit** or **Abort**.
  - Sometimes omitted if not relevant.

- Example Txn: T_1: R(O), R(P), W(O), W(M), Commit

Scheduler

- The scheduler decides the execution order of concurrent database access



Schedules

- A schedule is a list of actions from a set of transactions.
    - A plan how to execute transactions.

- In a schedule, the order in which 2 actions of a transaction T appear must be the same order as they appear in the description of T.

Transaction Schedule

- T_1: R(V) W(V)
- T_2: R(Y) W(Y)

- S_1: R(V) R(Y) W(Y) W(V)
- S_2: W(V) R(Y) W(Y) R(V)

Transaction Schedule

- T_1: R(V) W(V)
- T_2: R(Y) W(Y)

- S_1: R(V) R(Y) W(Y) W(V)
- S_2: W(V) R(Y) W(Y) R(V)

Serial Schedule

- A schedule is **serial** if the actions of the different transactions **are not interleaved**; they are executed one after the other.

- T_1: R(V) W(V)
- T_2: R(Y) W(Y)

- S_1: R(Y) W(Y) R(V) W(V)

Serializable Schedule

- A schedule is serializable if **its effect on the database** is the same as that of *some* serial schedule.

- We usually only want to allow serializable schedules.
  - Why?

Serializable Schedule

- A schedule is serializable if **its effect on the database** is the same as that of *some* serial schedule.

- We usually only want to allow serializable schedules.
  - Why?
  - Then the transactions appear to be in isolation.
  - ➢No concurrency anomalies

Conflicts

- Two actions in a schedule conflict if they:
  - Are from different transactions,
  - Involve the same data item, and
  - One of the actions is a write.

- Example

T_1:	R(Y)	W(Y)			W(X)
T_2:			R(Y)	W(Z)	

- Data items: X, Y, Z
  - Y involves write.

Conflicts

- Two actions in a schedule conflict if they:
  - Are from different transactions,
  - Involve the same data item, and
  - One of the actions is a write.

- Example: Which actions conflict?

T_1:	R(Y)	W(Y)			W(X)	Co/Ab
T_2:			R(Y)	W(Z)		

Conflicts

- Two actions in a schedule conflict if they:
  - Are from different transactions,
  - Involve the same data item, and
  - One of the actions is a write.

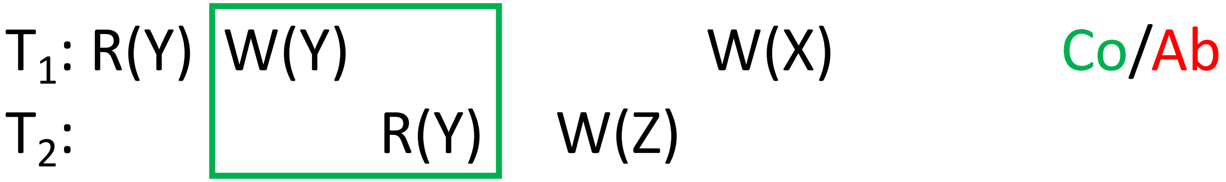
- Example: Which actions conflict?

T_1:	$\boxed{R(Y)}$	W(Y)			W(X)	Co/Ab
T_2:			$\boxed{R(Y)}$	W(Z)		

Conflicts

- Two actions in a schedule conflict if they:
  - Are from different transactions,
  - Involve the same data item, and
  - One of the actions is a write.

- Example: Which actions conflict?



T_1:	R(Y)	W(Y)			W(X)	Co/Ab
T_2:			R(Y)	W(Z)		

Types of Conflicts

- write read (WR)
- read write (RW)
- write write (WW)

➢**Conflicts cause a schedule to be not serializable.**

WR conflict

- There is WR conflict between T_1 and T_2 if there is an item Y which T_1 writes and afterwards T_2 reads Y.

- If T_1 is not committed, then this is a **dirty read** by T_2

- Example: Find all WR conflicts in the following schedule

• T_1			W(Y)		
• T_2	R(V)			R(Y)	W(Z)
• T_3		W(V)			

WR conflict

- There is WR conflict between T_1 and T_2 if there is an item Y which T_1 writes and afterwards T_2 reads Y.

- If T_1 is not committed, then this is a **dirty read** by T_2

- Example: Find all WR conflicts in the following schedule
- T_1 W(Y)
- T_2 R(V) R(Y) W(Z)
- T_3 W(V)

RW Conflict

- There is RW conflict between T_1 and T_2 if there is an item Y which T_1 reads and afterwards T_2 writes Y.

- This read becomes an **unrepeatable** read.

- Example: Find all RW conflicts in the following schedule

T_1			W(Y)			
T_2	R(V)			R(Y)	W(Z)	R(V)
T_3		W(V)				

RW Conflict

- There is RW conflict between T_1 and T_2 if there is an item Y which T_1 reads and afterwards T_2 writes Y.
- This read becomes an **unrepeatable** read.
- Example: Find all RW conflicts in the following schedule
- T_1 W(Y)
- T_2 R(V) R(Y) W(Z) R(V)
- T_3 W(V)

WW Conflict

- There is WW conflict between T_1 and T_2 if there is an item Y which T_1 writes and afterwards T_2 writes Y.

- This is a **lost update** problem as write becomes overwritten

- Example: Find all WW conflicts in the following schedule

• T_1	W(Y)					
• T_2		W(V)		R(Z)	W(Y)	W(Z)
• T_3			W(V)			

WW Conflict

- There is WW conflict between T_1 and T_2 if there is an item Y which T_1 writes and afterwards T_2 writes Y.

- This is a **lost update** problem as write becomes overwritten

- Example: Find all WW conflicts in the following schedule
- T_1 W(Y)
- T_2 W(V) R(Z) W(Y) W(Z)
- T_3 W(V)

Swapping Actions

- We can swap actions (of different transactions) without changing the outcome if the actions are non-conflicting.

• T_1		R(Y)		
• T_2	R(V)		R(Y)	W(Y)

- Swap Reads of same item

• T_1			R(Y)	
• T_2	R(V)	R(Y)		W(Y)

- Swap Reads of of different item

• T_1	R(Y)			
• T_2		R(V)	R(Y)	W(Y)

Conflict Equivalent Schedules--Definition

- Two schedules are **conflict equivalent** if they can be transformed into each other by a sequence of **swaps of non-conflicting, adjacent actions.**

- ➢In other words, the types and number of conflicts in the two schedules are the same.

Example: Conflict-Equivalent Schedules

s

T_1:	W(V)		R(V)	W(V)
T_2:		R(V)		

---

s

T_1:	W(V)	R(V)		W(V)
T_2:			R(V)	

---

s

T_1:		W(V)	R(V)	W(V)
T_2:	R(V)			

Example: Conflict-Equivalent Schedules

S	T_1:	W(V)		R(V)	W(V)
	T_2:		R(V)		

S	T_1:	W(V)	R(V)		W(V)
	T_2:			R(V)	

S	T_1:		W(V)	R(V)	W(V)
	T_2:	R(V)			

Conflict Serializable Schedules

- A schedule is conflict-serializable if it is conflict-equivalent to some serial schedule.

➢ Conflict-serializable schedules are serializable (but not necessarily vice-versa)

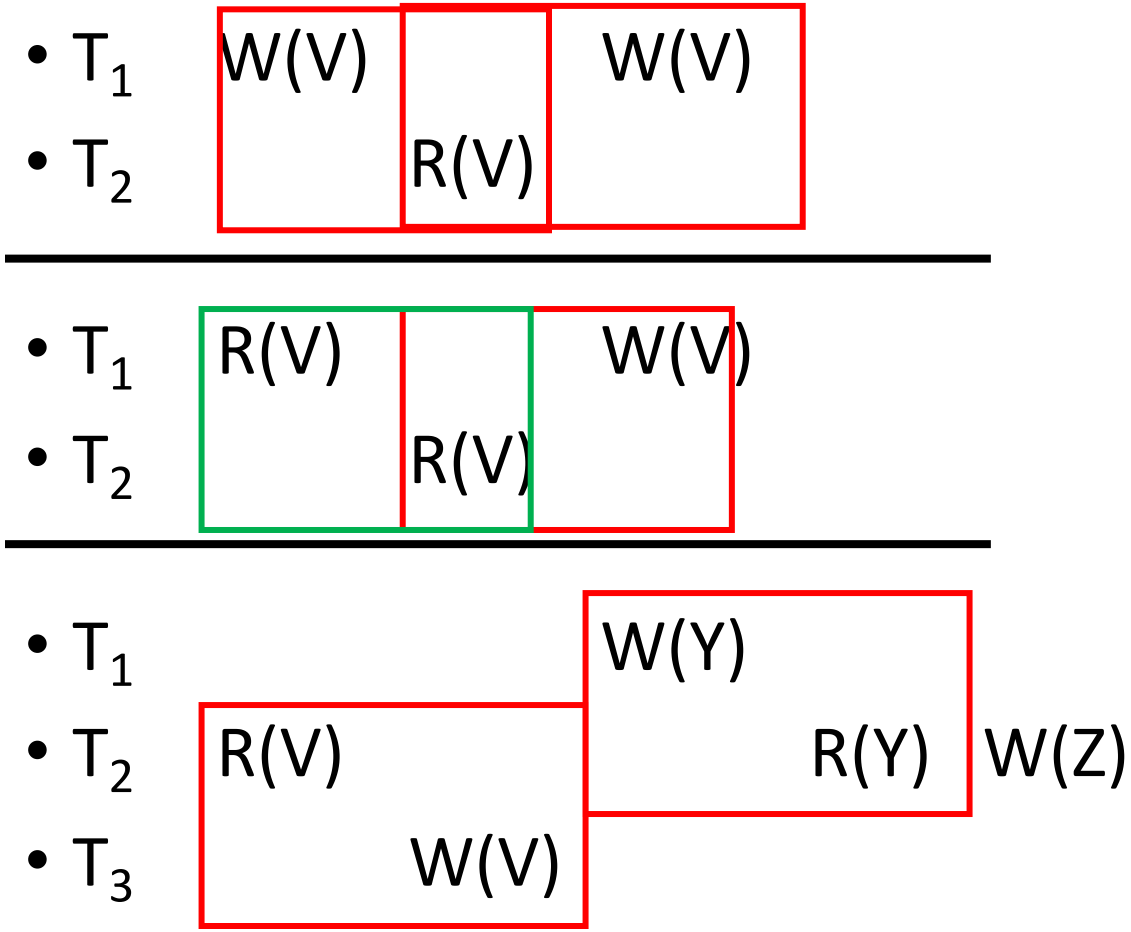
Example: Conflict Serializable

• T_1	W(V)		W(V)		
• T_2		R(V)			

• T_1	R(V)		W(V)		
• T_2		R(V)			

• T_1			W(Y)		
• T_2	R(V)			R(Y)	W(Z)
• T_3		W(V)			

Example: Conflict Serializable



• T_1	W(V)		W(V)		
• T_2		R(V)			

• T_1	R(V)		W(V)		
• T_2		R(V)			

• T_1			W(Y)		
• T_2	R(V)			R(Y)	W(Z)
• T_3		W(V)			

• T_1	W(V)		W(V)		
• T_2		R(V)			

• T_1		R(V)	W(V)		
• T_2	R(V)				

• T_1	W(Y)				
• T_2		R(V)	R(Y)	W(Z)	
• T_3					W(V)

Checking Conflict Serializability

- Given a schedule create precedence graph:
    - Graph has a node for each transaction
    - There is an edge from T_1 to T_2 if there is a conflicting action between T_1 and T_2 in which T_1 occurs first.

    - No need to repeat edges if multiple conflicting actions between T_1 and T_2 in which T_1 occurs first.

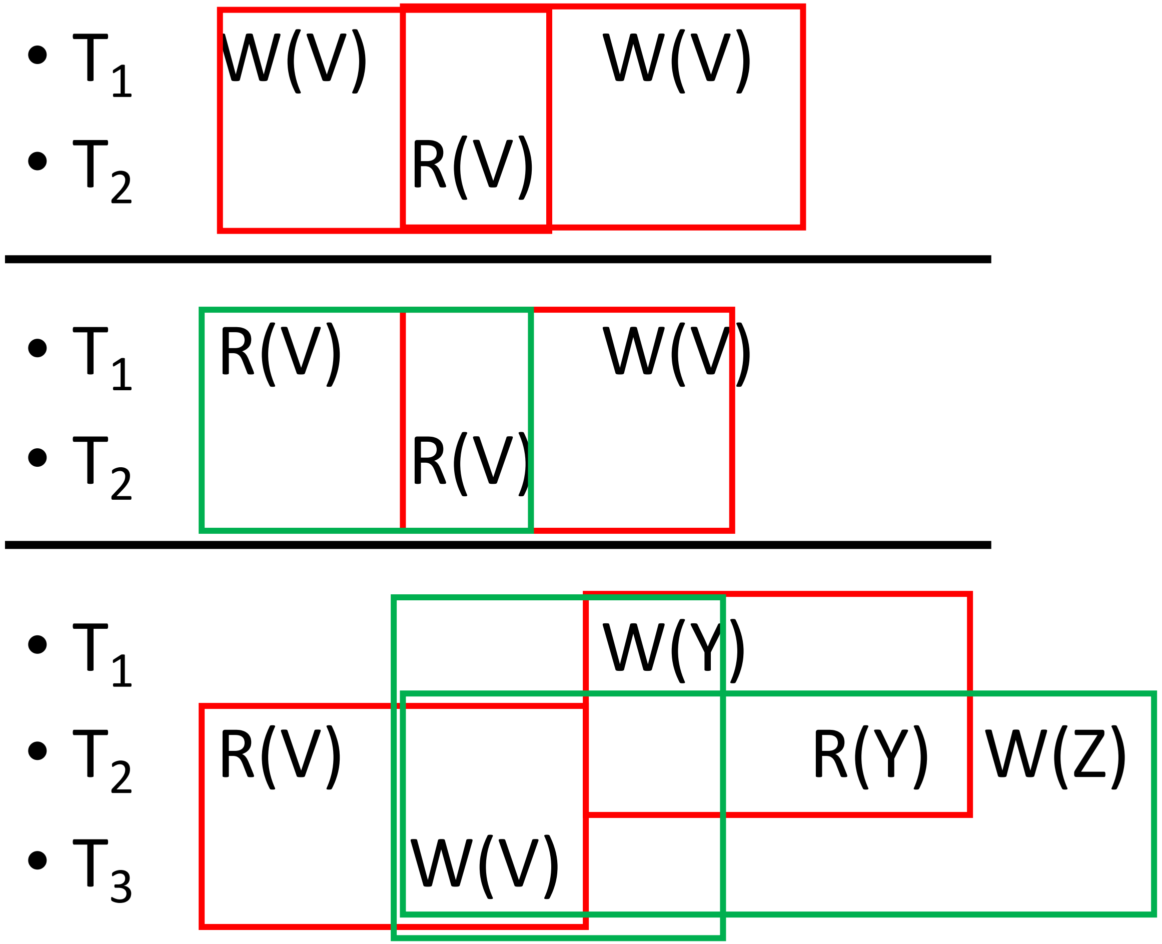
Example: Checking Conflict Serializability

T_1	W(V)		W(V)		
T_2		R(V)			

T_1	R(V)		W(V)		
T_2		R(V)			

T_1		W(Y)			
T_2	R(V)			R(Y)	W(Z)
T_3			W(V)		

Example: Conflict Serializable



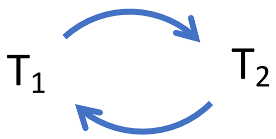
T_1	W(V)		W(V)
T_2		R(V)	

T_1	R(V)		W(V)
T_2		R(V)	

T_1			W(Y)		
T_2	R(V)			R(Y)	W(Z)
T_3		W(V)			

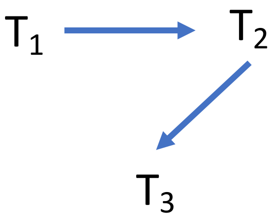
Example: Checking Conflict Serializability

T_1	W(V)		W(V)		
T_2		R(V)			



T_1	R(V)		W(V)		
T_2		R(V)			

T_1		W(Y)			
T_2	R(V)			R(Y)	W(Z)
T_3			W(V)		

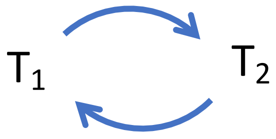


Checking Conflict Serializability

- A schedule is conflict serializable if and only if there is no cycle in the precedence graph

Example: Checking Conflict Serializability

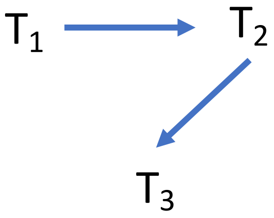
• T_1	W(V)		W(V)		
• T_2		R(V)			



• T_1	R(V)		W(V)		
• T_2		R(V)			

CZ

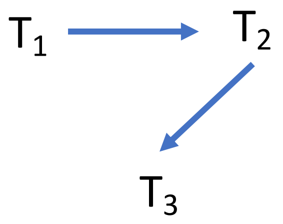
• T_1		W(Y)			
• T_2	R(V)			R(Y)	W(Z)
• T_3			W(V)		



CZ

Checking Conflict Serializability

- If the precedence graph has no cycle, then an equivalent serial schedule is obtained by the **topological sort** of the precedence graph
- T_1 W(Y)
- T_2 R(V) R(Y) W(Z)
- T_3 W(V)



There is an edge from T_1 to T_2 therefore T_1 must come before T_2.

There is an edge from T_2 to T_3 therefore T_2 must come before T_3.

Equivalent T_3 must follow T_2 which must follow T_1

- T_1 W(Y)
- T_2 R(V) R(Y) W(Z)
- T_3 W(V)

Example

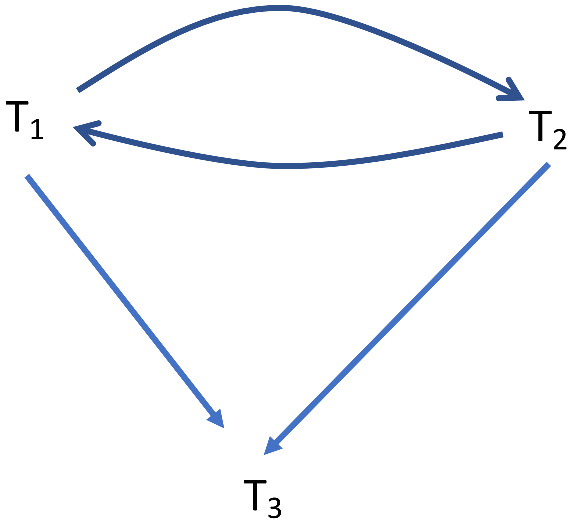
- Is the following schedule conflict serializable?

• T_1	R(V)		W(V)	
• T_2		W(V)		
• T_3				W(V)

Example

- Is the following schedule conflict serializable?

• T_1	R(V)		W(V)	
• T_2		W(V)		
• T_3				W(V)



- Not conflict serializable!
- However, the schedule is view-serializable: T_1, T_2, T_3!
- The writes of T_1 and T_2 are blind writes.

Example

- Is the following schedule conflict serializable?

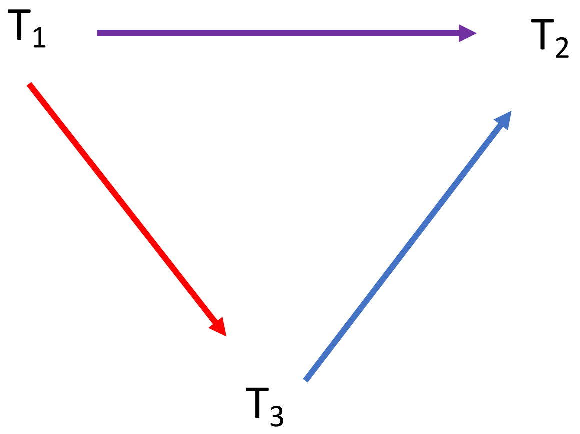
• T_1	R(V)			R(Z)		R(Y)	
• T_2		R(Y)			W(V)		
• T_3			W(V)				W(Z)

Example

- Is the following schedule conflict serializable?

• T_1	R(V)			R(Z)		R(Y)	
• T_2		R(Y)			W(V)		
• T_3			W(V)				W(Z)

• T_1	R(V)	R(Z)	R(Y)				
• T_2						R(Y)	W(V)
• T_3				W(V)	W(Z)		



Example

• T_1			W(B)		R(D)			
• T_2	R(B)	W(C)					R(A)	
• T_3				W(B)		R(C)		W(C)

- $T_1 \rightarrow T_2$
- $T_1 \rightarrow T_3$
- $T_2 \rightarrow T_1$
- $T_2 \rightarrow T_3$
- $T_3 \rightarrow T_1$
- $T_3 \rightarrow T_2$

Example

- T_1 W(B) R(D)
- T_2 R(B) W(C) R(A)
- T_3 W(B) R(C) W(C)

- ~~$T_1 \rightarrow T_2$~~
- $T_1 \rightarrow T_3$
- $T_2 \rightarrow T_1$
- $T_2 \rightarrow T_3$
- ~~$T_3 \rightarrow T_1$~~
- ~~$T_3 \rightarrow T_2$~~

Example

- T_1 R(D) W(B)
- T_2 R(C)
- T_3 W(B) R(A) W(C) W(A) R(C)

- $T_1 \rightarrow T_2$
- $T_1 \rightarrow T_3$
- $T_2 \rightarrow T_1$
- $T_2 \rightarrow T_3$
- $T_3 \rightarrow T_1$
- $T_3 \rightarrow T_2$

Schedule with Aborted Transactions

- When a transaction aborts, the scheduler must undo its updates
- But some of its updates may have affected other transactions!

- T_1 R(V) W(V) Ab
- T_2 R(V) W(V) R(Y) W(Y) Co

- Cannot abort T_1 because cannot undo T_2

Recoverable Schedules

- A schedule is **recoverable** if:
  - It is conflict-serializable, and
  - Whenever a transaction T commits, all transactions that **have written elements** read by T have **already committed**

Recoverable Schedules

- A schedule is **recoverable** if:
  - It is conflict-serializable, and
  - Whenever a transaction T commits, all transactions that **have written elements** read by T have **already committed**

- T_1 R(V) W(V) Ab
- T_2 R(V) W(V) R(Y) W(Y) Co

Non-recoverable!

- T_1 R(V) W(V) Co
- T_2 R(V) W(V) R(Y) W(Y) Co

Recoverable!

Example

- T_1 R(X) W(X) Ab
- T_2 R(X) W(X) R(Y) W(Y)
- T_1 R(Y) W(Y) R(Z) W(Z)
- T_2 R(Z) W(Z) R(A) W(A)

Cascading Aborts

- If a transaction T aborts, then we need to abort any other transaction T’ that has read an element written by T.

- A schedule **avoids cascading aborts** if whenever a transaction reads an element, the transaction that has **last written** it has **already committed**.

Example

- T_1 R(X) W(X) Co
- T_2 R(X) W(X) R(Y) W(Y) Co
- T_1 R(Y) W(Y) R(Z) W(Z) Co
- T_2 R(Z) W(Z) R(A) W(A)

Serializability and Recoverability

- Serializability
  - Serial
  - Serializable
  - Conflict serializable
  - Conflict serializable is serializable, but not vice-versa
- Recoverability
  - Recoverable
  - Avoids cascading deletes

Scheduler

- The scheduler:
  - Module that schedules the transaction’s actions, ensuring serializability
- Two main approaches
  - Pessimistic: locks
  - Optimistic: timestamps, multi-version, validation

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serializability after it has executed is a little too late!
- Need conc. control protocols that assure serializability

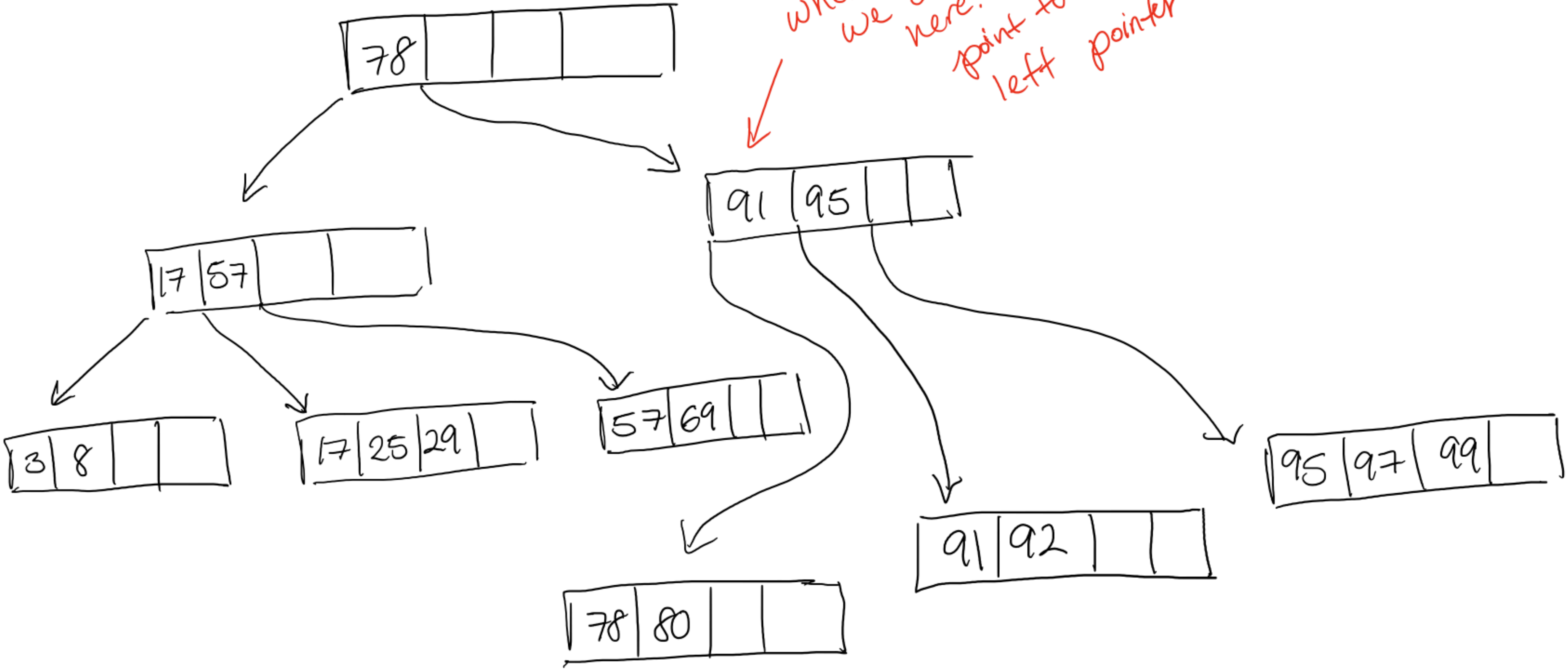
Weak Levels of Consistency

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

Insertions

97

need to split Root!



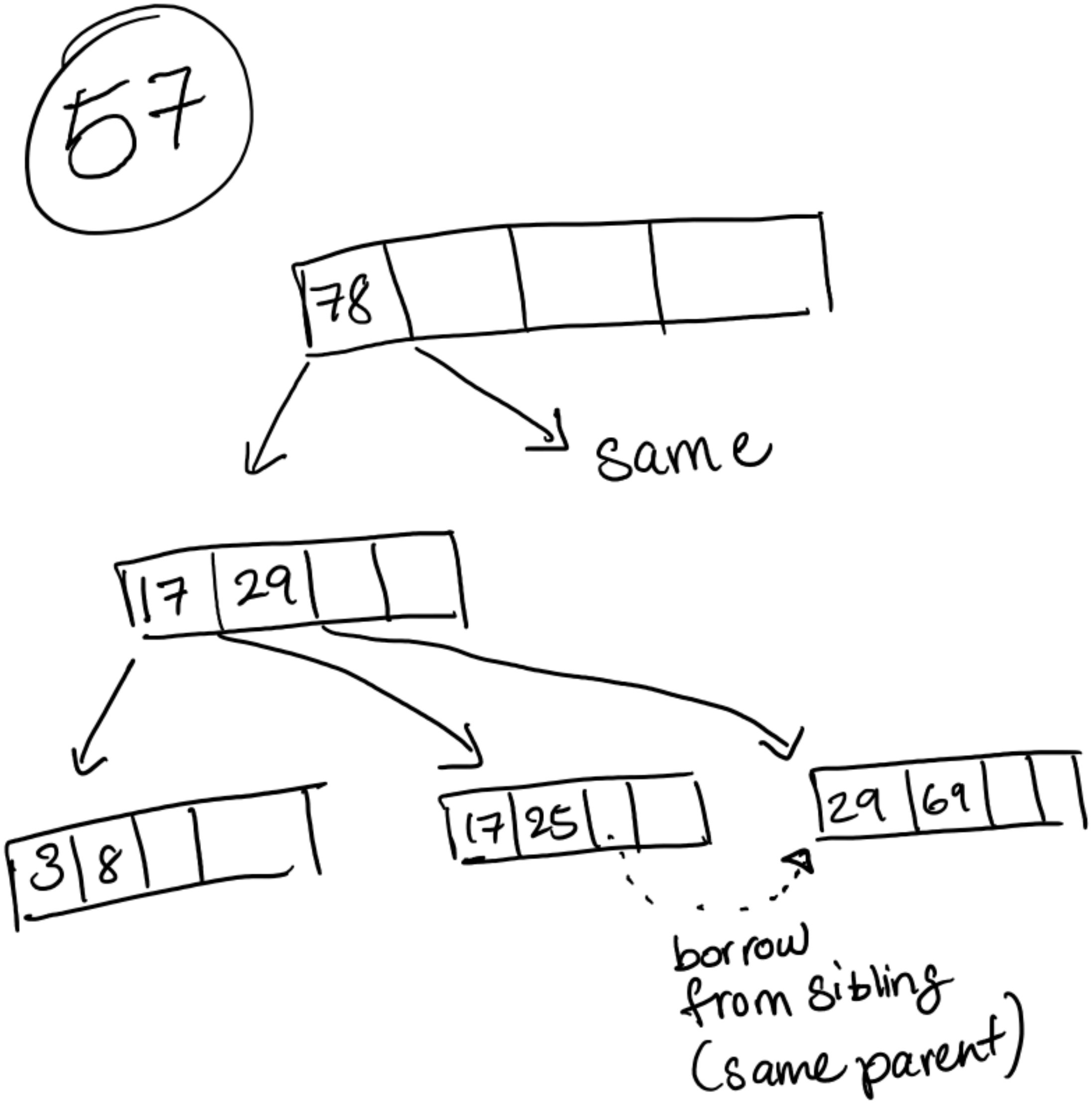
when splitting Root Node
we dont move 78
here. we can simply
point to it using
left pointer

Problem 1:
B+ tree insertion and deletion

- Now delete all nodes in the following order:

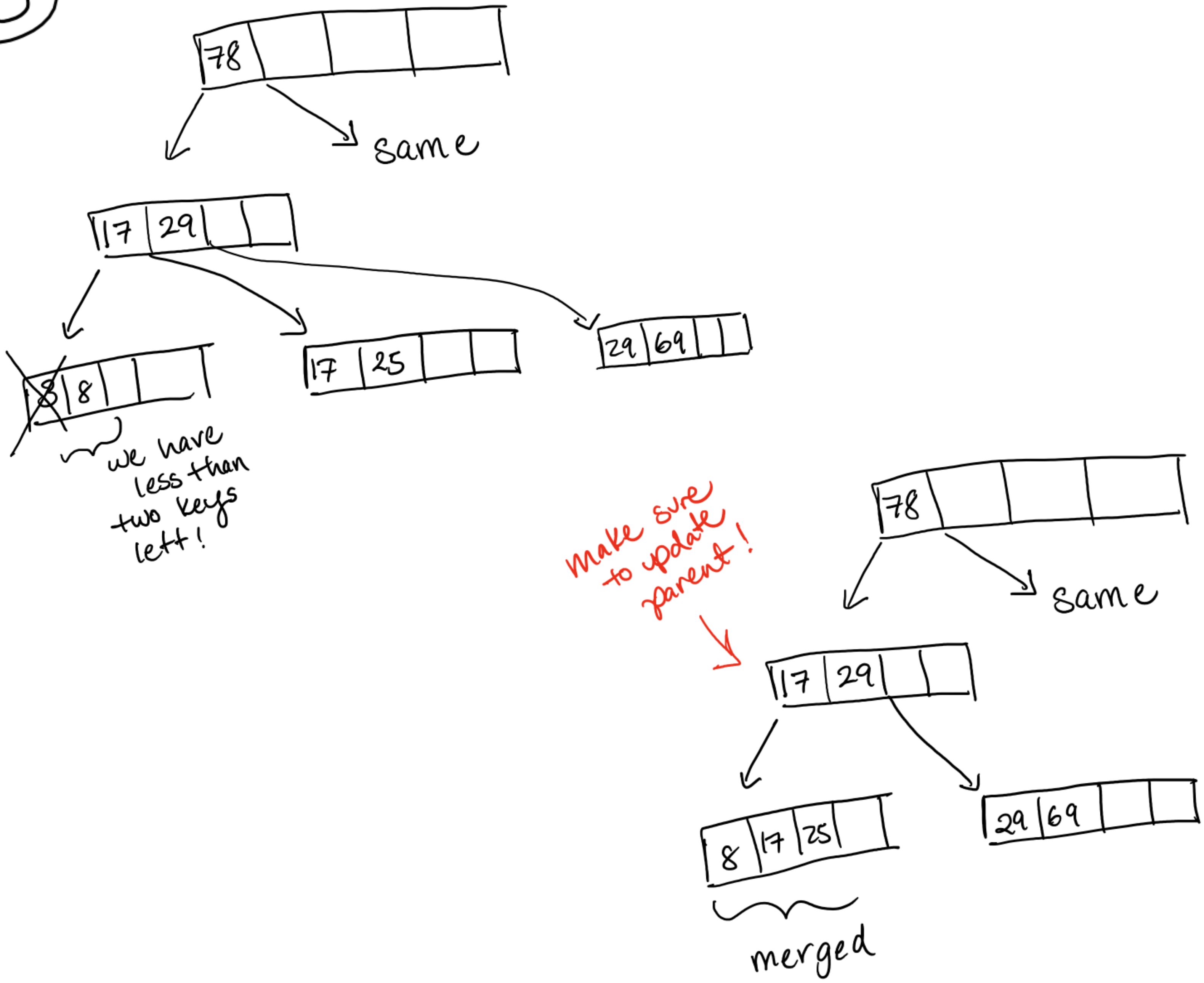
57, 3, 99, 29, 17, 25, 95, 8, 78, 92, 69, 97, 91

Deletions



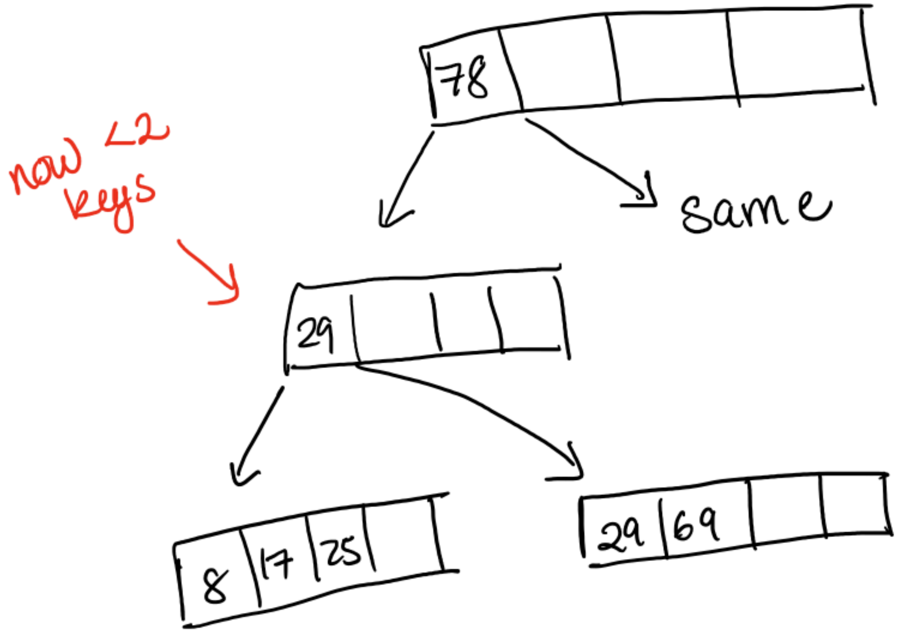
③

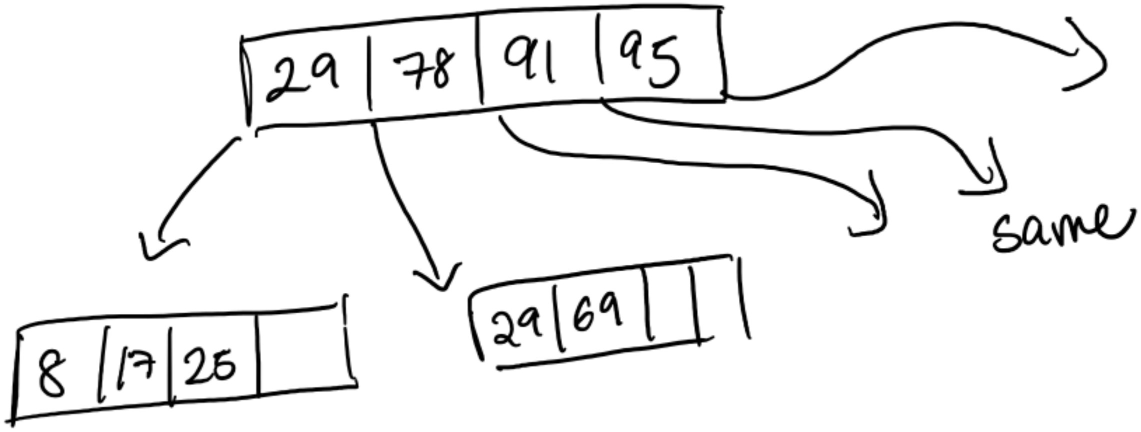
Deletions


78
same
17
29
~~8~~
8
17
25
29
69
we have less than two keys left!
make sure to update parent!
78
same
17
29
8
17
25
29
69
merged

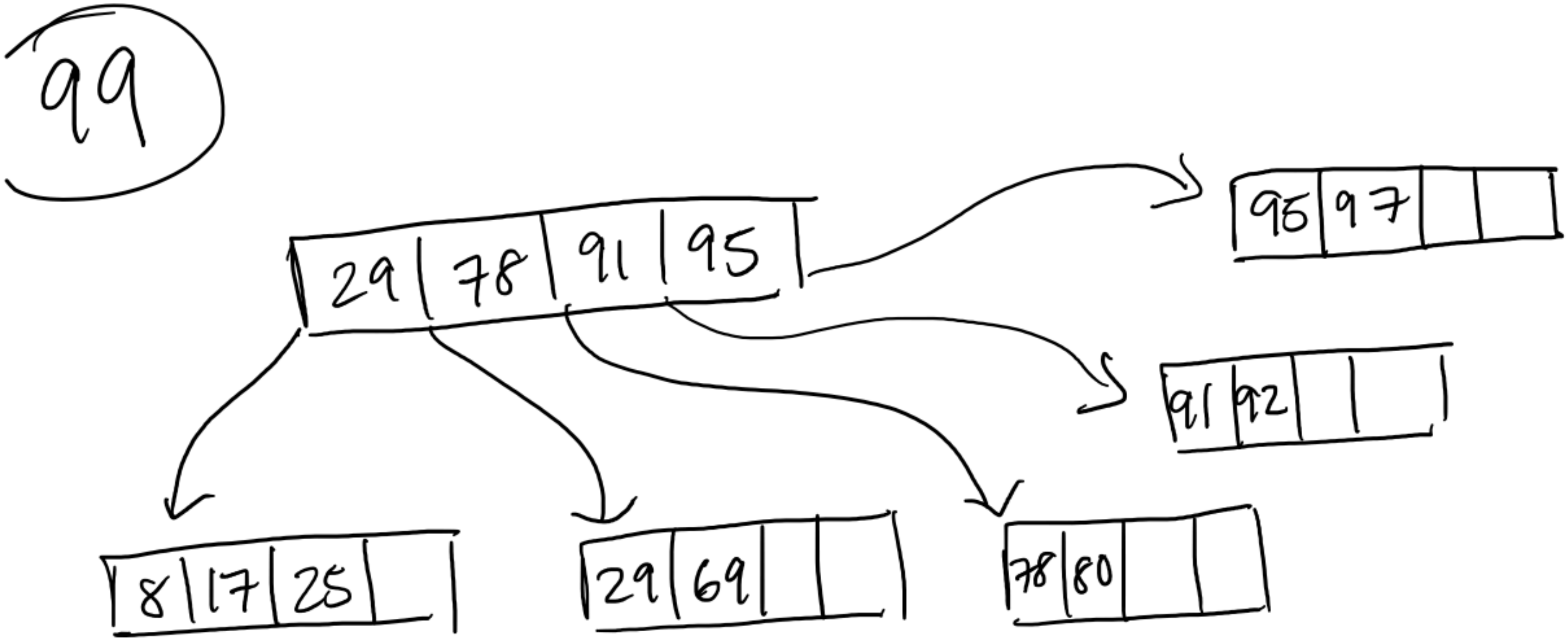
③

Deletions (continued for 3)



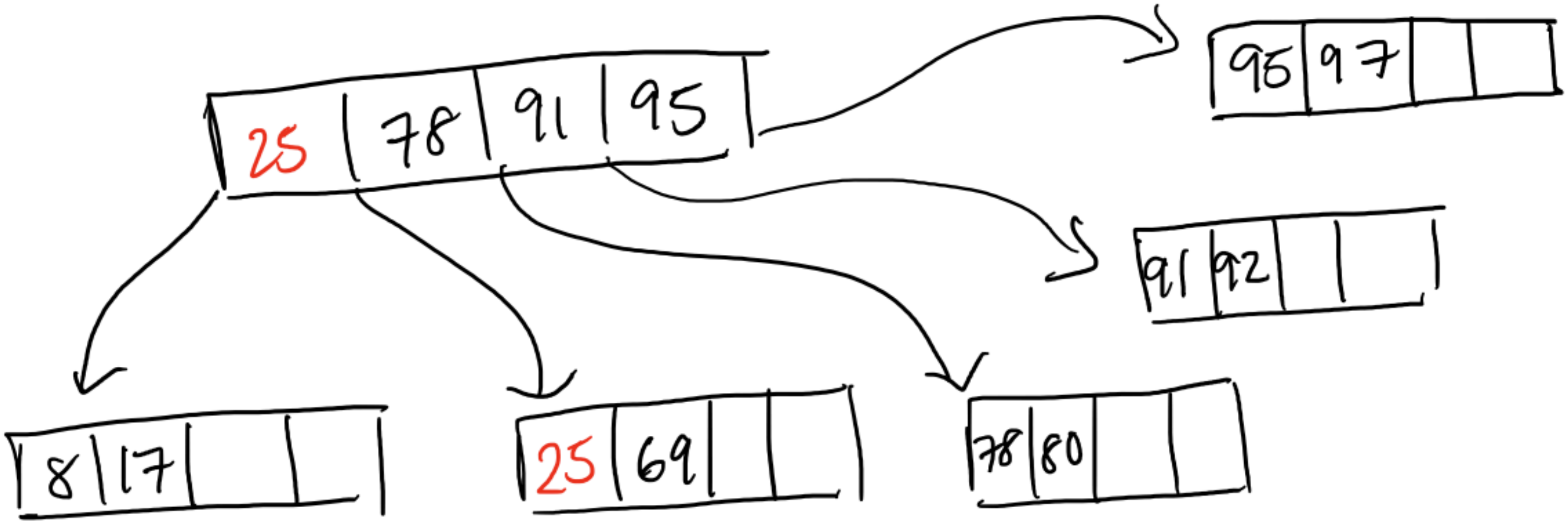


Deletions



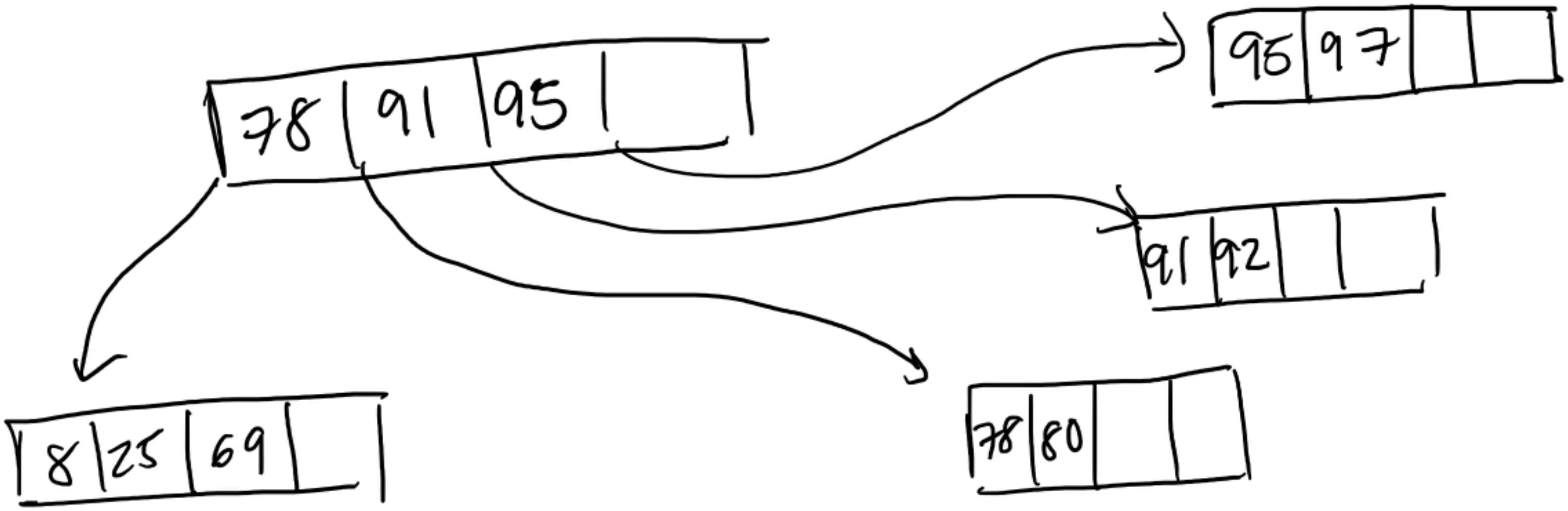
Deletions

㉙



Deletions

⑰



when merging,
delete seperating key
in parent!