

CSE 444: Database Internals

Lectures 13 Transaction Schedules

CSE 444 - Spring 2015

1

Announcements

- Lab 2 is due TODAY
 - Lab 3 will be released tomorrow, part 1 due next Monday
- HW4 is due on Wednesday
 - HW3 has been released, due next week
- 544M: Paper 3 reading is due TODAY
 - Papers 4 and 5 are due on same day in a few weeks
 - Write-up should be 2 to 3 pages long since 2 papers

CSE 444 - Spring 2015

2

Motivating Example

Client 1:
`UPDATE Budget`
`SET money=money-100`
`WHERE pid = 1`

`UPDATE Budget`
`SET money=money+60`
`WHERE pid = 2`

`UPDATE Budget`
`SET money=money+40`
`WHERE pid = 3`

Client 2:
`SELECT sum(money)`
`FROM Budget`

Would like to treat
each group of
instructions as a unit

CSE 444 - Spring 2015

3

Transaction

Definition: a transaction is a sequence of updates to the database with the property that either all complete, or none completes (all-or-nothing).

`BEGIN TRANSACTION`
[SQL statements]
`COMMIT` or `ROLLBACK (=ABORT)`

May be omitted if
autocommit is off:
first SQL query
starts txn

In ad-hoc SQL: each statement = one transaction
This is referred to as autocommit

CSE 444 - Spring 2015

4

Motivating Example

`START TRANSACTION`
`UPDATE Budget`
`SET money=money-100`
`WHERE pid = 1`

`UPDATE Budget`
`SET money=money+60`
`WHERE pid = 2`

`UPDATE Budget`
`SET money=money+40`
`WHERE pid = 3`
`COMMIT (or ROLLBACK)`

`SELECT sum(money)`
`FROM Budget`

With autocommit and
without `START TRANSACTION`,
each SQL command
is a transaction

CSE 444 - Spring 2015

5

Transactions

- Major component of database systems
- Critical for most applications; arguably more so than SQL
- Turing awards to database researchers:
 - Charles Bachman 1973
 - Edgar Codd 1981 for inventing relational dbs
 - Jim Gray 1998 for inventing transactions
 - Mike Stonebraker 2015 for INGRES and Postgres
 - And many other ideas after that

CSE 444 - Spring 2015

6

ROLLBACK

- If the app gets to a place where it can't complete the transaction successfully, it can execute **ROLLBACK**
- This causes the system to "abort" the transaction
 - Database returns to a state without any of the changes made by the transaction
- Several reasons: user, application, system

CSE 444 - Spring 2015

7

ACID Properties

- **Atomicity**: Either all changes performed by transaction occur or none occurs
- **Consistency**: A transaction as a whole does not violate integrity constraints
- **Isolation**: Transactions appear to execute one after the other in sequence
- **Durability**: If a transaction commits, its changes will survive failures

CSE 444 - Spring 2015

8

What Could Go Wrong?

Why is it hard to provide ACID properties?

- **Concurrent** operations
 - Isolation problems
 - We saw one example earlier
- **Failures** can occur at any time
 - Atomicity and durability problems
 - Later lectures
- Transaction may need to **abort**

CSE 444 - Spring 2015

9

Different Types of Problems

```
Client 1: INSERT INTO SmallProduct(name, price)
         SELECT pname, price
         FROM Product
         WHERE price <= 0.99

         DELETE Product
         WHERE price <= 0.99

Client 2: SELECT count(*)
         FROM Product

         SELECT count(*)
         FROM SmallProduct
```

What could go wrong ? Inconsistent reads

CSE 444 - Spring 2015

10

Different Types of Problems

```
Client 1:
UPDATE Product
SET Price = Price - 1.99
WHERE pname = 'Gizmo'

Client 2:
UPDATE Product
SET Price = Price*0.5
WHERE pname='Gizmo'
```

What could go wrong ? Lost update

CSE 444 - Spring 2015

11

Different Types of Problems

```
Client 1: UPDATE SET Account.amount = 1000000000
         WHERE Account.number = 'my-account'

Client 2: SELECT Account.amount
         FROM Account
         WHERE Account.number = 'my-account'
```

Aborted by system

What could go wrong ? Dirty reads

CSE 444 - Spring 2015

12

Types of Problems: Summary

- **Concurrent execution problems**
 - **Write-read conflict: dirty read (includes inconsistent read)**
 - A transaction reads a value written by another transaction that has not yet committed
 - **Read-write conflict: unrepeatable read**
 - A transaction reads the value of the same object twice. Another transaction modifies that value in between the two reads
 - **Write-write conflict: lost update**
 - Two transactions update the value of the same object. The second one to write the value overwrite the first change
- **Failure problems**
 - DBMS can crash in the middle of a series of updates
 - Can leave the database in an inconsistent state

CSE 444 - Spring 2015

13

Terminology Needed For Lab 3 Buffer Manager Policies

- **STEAL or NO-STEAL**
 - Can an update made by an uncommitted transaction overwrite the most recent committed value of a data item on disk?
- **FORCE or NO-FORCE**
 - Should all updates of a transaction be forced to disk before the transaction commits?
- Easiest for recovery: NO-STEAL/FORCE (lab 3)
- Highest performance: STEAL/NO-FORCE (lab 5)
- We will get back to this next week

CSE 444 - Spring 2015

14

Outline

- Transactions motivation, definition, properties
- Concurrency Control (the C in ACID)
 - This week
- Recovery from failures (the A in ACID)
 - Next week

CSE 444 - Spring 2015

15

Schedules

A *schedule* is a sequence of interleaved actions from all transactions

CSE 444 - Spring 2015

16

Example

A and B are elements in the database
t and s are variables in tx source code

T1	T2
READ(A, t)	READ(A, s)
t := t+100	s := s*2
WRITE(A, t)	WRITE(A, s)
READ(B, t)	READ(B, s)
t := t+100	s := s*2
WRITE(B, t)	WRITE(B, s)

CSE 444 - Spring 2015

17

A Serial Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B, t)	
	READ(A, s)
	s := s*2
	WRITE(A, s)
	READ(B, s)
	s := s*2
	WRITE(B, s)

CSE 444 - Spring 2015

18

Serializable Schedule

A schedule is **serializable** if it is equivalent to a serial schedule

CSE 444 - Spring 2015

19

A Serializable Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A, s)
	s := s*2
	WRITE(A, s)
READ(B, t)	
t := t+100	
WRITE(B, t)	
	READ(B, s)
	s := s*2
	WRITE(B, s)

This is a **serializable** schedule.
This is NOT a serial schedule

CSE 444 - Spring 2015

20

A Non-Serializable Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A, s)
	s := s*2
	WRITE(A, s)
	READ(B, s)
	s := s*2
	WRITE(B, s)
READ(B, t)	
t := t+100	
WRITE(B, t)	

Why is it non-serializable?

CSE 444 - Spring 2015

21

Serializable Schedules

- The role of the scheduler is to ensure that the schedule is serializable

Q: Why not run only serial schedules ?
I.e. run one transaction after the other ?

CSE 444 - Spring 2015

22

Serializable Schedules

- The role of the scheduler is to ensure that the schedule is serializable

Q: Why not run only serial schedules ?
I.e. run one transaction after the other ?

A: Because of very poor throughput due to disk latency.

Lesson: main memory databases may schedule TXNs serially

CSE 444 - Spring 2015

23

Still Serializable, but...

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A, s)
	s := s + 200
	WRITE(A, s)
	READ(B, s)
	s := s + 200
	WRITE(B, s)
READ(B, t)	
t := t+100	
WRITE(B, t)	

Schedule is serializable because $t=t+100$ and $s=s+200$ commute

...we don't expect the scheduler to schedule this

Ignoring Details

- Assume worst case updates:
 - We never commute actions done by transactions
- As a consequence, we only care about reads and writes
 - Transaction = sequence of R(A)'s and W(A)'s

$T_1: r_1(A); w_1(A); r_1(B); w_1(B)$
 $T_2: r_2(A); w_2(A); r_2(B); w_2(B)$

CSE 444 - Spring 2015

25

Conflicts

- Write-Read – WR
- Read-Write – RW
- Write-Write – WW

CSE 444 - Spring 2015

26

Conflict Serializability

Conflicts:

Two actions by same transaction T_i :

$r_i(X); w_i(Y)$

Two writes by T_i, T_j to same element

$w_i(X); w_j(X)$

Read/write by T_i, T_j to same element

$w_i(X); r_j(X)$

$r_i(X); w_j(X)$

CSE 444 - Spring 2015

27

Conflict Serializability

Definition A schedule is conflict serializable if it can be transformed into a serial schedule by a series of swappings of adjacent non-conflicting actions

- Every conflict-serializable schedule is serializable
- The converse is not true in general

CSE 444 - Spring 2015

28

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

CSE 444 - Spring 2015

29

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$



$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

CSE 444 - Spring 2015

30

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$



$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

CSE 444 - Spring 2015

31

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$



$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B)$



$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

CSE 444 - Spring 2015

32

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$



$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B)$



$r_1(A); w_1(A); r_1(B); r_2(A); w_2(A); w_1(B); r_2(B); w_2(B)$



....

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

CSE 444 - Spring 2015

33

Testing for Conflict-Serializability

Precedence graph:

- A node for each transaction T_i ,
- An edge from T_i to T_j whenever an action in T_i conflicts with, and comes before an action in T_j
- The schedule is serializable iff the precedence graph is acyclic

CSE 444 - Spring 2015

34

Example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

①

②

③

CSE 444 - Spring 2015

35

Example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

①

②

③

This schedule is **conflict-serializable**

CSE 444 - Spring 2015

36

Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

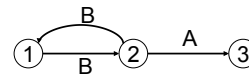


CSE 444 - Spring 2015

37

Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



This schedule is **NOT** conflict-serializable

CSE 444 - Spring 2015

38

View Equivalence

- A serializable schedule need not be conflict serializable, even under the “worst case update” assumption

$w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);$

Is this schedule conflict-serializable ?

CSE 444 - Spring 2015

39

View Equivalence

- A serializable schedule need not be conflict serializable, even under the “worst case update” assumption

$w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);$

Is this schedule conflict-serializable ?

No...

CSE 444 - Spring 2015

40

View Equivalence

- A serializable schedule need not be conflict serializable, even under the “worst case update” assumption

$w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);$

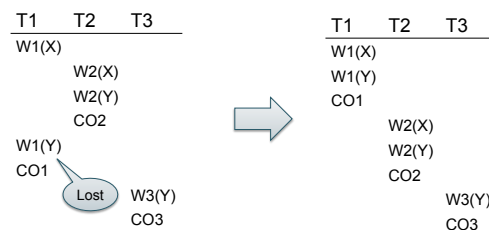
Lost write

$w_1(X); w_1(Y); w_2(X); w_2(Y); w_3(Y);$

Equivalent, but not conflict-equivalent

41

View Equivalence



Serializable, but not conflict serializable

42

View Equivalence

Two schedules S, S' are **view equivalent** if:

- If T reads an **initial value** of A in S, then T reads the **initial value** of A in S'
- If T reads a value of A **written by T'** in S, then T reads a value of A **written by T'** in S'
- If T writes the **final value** of A in S, then T writes the **final value** of A in S'

CSE 444 - Spring 2015

43

View-Serializability

A schedule is **view serializable** if it is view equivalent to a serial schedule

Remark:

- If a schedule is **conflict serializable**, then it is also **view serializable**
- But not vice versa

CSE 444 - Spring 2015

44

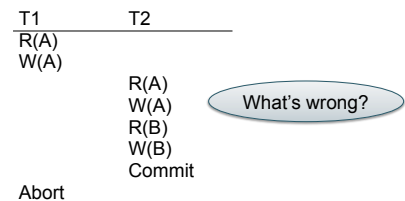
Schedules with Aborted Transactions

- When a transaction aborts, the recovery manager undoes its updates
- But some of its updates may have affected other transactions !

CSE 444 - Spring 2015

45

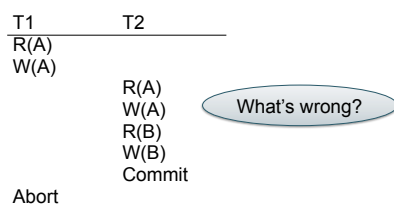
Schedules with Aborted Transactions



CSE 444 - Spring 2015

46

Schedules with Aborted Transactions



Cannot abort T1 because cannot undo T2

Recoverable Schedules

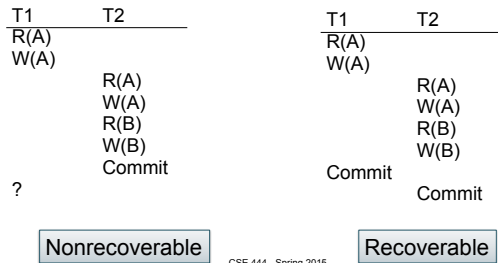
A schedule is **recoverable** if:

- It is conflict-serializable, and
- Whenever a transaction T commits, all transactions who have written elements read by T have already committed

CSE 444 - Spring 2015

48

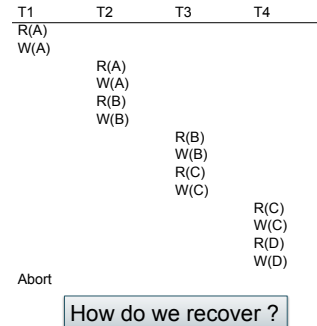
Recoverable Schedules



CSE 444 - Spring 2015

49

Recoverable Schedules



50

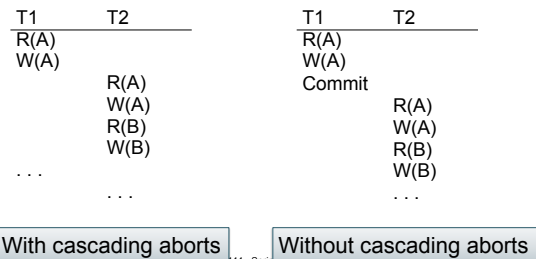
Cascading Aborts

- 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.

CSE 444 - Spring 2015

51

Avoiding Cascading Aborts



CSE 444 - Spring 2015

52

Review of Schedules

Serializability

- Serial
- Serializable
- Conflict serializable
- View serializable

Recoverability

- Recoverable
- Avoids cascading deletes

CSE 444 - Spring 2015

53

Scheduler

- The scheduler:
- Module that schedules the transaction's actions, ensuring serializability
- Two main approaches
 - **Pessimistic**: locks
 - **Optimistic**: time stamps, MV, validation

CSE 444 - Spring 2015

54