CS315: Database Systems
Schedules

Arnab Bhattacharya
arnabb@cse.iitk.ac.in

Computer Science and Engineering,
Indian Institute of Technology, Kanpur
http://web.cse.iitk.ac.in/~cs315/

2nd semester, 2019-20

Tue, Wed 12:00-13:15

Schedule

- A schedule is a *chronological* sequence of instructions from *concurrent* transactions

Schedule

- A schedule is a *chronological* sequence of instructions from *concurrent* transactions
- If a transaction appears in a schedule, *all* its instructions must appear in the schedule
- *Order* of instructions within a transaction must be maintained in the schedule

Schedule

- A schedule is a *chronological* sequence of instructions from *concurrent* transactions
- If a transaction appears in a schedule, *all* its instructions must appear in the schedule
- *Order* of instructions within a transaction must be maintained in the schedule
- A transaction finishing successfully will have *commit* as the last instruction
- A transaction not finishing successfully will have *abort* as the last instruction
- Commit and abort statements may be omitted if obvious

Example

- T1 transfers 50 from A to B and then T2 transfers 10% of A to B
- A serial schedule:
  $r_1(A); A := A - 50; w_1(A); r_1(B); B := B + 50; w_1(B);$
  $r_2(A); t := 0.1A; A := A - t; w_2(A); r_2(B); B := B + t; w_2(B);$

Example

- T1 transfers 50 from A to B and then T2 transfers 10% of A to B
- A serial schedule:
  $r_1(A); A := A - 50; w_1(A); r_1(B); B := B + 50; w_1(B);$
  $r_2(A); t := 0.1A; A := A - t; w_2(A); r_2(B); B := B + t; w_2(B);$
- Another schedule:
  $r_1(A); A := A - 50; w_1(A); r_2(A); t := 0.1A; A := A - t;$
  $w_2(A); r_1(B); B := B + 50; w_1(B); r_2(B); B := B + t; w_2(B);$

Example

- T1 transfers 50 from A to B and then T2 transfers 10% of A to B
- A serial schedule:
  $r_1(A); A := A - 50; w_1(A); r_1(B); B := B + 50; w_1(B);$
  $r_2(A); t := 0.1A; A := A - t; w_2(A); r_2(B); B := B + t; w_2(B);$
- Another schedule:
  $r_1(A); A := A - 50; w_1(A); r_2(A); t := 0.1A; A := A - t;$
  $w_2(A); r_1(B); B := B + 50; w_1(B); r_2(B); B := B + t; w_2(B);$
- This is not a serial schedule but is equivalent

Example

- T1 transfers 50 from A to B and then T2 transfers 10% of A to B
- A serial schedule:
  $r_1(A); A := A - 50; w_1(A); r_1(B); B := B + 50; w_1(B);$
  $r_2(A); t := 0.1A; A := A - t; w_2(A); r_2(B); B := B + t; w_2(B);$
- Another schedule:
  $r_1(A); A := A - 50; w_1(A); r_2(A); t := 0.1A; A := A - t;$
  $w_2(A); r_1(B); B := B + 50; w_1(B); r_2(B); B := B + t; w_2(B);$
- This is not a serial schedule but is equivalent [Effect of both schedules on database is equivalent]
- Yet another schedule: → old value of A
  $r_1(A); A := A - 50; \underline{r_2(A)}; t := 0.1A; A := A - t; \underline{w_2(A)};$
  $\underline{w_1(A)}; r_1(B); B := B + 50; w_1(B); r_2(B); B := B + t; w_2(B);$

  Not equivalent

Example

- T1 transfers 50 from A to B and then T2 transfers 10% of A to B
- A serial schedule:
  $r_1(A); A := A - 50; w_1(A); r_1(B); B := B + 50; w_1(B);$
  $r_2(A); t := 0.1A; A := A - t; w_2(A); r_2(B); B := B + t; w_2(B);$
- Another schedule:
  $r_1(A); A := A - 50; w_1(A); r_2(A); t := 0.1A; A := A - t;$
  $w_2(A); r_1(B); B := B + 50; w_1(B); r_2(B); B := B + t; w_2(B);$
- This is not a serial schedule but is equivalent
- Yet another schedule:
  $r_1(A); A := A - 50; r_2(A); t := 0.1A; A := A - t; w_2(A);$
  $w_1(A); r_1(B); B := B + 50; w_1(B); r_2(B); B := B + t; w_2(B);$
- This is not a serial schedule and is not equivalent either

Serializability

- Each transaction preserves database consistency
- Hence, a serial schedule also does that

Serializability

- Each transaction preserves database consistency
- Hence, a serial schedule also does that
- A schedule is serializable if it is *equivalent* to a serial schedule
- There are different notions of equivalence
  - Conflict serializability
  - View serializability
- Operations other than *read* and *write* are ignored

Serializability

- Each transaction preserves database consistency
- Hence, a serial schedule also does that
- A schedule is serializable if it is *equivalent* to a serial schedule
- There are different notions of equivalence
  - Conflict serializability
  - View serializability
- Operations other than *read* and *write* are ignored
- Instruction I_i of transaction T_i conflicts with I_j of T_j if and only if they access the same data item and at least one of them is a write

Serializability

- Each transaction preserves database consistency
- Hence, a serial schedule also does that
- A schedule is serializable if it is *equivalent* to a serial schedule
- There are different notions of equivalence
  - Conflict serializability
  - View serializability
- Operations other than *read* and *write* are ignored
- Instruction I_i of transaction T_i conflicts with I_j of T_j if and only if they access the same data item and at least one of them is a write
- Intuitively, a conflict enforces a logical temporal order of the instructions
  This can't be violated
- Consequently, if two instructions do not conflict, they can be interchanged
  [placed before or after]

Conflict Serializability

- A schedule S is conflict equivalent to another schedule S' if it can be transformed to S' by a series of swaps of *non-conflicting* instructions

Conflict Serializability

- A schedule S is conflict equivalent to another schedule S' if it can be transformed to S' by a series of swaps of *non-conflicting* instructions
- A schedule S is conflict serializable if it is conflict equivalent to a serial schedule

Conflict Serializability

- A schedule S is conflict equivalent to another schedule S' if it can be transformed to S' by a series of swaps of *non-conflicting* instructions
  if serial ⇒ conflict serializable.
- A schedule S is conflict serializable if it is conflict equivalent to a serial schedule
  View serializable
- A serial schedule is conflict serializable, but not vice versa
- If a schedule is conflict serializable, it is correct in the sense that it preserves database consistency

Example

- $S: r_1(a) w_1(a) r_2(a) w_2(a) r_1(b) w_1(b) r_2(b) w_2(b)$

Example

- $S : r_1(a)w_1(a)r_2(a)\boxed{w_2(a)}r_1(b)\boxed{w_1(b)}r_2(b)w_2(b)$
  is *conflict serializable* as it is conflict equivalent to the serial schedule
  Swapped ⇒ Non-conflicting
  $T_1T_2 : r_1(a)w_1(a)r_1(b)\boxed{w_1(b)}r_2(a)\boxed{w_2(a)}r_2(b)w_2(b)$
  - It is not required to be conflict equivalent to T_2T_1 as well

T_1 written in the order the inst. appear (can't be violated)

$T_1: r_1(a)\, w_1(a)\, r_1(b)\, w_1(b)$

Either T_1T_2 or T_2T_1

$T_2: r_2(a)\, w_2(a)\, r_2(b)\, w_2(b)$

read : Non-conflicting : don't operate on the same data

write : Non-conflicting : — " — " —

Example

- $S : r_1(a)w_1(a)r_2(a)w_2(a)r_1(b)w_1(b)r_2(b)w_2(b)$
  is *conflict serializable* as it is conflict equivalent to the serial schedule
  $T_1 T_2 : r_1(a)w_1(a)r_1(b)w_1(b)r_2(a)w_2(a)r_2(b)w_2(b)$
  - It is not required to be conflict equivalent to $T_2 T_1$ as well
- $r_1(a)w_2(a)w_1(a)$

Example

- $S : r_1(a)w_1(a)r_2(a)w_2(a)r_1(b)w_1(b)r_2(b)w_2(b)$
  is *conflict serializable* as it is conflict equivalent to the serial schedule
  $T_1T_2 : r_1(a)w_1(a)r_1(b)w_1(b)r_2(a)w_2(a)r_2(b)w_2(b)$
  - It is not required to be conflict equivalent to T_2T_1 as well
- $r_1(a)\underline{w_2(a)}\underline{w_1(a)}$ → old a is final
  is *not* conflict serializable as it is not conflict equivalent to either of the two serial schedules T_1T_2 and T_2T_1

$T_1 : r_1(a)w_1(a)$

$T_2 : w_2(a)$

$T_1T_2 : r_1(a)\underline{w_1(a)}\,\underline{w_2(a)}$ (some new a)

⇒ Operate on the same data [one of them is a write]

$T_2T_1 : w_2(a)r_1(a)w_1(a)$ same logic

⇒ Non conflict serializable

View Serializability

- Two schedules are view equivalent if the reads in them get the same "view", i.e., they read the value produced by the same write operation

T_1, T_2 S, S'

1. x $r_1(x)$ $r_2(x)$ initial read [same] transaction

final write [same]

2. x $w_1(x)$ $w_2(x)$

3. x $T_1 \leftarrow T_2$ S

$T_1 \leftarrow T_2$ S'

If T_1 reads value produced by T_2 in S

$\Downarrow$

same should happen in S'

View Serializability

- Two schedules are view equivalent if the reads in them get the same "view", i.e., they read the value produced by the same write operation
- Formally, two schedules S and S' are view equivalent if
  1. For each data item x, if a transaction T reads the initial value of x in S, it reads the same initial value of x in S' as well
  2. For each data item x, if a transaction T writes the final value of x in S, it writes the final value of a in S' as well
  3. If transaction T_i reads the value of data item x produced by write by transaction T_j in S, it must read the value written by T_j in S' as well
- A schedule S is view serializable if it is view equivalent to a serial schedule

Example

- $S : \boxed{r_1(a)} w_1(a) r_2(a) \boxed{w_2(a)} \boxed{r_1(b)} w_1(b) r_2(b) \boxed{w_2(b)}$

$T_1 T_2 : \boxed{r_1(a)} \boxed{w_1(a)} \boxed{r_1(b)} w_1(b) \boxed{r_2(a)} \boxed{w_2(a)} r_2(b) \boxed{w_2(b)}$

a:
- a : Read ~~r(a)~~ T_1 in both
- Write: T_2 in both
- S: NULL ← r_1, w_1 ← r_2 ✓
- S': Null ← r_1, w_1 ← r_2 ✓

b:
- R: T_1 ✓
- W: T_2 ✓
- S: NULL ← r_1, w_1 ← r_2 ✓
- S': NULL ← r_1, w_1 ← r_2 ✓

Example

- $S : r_1(a)w_1(a)r_2(a)w_2(a)r_1(b)w_1(b)r_2(b)w_2(b)$
  is *view serializable* as it is view equivalent to the serial schedule
  $T_1 T_2 : r_1(a)w_1(a)r_1(b)w_1(b)r_2(a)w_2(a)r_2(b)w_2(b)$

Example

- $S : r_1(a)w_1(a)r_2(a)w_2(a)r_1(b)w_1(b)r_2(b)w_2(b)$
  is *view serializable* as it is view equivalent to the serial schedule
  $T_1 T_2 : r_1(a)w_1(a)r_1(b)w_1(b)r_2(a)w_2(a)r_2(b)w_2(b)$
- $r_1(a)w_2(a)w_1(a)w_3(a)$

$T_1 T_2 T_3 : r_1(a) w_1(a) w_2(a) w_3(a)$ } View serializable.

a:

Example

- $S : r_1(a)w_1(a)r_2(a)w_2(a)r_1(b)w_1(b)r_2(b)w_2(b)$
  is *view serializable* as it is view equivalent to the serial schedule
  $T_1 T_2 : r_1(a)w_1(a)r_1(b)w_1(b)r_2(a)w_2(a)r_2(b)w_2(b)$
- $r_1(a)w_2(a)w_1(a)w_3(a)$
  is *view serializable* as it is view equivalent to the serial schedule
  $T_1 T_2 T_3 : r_1(a)w_1(a)w_2(a)w_3(a)$

Example

- $S : r_1(a)w_1(a)r_2(a)w_2(a)r_1(b)w_1(b)r_2(b)w_2(b)$
  is *view serializable* as it is view equivalent to the serial schedule
  $T_1 T_2 : r_1(a)w_1(a)r_1(b)w_1(b)r_2(a)w_2(a)r_2(b)w_2(b)$
- $r_1(a)w_2(a)w_1(a)w_3(a)$
  is *view serializable* as it is view equivalent to the serial schedule
  $T_1 T_2 T_3 : r_1(a)w_1(a)w_2(a)w_3(a)$
- $r_1(a)w_2(a)w_1(a)$

$T_1 T_2$: $r_1(a)$ $w_1(a)$ $w_2(a)$
$T_2 T_1$: $w_2(a)$ $r_1(a)$ $w_1(a)$

Not View serializable. ✗ 3. T_1 doesn't read what T_2 writes

Example

- $S : r_1(a)w_1(a)r_2(a)w_2(a)r_1(b)w_1(b)r_2(b)w_2(b)$
  is *view serializable* as it is view equivalent to the serial schedule
  $T_1T_2 : r_1(a)w_1(a)r_1(b)w_1(b)r_2(a)w_2(a)r_2(b)w_2(b)$
- $r_1(a)w_2(a)w_1(a)w_3(a)$
  is *view serializable* as it is view equivalent to the serial schedule
  $T_1T_2T_3 : r_1(a)w_1(a)w_2(a)w_3(a)$
- $r_1(a)w_2(a)w_1(a)$
  is *not* view serializable as it is not view equivalent to either of the two serial schedules T_1T_2 and T_2T_1

Relationship between View and Conflict Serializability

Relationship between View and Conflict Serializability

- Every conflict serializable schedule is view serializable, but not vice versa
- Conflict serializability is *stricter* than view serializability

Conflict serializable $\Rightarrow$ View serializable

Relationship between View and Conflict Serializability

- Every conflict serializable schedule is view serializable, but not vice versa
- Conflict serializability is *stricter* than view serializability
- They are same under the constrained write assumption
- In this assumption, every write of a data item x is constrained by the value of x it has read
  - write(f(read(x)))

Relationship between View and Conflict Serializability

- Every conflict serializable schedule is view serializable, but not vice versa
- Conflict serializability is *stricter* than view serializability
- They are same under the constrained write assumption
- In this assumption, every write of a data item x is constrained by the value of x it has read
  - write(f(read(x)))
- With unconstrained writes (blind writes), a schedule that is view serializable is not necessarily conflict serializable
  - Blind writes: writes to a data item without reading it

Relationship between View and Conflict Serializability

- Every conflict serializable schedule is view serializable, but not vice versa
- Conflict serializability is *stricter* than view serializability
- They are same under the constrained write assumption
- In this assumption, every write of a data item x is constrained by the value of x it has read
  - write(f(read(x)))
- With unconstrained writes (blind writes), a schedule that is view serializable is not necessarily conflict serializable
  - Blind writes: writes to a data item without reading it
- Every view serializable schedule that is not conflict serializable must have blind writes

Other Notions of Equivalence

- Conflict and view serializable schedules are restrictive in the sense that they aim to guarantee database consistency without analyzing the result

Other Notions of Equivalence

- Conflict and view serializable schedules are restrictive in the sense that they aim to guarantee database consistency without analyzing the result
- A schedule S is result equivalent to a schedule S' if it produces the same result as S'
- Consider
  ~~View~~
  $r_1(A); A := A - 50; w_1(A); \boxed{r_2(B)}; B := B - 10; w_2(B);$ } View
  $r_1(B); B := B + 50; w_1(B); r_2(A); A := A + 10; w_2(A);$
- It produces the same result as the serial schedule
  View
  $r_1(A); A := A - 50; w_1(A); \boxed{r_1(B)}; B := B + 50; w_1(B);$
  $r_2(B); B := B - 10; w_2(B); r_2(A); A := A + 10; w_2(A);$
  but is not conflict or view serializable
- Determining such equivalence requires *semantic* analysis of operations other than read and write

Testing for Serializability

- Create a precedence graph for the schedule
- Directed graph where each transaction is a vertex
- An edge from transaction T_i to T_j exists if
  - $w_i(x)$ precedes $r_j(x)$, or RAW
  - $r_i(x)$ precedes $w_j(x)$, or WAR
  - $w_i(x)$ precedes $w_j(x)$ WAW

Conflict cond^n

Testing for Serializability

- Create a precedence graph for the schedule
- Directed graph where each transaction is a vertex
- An edge from transaction T_i to T_j exists if
  - $w_i(x)$ precedes $r_j(x)$, or
  - $r_i(x)$ precedes $w_j(x)$, or
  - $w_i(x)$ precedes $w_j(x)$
- A schedule is conflict serializable if and only if its precedence graph is *acyclic*
- *Depth-first search* can detect cycles in $O(n + m)$ time
- *Topological sorting* produces an equivalent serial order

Testing for Serializability

- Create a precedence graph for the schedule
- Directed graph where each transaction is a vertex
- An edge from transaction T_i to T_j exists if
  - $w_i(x)$ precedes $r_j(x)$, or
  - $r_i(x)$ precedes $w_j(x)$, or
  - $w_i(x)$ precedes $w_j(x)$
- A schedule is conflict serializable if and only if its precedence graph is *acyclic*
- *Depth-first search* can detect cycles in $O(n + m)$ time
- *Topological sorting* produces an equivalent serial order
- Testing for view serializability is *NP-complete*
- Practical algorithms
  - Catches all non view serializable schedules
  - But can miss a view serializable schedule

Recoverable Schedule

- Conflict and view serializability do not address failures
- Order of commits and aborts are important

Recoverable Schedule

- Conflict and view serializability do not address failures
- Order of commits and aborts are important
- A schedule is called a recoverable schedule if
  - A transaction T_i reads a data item previously written by T_j, and
  - T_j commits *before* T_i commits

Recoverable Schedule

- Conflict and view serializability do not address failures
- Order of commits and aborts are important
- A schedule is called a recoverable schedule if
  - A transaction T_i reads a data item previously written by T_j, and
  - T_j commits *before* T_i commits
- Consider $r_1(a)w_1(a)r_2(a)r_1(b)$

Recoverable Schedule

- Conflict and view serializability do not address failures
- Order of commits and aborts are important
- A schedule is called a recoverable schedule if
  - A transaction T_i reads a data item previously written by T_j, and
  - T_j commits *before* T_i commits
- Consider $r_1(a)w_1(a)r_2(a)r_1(b)$
- If T_2 commits just after $r_2(a)$, i.e., if the schedule is $r_1(a)w_1(a)r_2(a)c_2r_1(b)a_1$, then it is *not* recoverable
  - If T_1 crashes, then $w_1(a)$ is undone, but T_2 has already read a wrong value of a and committed

Recoverable Schedule

- Conflict and view serializability do not address failures
- Order of commits and aborts are important
- A schedule is called a recoverable schedule if
  - A transaction T_i reads a data item previously written by T_j, and
  - T_j commits *before* T_i commits ⇒ $w_j(X) \ldots \text{commit}(T_j)$ $r_i(X)$ commit T_i
- Consider $r_1(a)w_1(a)r_2(a)r_1(b)$ c_2 c_1
- If T_2 commits just after $r_2(a)$, i.e., if the schedule is $r_1(a)w_1(a)r_2(a)c_2r_1(b)a_1$, then it is *not* recoverable
  - If T_1 crashes, then $w_1(a)$ is undone, but T_2 has already read a wrong value of a and committed
- Therefore, to make it recoverable, the schedule should be $r_1(a)w_1(a)r_2(a)r_1(b)c_1c_2$
  - If T_1 aborts, T_2 can also abort

Cascading Rollbacks

- In recoverable schedules, a single transaction failure may lead to a series of rollbacks
- This is called cascading rollbacks or cascading aborts
- Consider $r_1(a)w_1(a)r_2(a)w_2(a)r_3(a)r_1(b)a_1\underset{a_2}{c_2}\underset{a_3}{c_3}$
- It is recoverable
- However, if T_1 fails, T_2 and T_3 must abort as well
- Not preferable as lot of work is undone

Cascadeless Schedule

- Cascading rollbacks are eliminated

Cascadeless Schedule

- Cascading rollbacks are eliminated
- A schedule is called a cascadeless schedule if
  - A transaction T_i reads a data item previously written by T_j, and
  - T_j commits before T_i *reads*

Cascadeless Schedule

- Cascading rollbacks are eliminated
- A schedule is called a cascadeless schedule if
  - A transaction T_i reads a data item previously written by T_j, and
  - T_j commits before T_i *reads*
- Consider $r_1(a)w_1(a)r_2(a)r_1(b)a_1c_2$

Cascadeless Schedule

- Cascading rollbacks are eliminated
- A schedule is called a cascadeless schedule if
  - A transaction T_i reads a data item previously written by T_j, and
  - T_j commits before T_i *reads*
- Consider $r_1(a)w_1(a)r_2(a)r_1(b)a_1c_2$
- It is not cascadeless as T_2 reads a written by T_1 before T_1 commits

Cascadeless Schedule

- Cascading rollbacks are eliminated
- A schedule is called a cascadeless schedule if
  - A transaction T_i reads a data item previously written by T_j, and
  - T_j commits before T_i *reads*
- Consider $r_1(a)w_1(a)r_2(a)r_1(b)a_1c_2$
- It is not cascadeless as T_2 reads a written by T_1 before T_1 commits
- Therefore, to make it cascadeless, the schedule should be $r_1(a)w_1(a)r_1(b)c_1r_2(a)c_2$
- No completed transaction needs to be rolled back

Cascadeless Schedule

- Cascading rollbacks are eliminated
- A schedule is called a cascadeless schedule if
  - A transaction T_i reads a data item previously written by T_j, and
  - T_j commits before T_i *reads*
- Consider $r_1(a)w_1(a)r_2(a)r_1(b)a_1c_2$
- It is not cascadeless as T_2 reads a written by T_1 before T_1 commits
- Therefore, to make it cascadeless, the schedule should be $r_1(a)w_1(a)r_1(b)c_1r_2(a)c_2$
- No completed transaction needs to be rolled back
- Every cascadeless schedule is recoverable, but not vice versa

Strict Schedule

- Problem of writes remains in the sense that a later transaction may overwrite an uncommitted write

Strict Schedule

- Problem of writes remains in the sense that a later transaction may overwrite an uncommitted write
- A schedule is called a strict schedule if
    - A transaction T_i reads or writes a data item previously written by T_j, and
    - T_j commits before T_i *reads* or *writes*

Strict Schedule

- Problem of writes remains in the sense that a later transaction may overwrite an uncommitted write
- A schedule is called a strict schedule if
  - A transaction T_i reads or writes a data item previously written by T_j, and
  - T_j commits before T_i *reads* or *writes*
- Consider $r_1(a) w_1(a) w_2(a) r_1(b) a_1 c_2$ {Cascadeless?}

$r_1(a)\, w_1(a)\, r_1(b)\, a_1 \quad w_2(a)\, c_2$

Strict Schedule

- Problem of writes remains in the sense that a later transaction may overwrite an uncommitted write
- A schedule is called a strict schedule if
  - A transaction T_i reads or writes a data item previously written by T_j, and
  - T_j commits before T_i *reads* or *writes*
- Consider $r_1(a)w_1(a)w_2(a)r_1(b)a_1c_2$
- It is not strict as T_2 writes a written by T_1 before T_1 commits

Strict Schedule

- Problem of writes remains in the sense that a later transaction may overwrite an uncommitted write
- A schedule is called a strict schedule if
  - A transaction T_i reads or writes a data item previously written by T_j, and
  - T_j commits before T_i *reads* or *writes*
- Consider $r_1(a) w_1(a) w_2(a) r_1(b) a_1 c_2$
- It is not strict as T_2 writes a written by T_1 before T_1 commits
- Therefore, to make it strict, the schedule should be $r_1(a) w_1(a) r_1(b) c_1 w_2(a) c_2$

Strict Schedule

- Problem of writes remains in the sense that a later transaction may overwrite an uncommitted write
- A schedule is called a strict schedule if
  - A transaction T_i reads or writes a data item previously written by T_j, and
  - T_j commits before T_i *reads* or *writes*
- Consider $r_1(a)w_1(a)w_2(a)r_1(b)a_1c_2$
- It is not strict as T_2 writes a written by T_1 before T_1 commits
- Therefore, to make it strict, the schedule should be $r_1(a)w_1(a)r_1(b)c_1w_2(a)c_2$
- Every strict schedule is cascadeless, but not vice versa

Takes care of WAW error

Relationship among Schedules