Introduction to Data Management
Transactions: Schedules

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle
Announcements

- HW 6 is hard
Announcements

- HW 6 is hard
  - Deadline extended to Sunday
Announcements

- HW 6 is hard
  - Deadline extended to Sunday

  Many people can say: “I know how to write a self-join in SQL.”

  Not many can say: “I implemented the cloud database back-end of a flights reservation system in Java.”
Announcements

- HW 6 is hard
  - Deadline extended to Sunday

  Many people can say: “I know how to write a self-join in SQL.”
  Not many can say: “I implemented the cloud database back-end of a flights reservation system in Java.”

Transactions today: think about how they apply to Flights application
Leveraging Indexes

- Often for applications, workloads can be well described
  - Flights application
    - Search method → query on city name values
  - Data visualization software (e.g. Tableau)
    - 2D plot → query on graph axis bounds

- **Create indexes to match expected query workload**
CREATE TABLE Users (  
id INT,  
age INT,  
score INT);

CREATE INDEX U_age ON Users(age)

Unclustered by default

CREATE INDEX U_age_score ON Users(age, score)

Order specifies precedence in sorting

CREATE CLUSTERED INDEX U_score_age ON Users(score, age)

Reorders data on disk! (Fails if another clustered index exists)
CREATE TABLE Users (  
id INT PRIMARY KEY,  
age INT,  
score INT, ...);

What indexes could we make on Users?

Expecting 1000 exec/day
SELECT *  
FROM Users, Assets  
WHERE Users.id = Assets.uid

Expecting 1000 exec/day
SELECT *  
FROM Users  
WHERE Users.score > 95

Expecting 10 exec/day
SELECT *  
FROM Users  
WHERE Users.age > 21
CREATE TABLE Users (  id INT PRIMARY KEY,  age INT,  score INT, ...);

What indexes could we make on Users?

Expecting 1000 exec/day
SELECT *  FROM Users, Assets  WHERE Users.id = Assets.uid

IDs are unique so an unclustered index would do fine.

Expecting 1000 exec/day
SELECT *  FROM Users  WHERE Users.score > 95

Expecting 10 exec/day
SELECT *  FROM Users  WHERE Users.age > 21
CREATE TABLE Users (  
id INT PRIMARY KEY,  
age INT,  
score INT, ...);  

What indexes could we make on Users?

IDs are unique so an unclustered index would do fine.

Expecting 1000 exec/day  
SELECT *  
  FROM Users, Assets  
WHERE Users.id = Assets.uid

Expecting 1000 exec/day  
SELECT *  
  FROM Users  
WHERE Users.score > 95

Only one can exist!

Expecting 10 exec/day  
SELECT *  
  FROM Users  
WHERE Users.age > 21

This range query would benefit from a clustered index on age

This range query would benefit from a clustered index on score
CREATE TABLE Users ( 
  id INT PRIMARY KEY, 
  age INT, 
  score INT, ...);

Expecting 1000 exec/day
SELECT * 
  FROM Users, Assets 
WHERE Users.id = Assets.uid

Expecting 1000 exec/day
SELECT * 
  FROM Users 
WHERE Users.score > 95

Expecting 10 exec/day
SELECT * 
  FROM Users 
WHERE Users.age > 21

What indexes could we make on Users?

IDs are unique so an unclustered index would do fine.

Things to consider:
• What is the expected result size for each query?
• Do either of these queries need to be returned ASAP?
CREATE TABLE Users ( 
    id INT PRIMARY KEY, 
    age INT, 
    score INT, ...);

What indexes could we make on Users?

IDs are unique so an unclustered index would do fine.

Without more information, default to clustering on the index that will be used more (clustered index on score)

Expecting 1000 exec/day
SELECT * 
    FROM Users, Assets
WHERE Users.id = Assets.uid

Expecting 1000 exec/day
SELECT * 
    FROM Users
WHERE Users.score > 95

Expecting 10 exec/day
SELECT * 
    FROM Users
WHERE Users.age > 21
Choosing how to configure a database system is an interesting (i.e. hard) problem.

A database that is used by many people will often need one or more dedicated personnel to manage it (Database Administrator)

- Logical design (multi-team coordination)
- Physical design (hardware and system considerations)
- Permission management (visibility and security)
- Integration (company acquisitions and mergers)
- ...
Multiple Joins

- **Pipelined Execution**
  - Tuples are processed through the entire query plan
  - Fast

(for your information, i.e. not on exam)
Pipelined Execution

- **Iterator interface of RA operators (Volcano Iterator Model)**
  - `open()` on every operator at start
  - `close()` on every operator at end
  - `next()` to get the next tuple from a child operator or input table

(for your information, i.e. not on exam)
Pipelined Execution Example

- Iterator interface of RA operators (Volcano Iterator Model)

(for your information, i.e. not on exam)
Pipelined Execution Example

- Iterator interface of RA operators (Volcano Iterator Model)
Pipelined Execution Example

- Iterator interface of RA operators (Volcano Iterator Model)

![Diagram showing the pipeline of RA operators:]

1. $\sigma_{R.b=5}$
2. $\bowtie_{S.a=T.a}$
3. $\bowtie_{R.a=S.a}$
4. $\sigma_{R.b=5}$

(open() for your information, i.e. not on exam)
Pipelined Execution Example

- Iterator interface of RA operators (Volcano Iterator Model)

(for your information, i.e. not on exam)
Pipelined Execution Example

- Iterator interface of RA operators (Volcano Iterator Model)

(for your information, i.e. not on exam)
Pipelined Execution Example

- Iterator interface of RA operators (*Volcano Iterator Model*)

\[
\begin{align*}
\sigma_{R.b=5} & \quad \text{next()}
\end{align*}
\]

\[
\begin{align*}
\bowtie_{S.a=T.a} & \\
\bowtie_{R.a=S.a} & \\
R & S & T
\end{align*}
\]

(for your information, i.e. not on exam)
Pipelined Execution Example

- Iterator interface of RA operators *(Volcano Iterator Model)*

\[ \sigma_{R.\text{b}=5} \quad \text{next()} \]

\[ \bowtie_{S.a=T.a} \quad \text{next()} \]

\[ \bowtie_{R.a=S.a} \]

\[ R \quad S \quad T \]
**Pipelined Execution Example**

- **Iterator interface of RA operators (Volcano Iterator Model)**

  next() implementation will depend on algorithm used (CSE 444)

```
(\sigma_{R.b=5} S.a=T.a) .
```

```
(\sigma_{R.a=S.a})
```

```
R \quad S \quad T
```
**Pipelined Execution Example**

- Iterator interface of RA operators (Volcano Iterator Model)

Diagram:

```
R  \( \bowtie \)  S  \( \bowtie \)  T
   \( \sigma_{R.b=5} \)
      \( \sigma_{S.a=T.a} \)
          \( \bowtie_{R.a=S.a} \)
```

(for your information, i.e. not on exam)
Pipelined Execution Example

- Iterator interface of RA operators (Volcano Iterator Model)

(for your information, i.e. not on exam)
Pipelined Execution Example

- Iterator interface of RA operators (Volcano Iterator Model)

\[
\begin{align*}
\text{t1} & \quad \sigma_{R.b=5} \quad \text{next()} \\
\bowtie_{S.a=T.a} & \quad \text{next()} \\
\bowtie_{R.a=S.a} & \quad \text{next()} \\
R & \quad \text{next()} \quad \text{next()} \ldots \\
S & \quad \text{next()} \quad \text{next()} \ldots \\
T & \quad \text{next()}
\end{align*}
\]

(for your information, i.e. not on exam)
Pipelined Execution Example

- Iterator interface of RA operators *(Volcano Iterator Model)*
Pipelined Execution Example

- Iterator interface of RA operators (Volcano Iterator Model)

\[
\begin{align*}
&T \bowtie S.a = T.a \\
&S \bowtie S.a = S.a \\
&\sigma_{R.b=5} \\
&\text{output}
\end{align*}
\]

(output for your information, i.e. not on exam)
Pipelined Execution Example

- Iterator interface of RA operators (Volcano Iterator Model)

(for your information, i.e. not on exam)
Transactions

How do we support multiple people using a database at the same time?

- Multiple application users
- Multiple application programmers
- Multiple analysts
- Imagine a world where each person had to wait in line to use your database 😞
Common Concurrency Control Problems

- Non-Atomic Operations
- Lost Update
- Dirty/Inconsistent Read
- Unrepeatable Read
- Phantom Read
Non-Atomic Operations

Confirm Purchase

6:23 left
Lost Update

- Write-Write (WW) conflict
- Consolidation scenario:

Time

Account 1 = 100, Account 2 = 100

User 1 wants to pool money into account 1
Set account 1 = 200

User 2 wants to pool money into account 2
Set account 2 = 0

Set account 2 = 200

Set account 1 = 0
Lost Update

- **Write-Write (WW) conflict**
- **Consolidation scenario:**

  - Account 1 = 100, Account 2 = 100
  - User 1 wants to pool money into account 1
  - Set account 1 = 200
  - Set account 2 = 0
  - User 2 wants to pool money into account 2
  - Set account 2 = 200
  - Set account 1 = 0

  At end: Account 1 = 0, Account 2 = 200
Lost Update

- Write-Write (WW) conflict
- Consolidation scenario:

Account 1 = 100, Account 2 = 100

User 1 wants to pool money into account 1
Set account 1 = 200
Set account 2 = 0

User 2 wants to pool money into account 2
Set account 2 = 200
Set account 1 = 0

November 15, 2019
Lost Update

- Write-Write (WW) conflict
- Consolidation scenario:

Account 1 = 100, Account 2 = 100

User 1 wants to pool money into account 1
Set account 1 = 200
Set account 2 = 0

User 2 wants to pool money into account 2
Set account 2 = 200
Set account 1 = 0

At end: Account 1 = 0, Account 2 = 0
Dirty/Inconsistent Read

- Write-Read (WR) conflict
- Budget management scenario:

  Manager wants to balance project budgets
  - $10mil from project A
  + $7mil to project B
  + $3mil to project C

  CEO wants to check company balance
  SELECT SUM(money) ...
Dirty/Inconsistent Read

- Write-Read (WR) conflict
- Budget management scenario:

Manager wants to balance project budgets

CEO wants to check company balance

- $10mil from project A
- $7mil to project B
- $3mil to project C

SELECT SUM(money) ...
Dirty/Inconsistent Read

- Write-Read (WR) conflict
- Budget management scenario:

  Manager wants to balance project budgets
  
  CEO wants to check company balance
  
  -$10mil from project A
  
  +$7mil to project B
  
  +$3mil to project C

  SELECT SUM(money) ...
Dirty/Inconsistent Read

- Write-Read (WR) conflict
- Budget management scenario:

  Manager wants to balance project budgets
  - $10mil from project A
  + $7mil to project B
  + $3mil to project C

  CEO wants to check company balance

  SELECT SUM(money) ...
Unrepeatable Read

- Read-Write (RW) conflict
- Asset checking scenario:

  Accountant wants to check company assets

  Application is automatically updating inventories

  SELECT inventory
  FROM Products
  WHERE pid = 1

  UPDATE Products
  SET inventory = 0
  WHERE pid = 1

  SELECT inventory * price
  FROM Products
  WHERE pid = 1
Unrepeatable Read

- **Read-Write (RW) conflict**
- **Asset checking scenario:**

  Accountant wants to check company assets

  Application is automatically updating inventories

  ```
  SELECT inventory
  FROM Products
  WHERE pid = 1
  ```

  ```
  SELECT inventory * price
  FROM Products
  WHERE pid = 1
  ```

  ```
  UPDATE Products
  SET inventory = 0
  WHERE pid = 1
  ```

  Might get a value that doesn’t correspond to previous read!
Phantom Read

- Same read has more rows
- Asset checking scenario:

Accountant wants to check company assets

SELECT *
FROM products
WHERE price < 10.00

INSERT INTO Products
VALUES (‘nuts’, 10, 8.99)

Warehouse catalogs new products

SELECT *
FROM products
WHERE price < 20.00
Phantom Read

- Same read has more rows
- Asset checking scenario:

  Accountant wants to check company assets

  Warehouse catalogs new products

  INSERT INTO Products
  VALUES ('nuts', 10, 8.99)

  SELECT *
  FROM products
  WHERE price < 10.00

  SELECT *
  FROM products
  WHERE price < 20.00

  Gets a row that should have been in the last read!
ACID

- Atomic
- Consistent
- Isolated
- Durable

Ideally a DBMS follows these principles, but sacrificing good behavior for performance gains is common

Definitely needs to follow these principles if you are dealing with $$$
ACID

- Atomic
- Consistent
- Isolated
- Durable

- Ideally a DBMS follows these principles, but sacrificing good behavior for performance gains is common.

- Definitely needs to follow these principles if you are dealing with $$$
Atomic

- Operation encapsulation
- An operation is atomic if everything works or nothing happens

- 2nd most important in ACID! And feature we need for programming
Consistent

- Integrity constraints and application specification
- Operations assume a valid database state and end in a valid database state
Isolated

- Concurrency management
- Isolated behavior is as if an operation ran as if it was the only one running
- The most important feature of ACID.
- Atomicity + Isolation is what gives us functional transactions.
Durable

- Crash recovery
- CSE 444 topic
  - not discussed in this class (but very interesting!)
Transactions

- An application function may involve multiple different operations
- We want to make sure the parts of an operation execute properly together as if it were a single action
- We say that a transaction is one of these groups of executions
  - DBMS usually automatically treats each SQL statement as its own transaction unless otherwise specified

BEGIN TRANSACTION
[SQL Statements]
COMMIT – finalizes execution

BEGIN TRANSACTION
[SQL Statements]
ROLLBACK – undo everything
Concurrent Control Problems

- We’ve (sorta) solved the first problem!
- **DBMS concurrency control is all based on specification**
- Merely specifying what your transactions are is good enough for the DBMS to take care of it as a single unit
Transaction Modeling

- Logical perspective → a database is a set of sets/bags of tuples
- Design perspective → a database is a schema that models information
- Physical perspective → a database is a catalog of organized files
- Transaction perspective → a database is a collection of elements that can be written to or read from
  - Definition of element can vary depending on DBMS and/or user specification
  - Usually element = 1 block of database
  - Transactions are sequence of element reads and/or writes
Schedules

- Transactions are sequence of element reads and/or writes
  - \( R_i(A) \rightarrow \text{read} \) element A
  - \( W_i(A) \rightarrow \text{update} \) element A

To add or remove tuples, we need more operations
- \( I_i(A) \rightarrow \text{insert} \) an element A
- \( D_i(A) \rightarrow \text{delete} \) an element A

- Schedules are a sequence of interleaved actions from all transactions
Serial Schedules

- A **serial schedule** is a schedule where each transaction would be executed in some order.
- A **serializable schedule** is a schedule where transaction reads and writes would be executed as if it were executed in serial order.
  - If the schedule were executed and you were given a before and after, you would not be able to tell if there was interleaving.
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>R(B)</td>
<td>R(B)</td>
</tr>
<tr>
<td>W(B)</td>
<td>W(B)</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A,s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>WRITE(B,s)</td>
</tr>
</tbody>
</table>
A Serial Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>WRITE(A, t)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>READ(B, t)</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>t := t+100</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>WRITE(B, t)</td>
</tr>
</tbody>
</table>

\[ A = 2 \]
\[ B = 2 \]

\[ A = 102 \]
\[ B = 102 \]

\[ A = 204 \]
\[ B = 204 \]
A Serial Schedule

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>READ(A, s)</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td></td>
<td>s := s*2</td>
<td>s := s*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(A, s)</td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td></td>
<td>READ(B, s)</td>
<td>READ(B, s)</td>
</tr>
<tr>
<td></td>
<td>s := s*2</td>
<td>s := s*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B, s)</td>
<td>WRITE(B, s)</td>
</tr>
</tbody>
</table>

A = 2
B = 2

A = 4
B = 4

A = 104
B = 104
A schedule is *serializable* if it is equivalent to a serial schedule.
### A Serializable Schedule

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td></td>
<td>READ(A,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>WRITE(A, t)</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td></td>
<td>WRITE(A,s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td></td>
<td>READ(B,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>WRITE(B,t)</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td></td>
<td>WRITE(B,s)</td>
</tr>
</tbody>
</table>

This is a serializable schedule.

This is NOT a serial schedule.
### A Non-Serializable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A,s)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>WRITE(B,s)</td>
</tr>
</tbody>
</table>

A = 2  B = 2
A = 102 B = 2
A = 204 B = 2
A = 204 B = 4
A = 204 B = 104

- A = 2
- B = 2
- A = 102
- B = 2
- A = 204
- B = 2
- A = 204
- B = 4
- A = 204
- B = 104
# A Non-Serializable Schedule

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>READ(A, t)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>t := t+100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE(A, t)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>READ(A, s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>s := s*2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>READ(B, s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>s := s*2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WRITE(B, s)</td>
</tr>
<tr>
<td></td>
<td>READ(B, t)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>t := t+100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE(B, t)</td>
<td></td>
</tr>
</tbody>
</table>

|   | A = 2                  |
|   | B = 2                  |
|   | A = 102                |
|   | B = 2                  |
|   | A = 204                |
|   | B = 2                  |
|   | A = 204                |
|   | B = 4                  |
|   | A = 204                |
|   | B = 104                |

Shouldn’t be possible!
## Serial Schedule Example

- **T1 then T2**

\[ R_1(A), W_1(A), R_1(B), W_1(B), R_2(A), W_2(A), R_2(B), W_2(B) \]

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
</tr>
</tbody>
</table>
Serial Schedule Example

- T2 then T1

\[ R_2(A), W_2(A), R_2(B), W_2(B), R_1(A), W_1(A), R_1(B), W_1(B) \]
Serializable Schedule

- Serializable to T1 then T2

\[ R_1(A), W_1(A), R_2(A), W_2(A), R_1(B), W_1(B), R_2(B), W_2(B) \]
Serializable Schedule

- Serializable to T1 then T2

\[ R_1(A), W_1(A), R_2(A), W_2(A), R_1(B), W_1(B), R_2(B), W_2(B) \]

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
</tr>
</tbody>
</table>

Looks like T2 finished after T1 for each element.
Serializable Schedule

- Not serializable to either order

\[ R_1(A), W_1(A), R_2(A), W_2(A), R_2(B), W_2(B), R_1(B), W_1(B) \]
 Serializable Schedule

- **Not serializable to either order**

\[ R_1(A), W_1(A), R_2(A), W_2(A), R_2(B), W_2(B), R_1(B), W_1(B) \]

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
</tbody>
</table>

Looks like T2 finished last looking at A

Looks like T1 finished last looking at B

November 15, 2019
Checking Serializability

- How does the DBMS tell if some schedule is serializable?

- We define operation “conflicts” and check for their interactions in a schedule
Most application concurrency problems are describable by conflicts

- Lost Update $\rightarrow$ Write-Write (WW) conflict
- Dirty Read $\rightarrow$ Write-Read (WR) conflict
- Unrepeatable Read $\rightarrow$ Read-Write (RW) conflict
- Phantom Read
  - We’ll talk about this later…

Individual conflicts aren’t “bad”!
Interleaving of conflicts can lead to trouble.
Types of Conflicts

- Changing the order of things in conflict will cause program behavior to behave badly

- **Intra-transaction conflicts**
  - Operations within a transaction cannot be swapped (you would be literally changing the program)

- **Inter-transaction conflicts**
  - WW conflicts → $W1(X), W2(X)$
  - WR conflicts → $W1(X), R2(X)$
  - RW conflicts → $R1(X), W2(X)$
Conflict Serializability

- Showing program serializability is hard
  - Needs lots of extra information besides R, W, I, D

- Observation: Enforce something simpler but stronger than serializability

All possible schedules (Venn diagram)
Conflict Serializability

- Showing program serializability is hard
  - Needs lots of extra information besides R, W, I, D

- Observation: Enforce something something simpler but stronger than serializability

- **Conflict serializability implies serializability**

- Serializability does not imply conflict serializability
## Conflict Serializable Schedule Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
</tr>
</tbody>
</table>
## Conflict Serializable Schedule Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
</tr>
</tbody>
</table>
## Conflict Serializable Schedule Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>W(B)</td>
<td>W(B)</td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
</tbody>
</table>
## Conflict Serializable Schedule Example

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
<td></td>
</tr>
</tbody>
</table>
# Conflict Serializable Schedule Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
</tr>
</tbody>
</table>
### Non Conflict Serializable Schedule Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>R(B)</td>
<td>R(B)</td>
</tr>
<tr>
<td>W(B)</td>
<td>W(B)</td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
</tbody>
</table>
### Non Conflict Serializable Schedule Example

**Serializability**

- **T1**
  - R(A)
  - W(A)
  - R(B)
  - R(B)

- **T2**
  - R(A)
  - W(A)
  - R(B)
  - W(B)

**Conflict rule broken!**
### Serializable vs Conflict Serializable

**Not serializable nor conflict serializable**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A←A+10</td>
<td>R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>B←B+10</td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
</tr>
</tbody>
</table>

A←(A+10)*2  B←(B*2)+10
Serializable but not conflict serializable
(because the values ended up equivalent to serial schedule)
Enforcing Conflict Serializability

- We only care if some conflict rule would be broken (no need to micromanage)
- Need an effective algorithm

Method:
- Model each transaction as a node
- Model a inter-transaction conflict as a directed edge
- If the resulting graph is a DAG then there is a serial order
- Conflict serializability enforcement turns into the graph cycle detection problem
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 $T_j$
- No edge for actions in the same transaction

Theorem:
The schedule is conflict-serializable iff the precedence graph is acyclic
Testing for Conflict-Serializability

Important:

Always draw the full graph, unless ONLY asked if (yes or no) the schedule is conflict serializable
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
Example 1

\[
\begin{align*}
\text{r}_2(\text{A}) & \quad \text{r}_1(\text{B}) \\
\text{r}_2(\text{A}); \text{r}_1(\text{B}); \text{w}_2(\text{A}); \text{r}_3(\text{A}); \text{w}_1(\text{B}); \text{w}_3(\text{A}); \text{r}_2(\text{B}); \text{w}_2(\text{B})
\end{align*}
\]
Example 1

\[ r_2(A) \quad r_1(B) \]

\[ r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B) \]

1 2 3
No edge because no conflict (A != B)
No edge because same txn (2)
Example 1

\[ r_2(A) \quad r_3(A) \quad ? \]

\[ r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B) \]

1  2  3
Example 1

\[ r_2(A) \quad w_1(B) \quad ? \]

\[ r_2(A); \quad r_1(B); \quad w_2(A); \quad r_3(A); \quad w_1(B); \quad w_3(A); \quad r_2(B); \quad w_2(B) \]

1 2 3
Example 1

\[ r_2(A); \quad w_3(A) \]

\[ r_2(A); \quad r_1(B); \quad w_2(A); \quad r_3(A); \quad w_1(B); \quad w_3(A); \quad r_2(B); \quad w_2(B) \]
Edge! Conflict from T2 to T3
Edge! Conflict from T2 to T3

r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)
Example 1

\[ r_2(A) \quad r_2(B) \quad ? \]

\[ r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B) \]

And so on until compared every pair of actions…
Example 1

More edges, but repeats of the same directed edge not necessary

\[ r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B) \]
Example 1

This schedule is conflict-serializable

\[ r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B) \]
Example 2

\[ r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B) \]
Example 2

\[ r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B) \]
Example 2

This schedule is NOT conflict-serializable

\[ r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B) \]