CS5200 Database Management Concurrency and Metadata

Ken Baclawski
Spring 2017

Outline

- Transactions
- Theory of Concurrency
  - View Equivalence
  - Conflict Equivalence
  - Serializability
  - Recoverability Criteria
  - Guaranteeing Serializability
- Specifying Transactions
- Metadata Management
- Assignment #9

Transactions

Concurrency

- Many users of the same data
- Intrinsic property of services
- Major problem for large systems
  - Banks
  - Reservations Systems
  - E-Commerce Sites
- Even a cellphone must control concurrency

Lost Update Problem

- Two persons are depositing $100 in the same bank account at the same time in different locations
- Assume the initial balance in the account is $1000

Person A	**Person B**	**Balance**
Read balance to a local variable		$1000
	Read balance to a local variable	$1000
Add $100 to the local value of the balance		$1000
	Add $100 to the local value of the balance	$1000
Write new balance		$1100
	Write new balance	$1100

Dirty Read Problem

- Person A transfers $100 from checking to savings
- Person B computes the total amount of both accounts

Person A	**Checking**	**Savings**	**Person B**
Read checking balance	$1000	$2000	
Read savings balance	$1000	$2000	
Subtract $100 from local checking value	$1000	$2000	
Write new checking balance	$900	$2000	
	$900	$2000	Read checking balance
Add $100 to local savings balance	$900	$2000	
	$900	$2000	Read savings balance
Write new savings balance	$900	$2100	
			Report total as $2900

Failure Problem

- A system failure occurs during the transfer

Person A	Checking	Savings	Person B
Read checking balance	$1000	$2000	
Read savings balance	$1000	$2000	
Subtract $100 from local checking value	$1000	$2000	
Write new checking balance	$900	$2000	
System Failure!	$900	$2000	Read checking balance
Add $100 to local savings balance	$900	$2000	
	$900	$2000	Read savings balance
Write new savings balance	$900	$2100	
			Report total as $2900

Transaction Boundaries

- Transaction Begin
  - Implicit when the database is accessed and a transaction is not currently running
- Transaction Commit
  - Requests that the transaction end
- Transaction Rollback
  - The transaction is canceled
  - All data items are returned to their original values
  - Can occur for various reasons, such as deadlocks

Transactions

- A transaction is a unit of work by one process
- The transaction should satisfy
  - **A**tomic. The work either entirely happens or does not happened at all (indivisible)
  - **C**onsistent. If the database is consistent when the work starts, then it is consistent when the work is finished
  - **I**solated. The work is executed as if nothing else was happening on the database
  - **D**urable. Upon completion of the work, all changes to the database are permanent even if the systems fails

Solving the Problems

- The Lost Update Problem is solved by Isolation
- The Dirty Read Problem is solved by Isolation
- The Failure Problem is solved by Atomicity
- Consistency is the responsibility of the programmer
- Durability will be discussed in a later class when we have covered buffer management
- Transactions are designed to solve all of these problems and many others
  - It is a general formulation for database access correctness

Theory of Concurrency

Data and Operations

- The database is assumed to consist of a collection of data items
  - Each item can be read and written independently of other items
  - Items will be denoted by A, B, C, etc.
- There are two operations on data items: read and write
  - R(A) means read data item A
  - W(B) means write a new value to data item B

Transactions

- A transaction is a sequence of operations on data items ending with a commit
  - The operations are determined by the program
  - Transactions are specified with numbers: 1, 2, 3, etc.
  - Commit is denoted by C
- Example of transactions:

  T1: `R1(C), W1(A), W1(A), W1(C), C1`

  T2: `R2(B), R2(C), W2(A), C2`

Rollback

- Many textbooks and papers distinguish commit and rollback for concurrency control
- In fact, a transaction that has been rolled back performs reads and writes and ends exactly like a normal transaction
- For example, if T2 fails after the write operation, then the rollback will write the old value of A and will then commit so it looks like this:

  T2-rolledback: `R2(B), R2(C), W2(A), W2(A), C2`

Schedule

- Sequence of operations by some transactions
- The operations by the transactions may be interleaved with one another
- Example of a schedule with transactions T1 and T2

```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A),
C2, W1(C), C1
```

  - Note that the operations of a transactions are *interleaved*
  - The order of operations of each transaction is unchanged

Equivalence of Schedules

- Two schedules are *equivalent* if their effects on the transactions and the database are the same
- There are two notions of equivalence
  - View
  - Conflict
- Equivalence is only concerned with the R and W operations of the schedules
  - The commits are not considered

View Equivalence

View Equivalence

- Two schedules are *view equivalent* when
  - The two schedules have the same transactions
  - Every read operation of a transaction reads the same value in the two schedules
  - The final value of every data item is the same when the two schedules complete
- To check view equivalence:
  - Link each operation in one schedule with the corresponding operation in the other
  - Check each read operation and final value

View Equivalence Example 1

The following slides show how to check whether these two schedules are view equivalent:

`R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1`

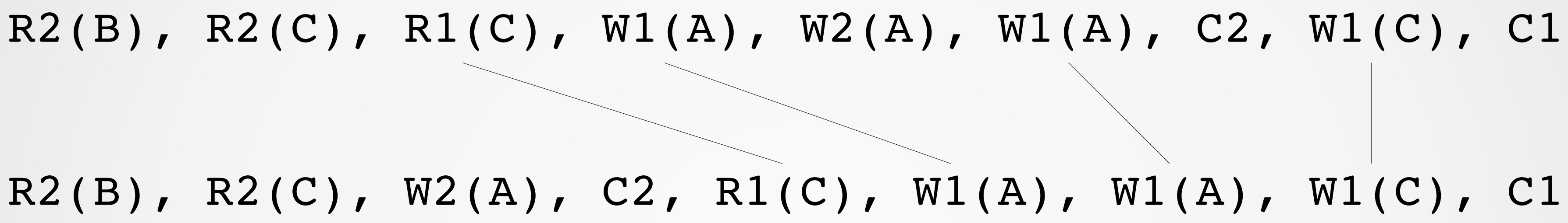
and

`R2(B), R2(C), W2(A), C2, R1(C), W1(A), W1(A), W1(C), C1`

```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1
```

```
R2(B), R2(C), W2(A), C2, R1(C), W1(A), W1(A), W1(C), C1
```

The two schedules to be compared

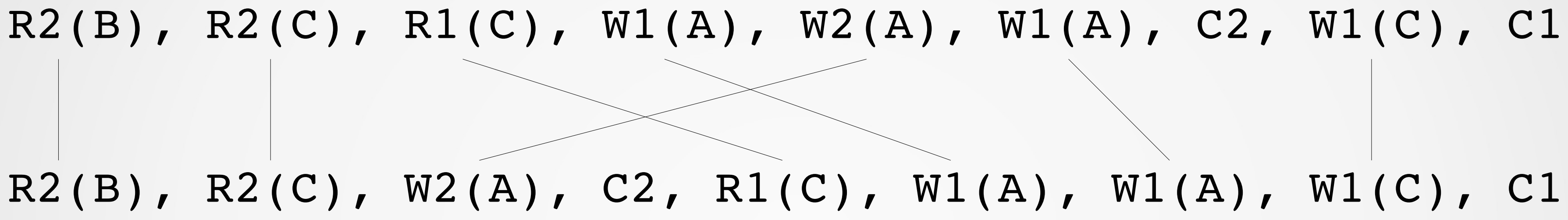


```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1

R2(B), R2(C), W2(A), C2, R1(C), W1(A), W1(A), W1(C), C1
```

Transaction 1 occurs in both schedules in the same order

```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1

R2(B), R2(C), W2(A), C2, R1(C), W1(A), W1(A), W1(C), C1
```

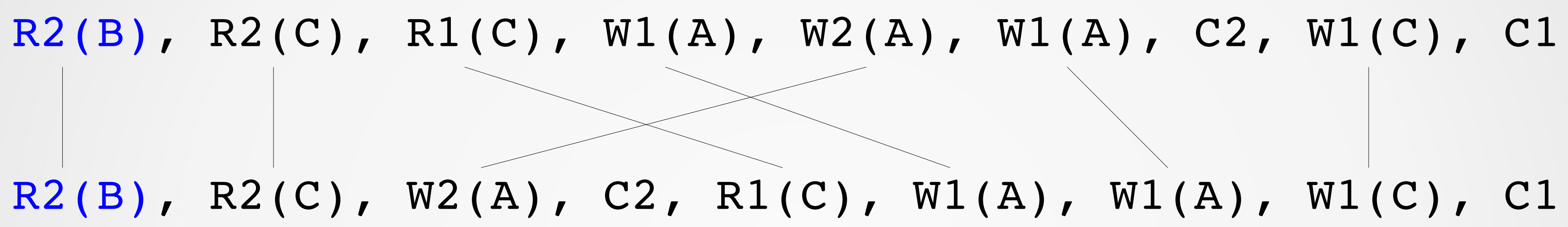
Transaction 2 occurs in both schedules in the same order



```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1

R2(B), R2(C), W2(A), C2, R1(C), W1(A), W1(A), W1(C), C1
```

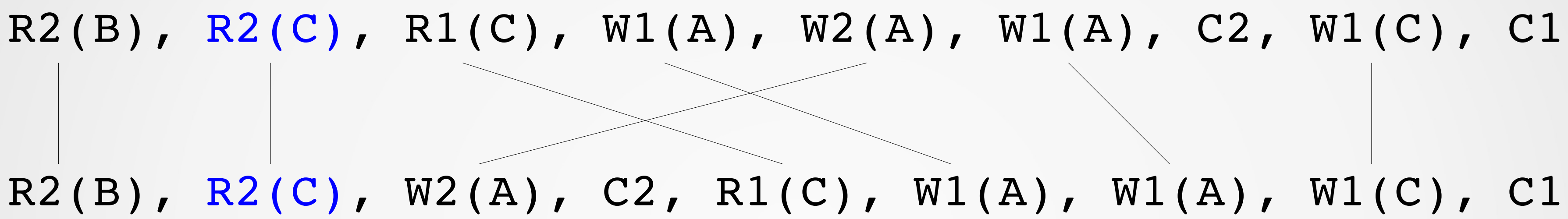
The matching of all read and write operations

```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1
```



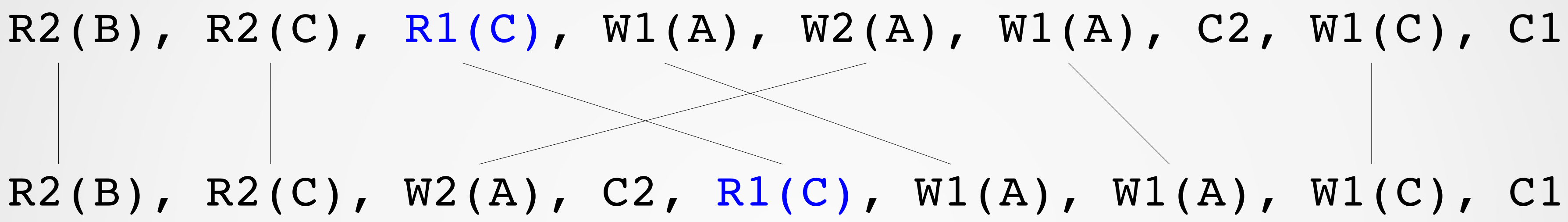
```
R2(B), R2(C), W2(A), C2, R1(C), W1(A), W1(A), W1(C), C1
```

R2(B) reads the initial value in both schedules



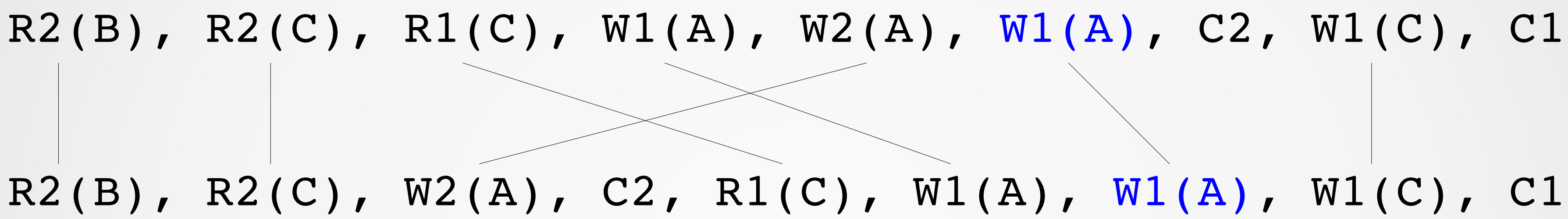
```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1

R2(B), R2(C), W2(A), C2, R1(C), W1(A), W1(A), W1(C), C1
```

R2(C) reads the initial value in both schedules



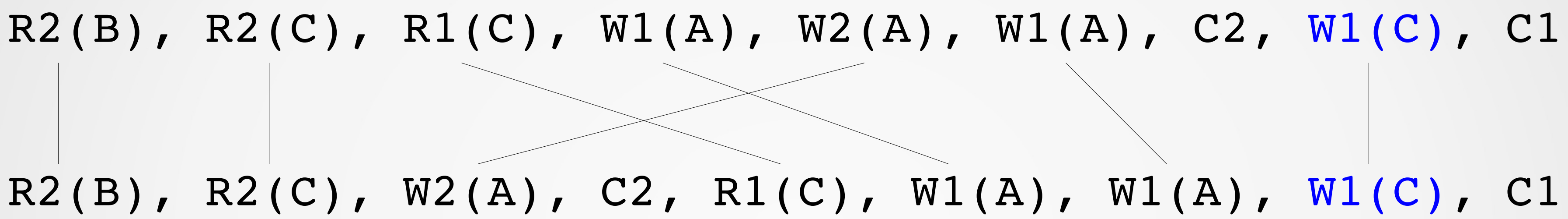
```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1

R2(B), R2(C), W2(A), C2, R1(C), W1(A), W1(A), W1(C), C1
```

R1(C) reads the initial value in both schedules



```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1

R2(B), R2(C), W2(A), C2, R1(C), W1(A), W1(A), W1(C), C1
```

The final value of A is the same in both schedules



```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1

R2(B), R2(C), W2(A), C2, R1(C), W1(A), W1(A), W1(C), C1
```

The final value of B is the same in both schedules because it is never written

The final value of C is the same in both schedules

View Equivalence Example 1

Conclusion: These two schedules **are** view equivalent

```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A),
C2, W1(C), C1
```

and

```
R2(B), R2(C), W2(A), C2, R1(C), W1(A),
W1(A), W1(C), C1
```

View Equivalence Example 2

The following slides show how to check whether these two schedules are view equivalent:

```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A),
C2, W1(C), C1
```

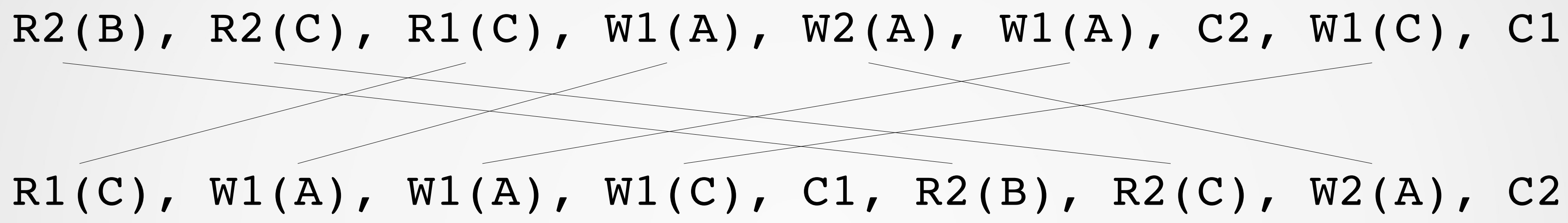
and

```
R1(C), W1(A), W1(A), W1(C), C1, R2(B),
R2(C), W2(A), C2
```

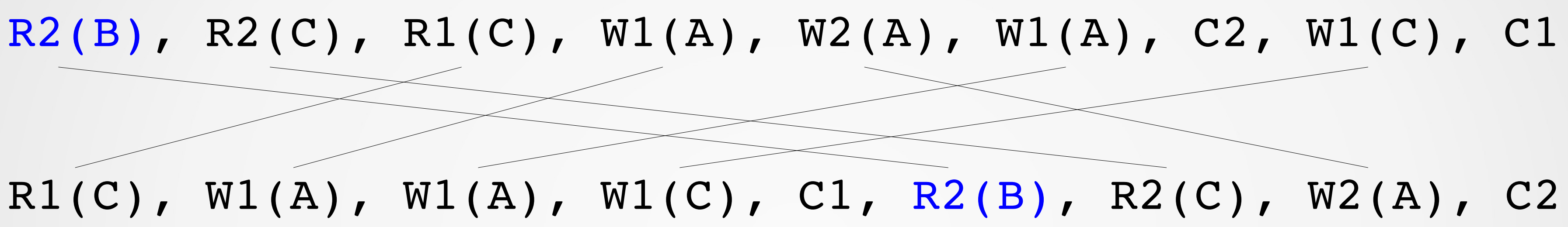
```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1
```

```
R1(C), W1(A), W1(A), W1(C), C1, R2(B), R2(C), W2(A), C2
```

The two schedules to be compared


R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1
R1(C), W1(A), W1(A), W1(C), C1, R2(B), R2(C), W2(A), C2

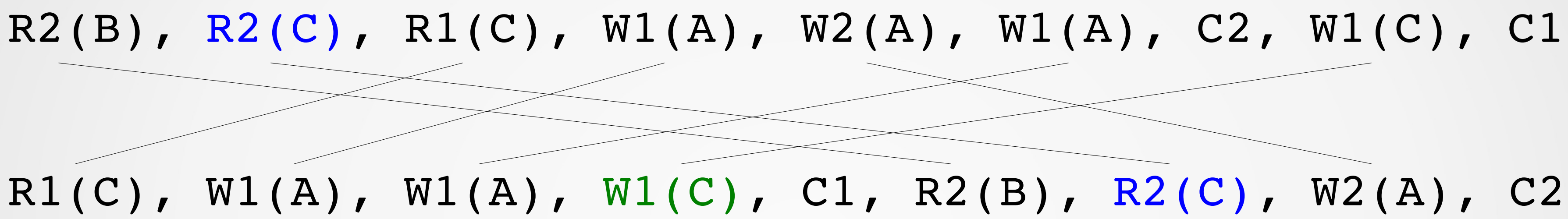
The matching of all read and write operations



R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1

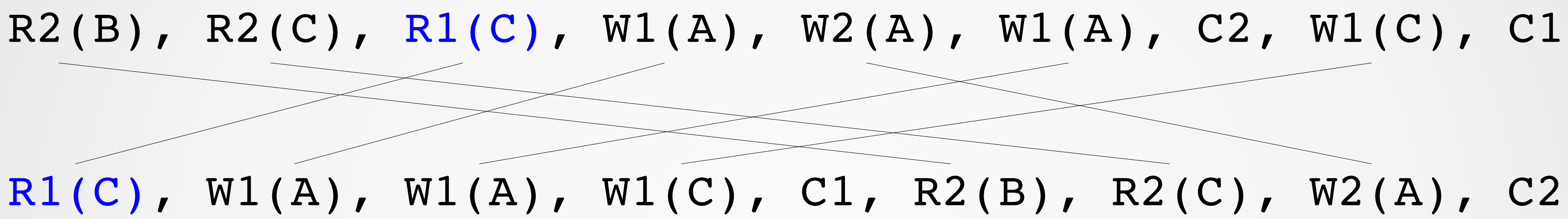
R1(C), W1(A), W1(A), W1(C), C1, R2(B), R2(C), W2(A), C2

R2(B) reads the initial value in both schedules



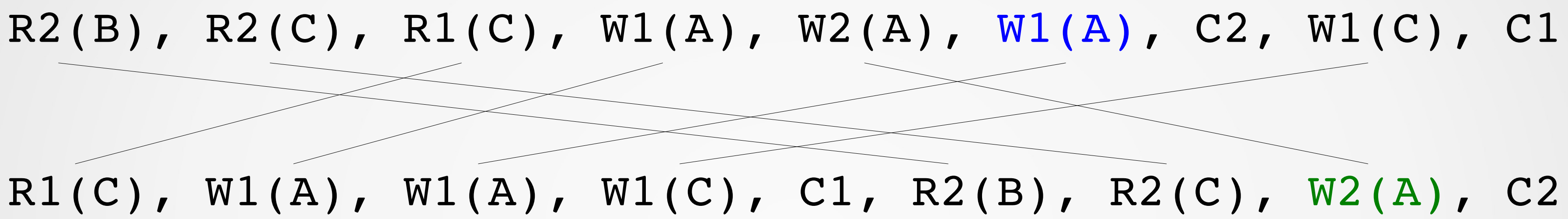
```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1

R1(C), W1(A), W1(A), W1(C), C1, R2(B), R2(C), W2(A), C2
```

R2(C) reads the initial value in the first schedule but W1(C) in the second
The two schedules are **not** view equivalent



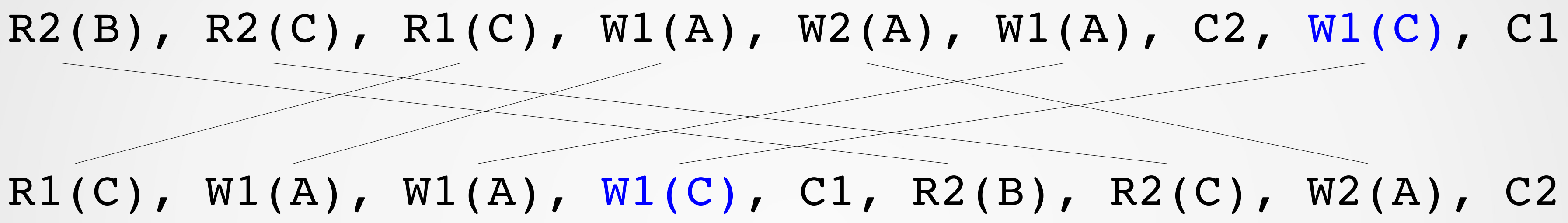
```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1

R1(C), W1(A), W1(A), W1(C), C1, R2(B), R2(C), W2(A), C2
```

R1(C) reads the initial value in both schedules



```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1

R1(C), W1(A), W1(A), W1(C), C1, R2(B), R2(C), W2(A), C2
```

The final value of A is different in the two schedules
The two schedules are **not** view equivalent



```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1

R1(C), W1(A), W1(A), W1(C), C1, R2(B), R2(C), W2(A), C2
```

The final value of B is the same in both schedules because it is never written

The final value of C is the same in both schedules

View Equivalence Example 2

Conclusion: These two schedules are **not** view equivalent

```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A),
C2, W1(C), C1
```

and

```
R1(C), W1(A), W1(A), W1(C), C1, R2(B),
R2(C), W2(A), C2
```

View Equivalence Example 3

The following slides show how to check whether these two schedules are view equivalent:

```
R2(C), W2(B), W1(C), R2(C), R1(C), R1(A),
C2, C1
```

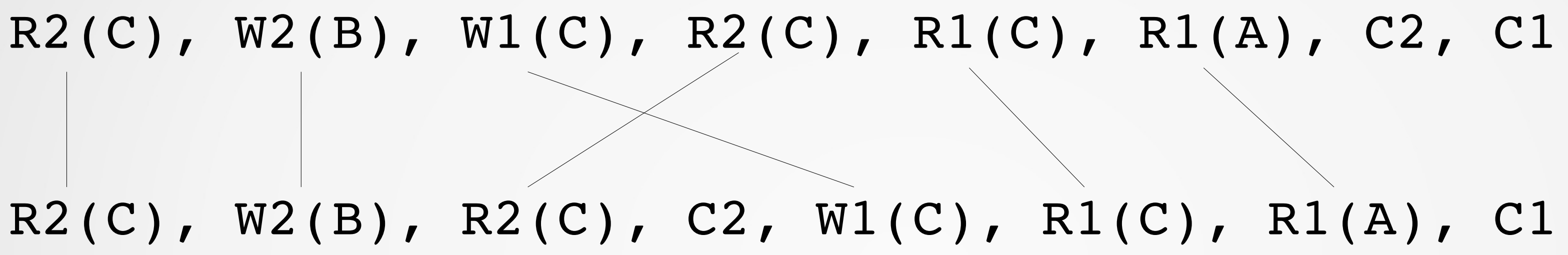
and

```
R2(C), W2(B), R2(C), C2, W1(C), R1(C),
R1(A), C1
```

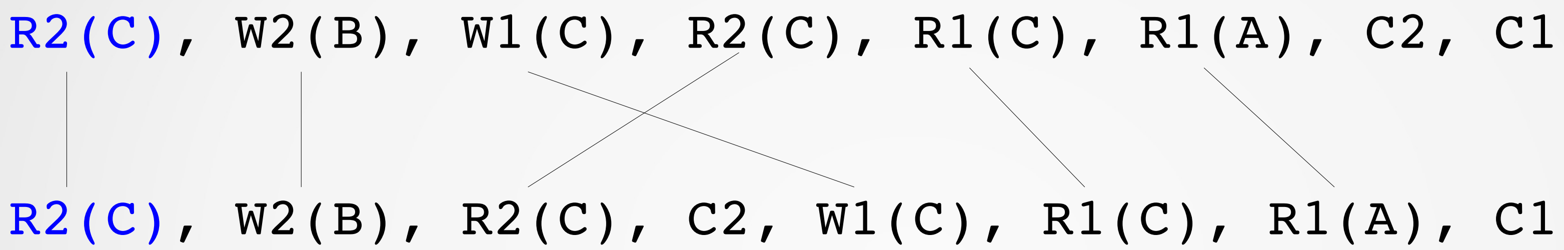
```
R2(C), W2(B), W1(C), R2(C), R1(C), R1(A), C2, C1
```

```
R2(C), W2(B), R2(C), C2, W1(C), R1(C), R1(A), C1
```

The two schedules to be compared

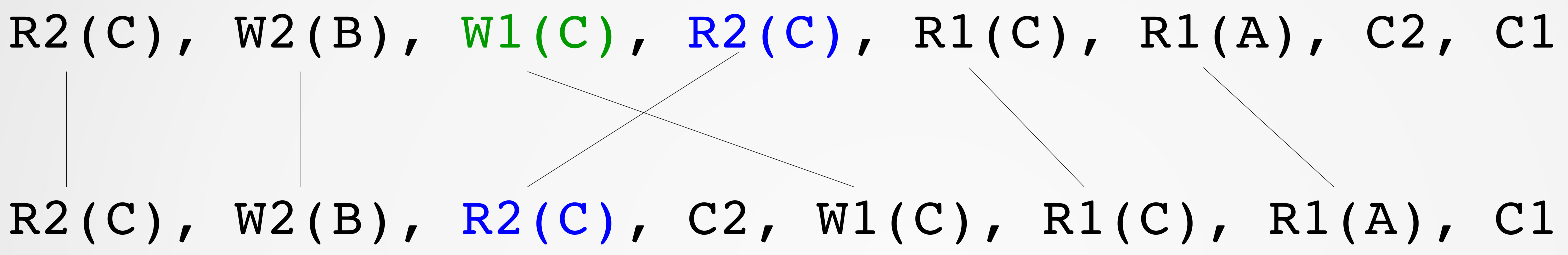

R2(C), W2(B), W1(C), R2(C), R1(C), R1(A), C2, C1
R2(C), W2(B), R2(C), C2, W1(C), R1(C), R1(A), C1

The matching of all read and write operations



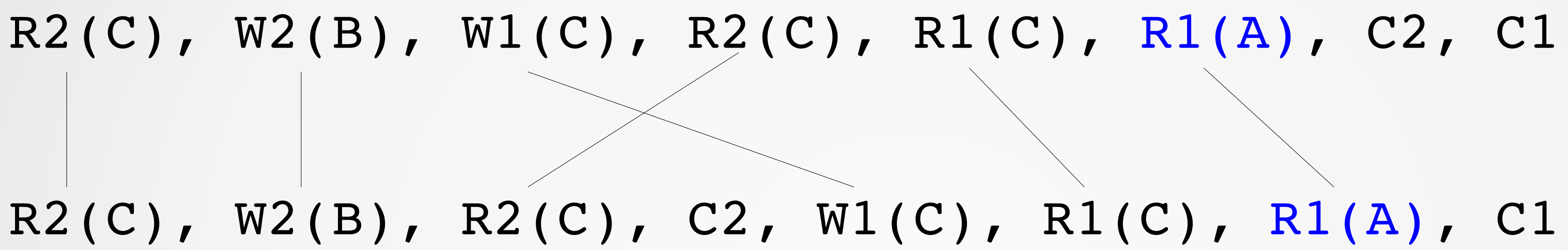
```
R2(C), W2(B), W1(C), R2(C), R1(C), R1(A), C2, C1

R2(C), W2(B), R2(C), C2, W1(C), R1(C), R1(A), C1
```

R2(C) reads the initial value in both schedules



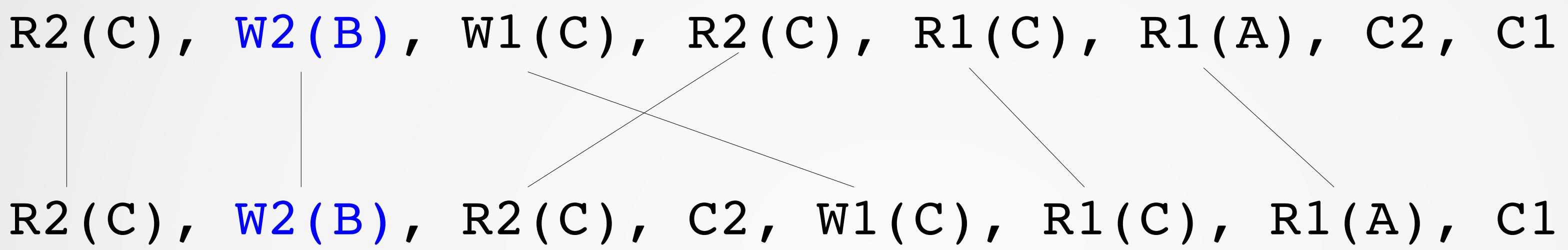
```
R2(C), W2(B), W1(C), R2(C), R1(C), R1(A), C2, C1

R2(C), W2(B), R2(C), C2, W1(C), R1(C), R1(A), C1
```

Not view equivalent because `R2(C)` reads `W1(C)` in one schedule and the initial value of C in the other



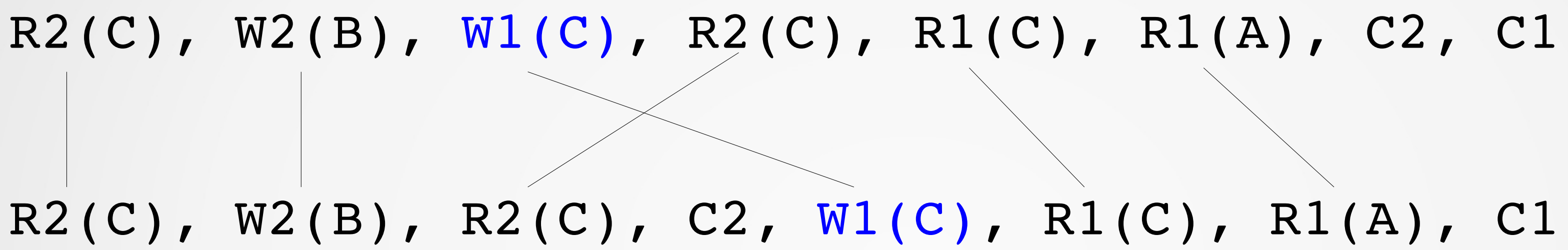
```
R2(C), W2(B), W1(C), R2(C), R1(C), R1(A), C2, C1

R2(C), W2(B), R2(C), C2, W1(C), R1(C), R1(A), C1
```

R1(A) reads the initial value in both schedules



```
R2(C), W2(B), W1(C), R2(C), R1(C), R1(A), C2, C1

R2(C), W2(B), R2(C), C2, W1(C), R1(C), R1(A), C1
```

The final value of B is the same in both schedules



```
R2(C), W2(B), W1(C), R2(C), R1(C), R1(A), C2, C1

R2(C), W2(B), R2(C), C2, W1(C), R1(C), R1(A), C1
```

The final value of C is the same in both schedules

View Equivalence Example 3

Conclusion: These two schedules are **not** view equivalent

```
R2(C), W2(B), W1(C), R2(C), R1(C), R1(A),
C2, C1
```

and

```
R2(C), W2(B), R2(C), C2, W1(C), R1(C),
R1(A), C1
```

Conflict Equivalence

Conflicts

- A *conflict* is a pair of operations such that
  - The operations are for different transactions
  - The operations are on the same data item
  - At least one of the two operations is a write
- There are three kinds of conflict
  - RW conflict: `...Ri(A)...Wj(A)...`
  - WR conflict: `...Wi(A)...Rj(A)...`
  - WW conflict: `...Wi(A)...Wj(A)...`

Conflict Equivalence

- Two schedules are *conflict equivalent* when
  - The two schedules have the same transactions
  - For every conflict, the operations are in the same order in both schedules
- To check conflict equivalence:
  - Link each operation in one schedule with the corresponding operation in the other
  - Check that each conflict is in the same order

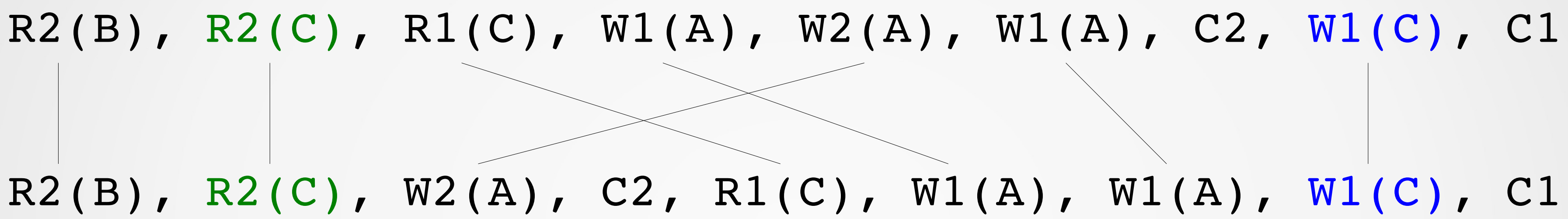
Conflict Equivalence Example

The following slides show how to check whether these two schedules are conflict equivalent:

```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A),
C2, W1(C), C1
```

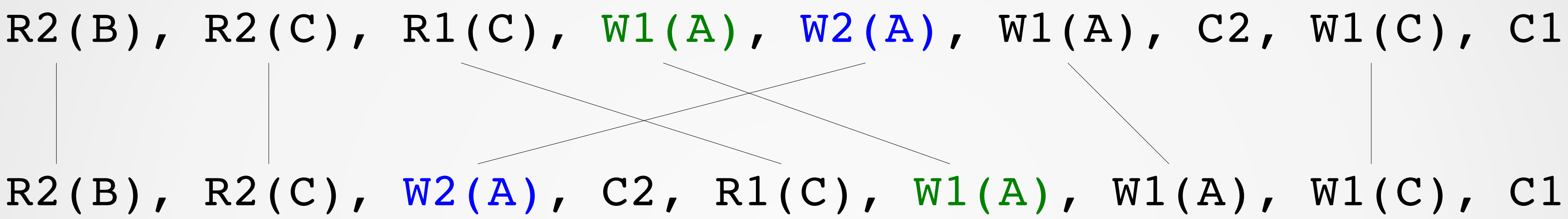
and

```
R2(B), R2(C), W2(A), C2, R1(C), W1(A),
W1(A), W1(C), C1
```



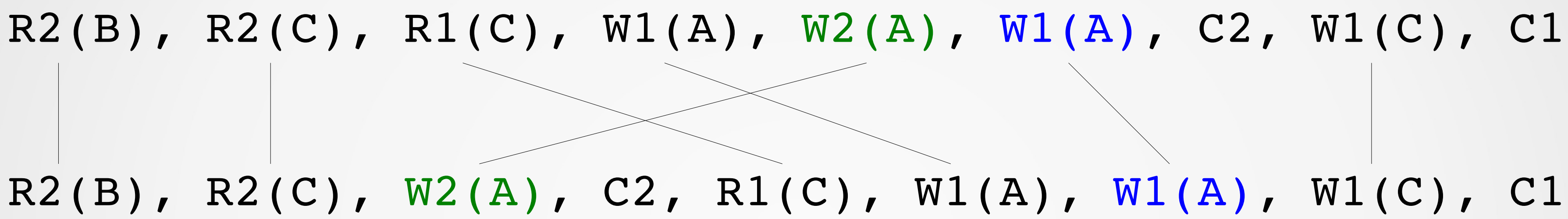
```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1

R2(B), R2(C), W2(A), C2, R1(C), W1(A), W1(A), W1(C), C1
```

These two operations conflict and are in the same order in both schedules



```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1

R2(B), R2(C), W2(A), C2, R1(C), W1(A), W1(A), W1(C), C1
```

These two operations conflict and are in different orders in the two schedules
These schedules are **not** conflict equivalent



```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1

R2(B), R2(C), W2(A), C2, R1(C), W1(A), W1(A), W1(C), C1
```

These two operations conflict and are in the same order in both schedules

Conflict Equivalence Example

- Conclusion: These two schedules are **not** conflict equivalent:

  `R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1`

  and

  `R2(B), R2(C), W2(A), C2, R1(C), W1(A), W1(A), W1(C), C1`

- As we saw earlier, the schedules **are** view equivalent

Serializability

Serial Schedule

- A schedule is *serial* when it consists of a sequence of transactions one after the other with no interleaving
- Examples

  R1(A), R1(B), W1(B), W1(A), C1, R2(B), W2(A), R2(D), W2(B), C2

  R2(B), W2(A), R2(D), W2(B), C2, R1(A), R1(B), W1(B), W1(A), C1
- If there are n transactions, then there are n! serial schedules

Serializable

- A schedule is *serializable* if it is equivalent to a serial schedule
- There are two notions of serializability: view and conflict
- Warning: There are other meanings of the word “serializable”. Be careful not to confuse them!

Checking View Serializability

- If there are n transactions, then there are n! serial schedules
- Many of them can be eliminated by considering some of the conflicts
- Be careful! Not all conflicts can be used for this purpose

Checking View Serializability

- RW Conflicts
  - IF the schedule has this form:

    `...Ri(X)...Wj(X)...`
  - AND `Wj(X)` is the first write to data item `X`
  - THEN an equivalent serial schedule must have Transaction i before Transaction j

Checking View Serializability

- WR Conflicts
  - IF the schedule has this form:

    `...Wi(X)...Rj(X)...`
  - AND there are no other writes of `X` between these two operations
  - THEN an equivalent serial schedule must have Transaction i before Transaction j

Checking View Serializability

- WW Conflicts
  - IF the schedule has this form:

    `...wi(X)...wj(X)...`
  - AND `wj(X)` is the last write operation on `X`
  - THEN an equivalent serial schedule must have Transaction i before Transaction j

Checking View Serializability

- Use conflicts to eliminate permutations of the transactions
- Check whether any of the remaining serial schedules are view equivalent to the original schedule
- If one of the serial schedules is view equivalent, then one can stop: The schedule is serializable

View Serializable Example

- Is this schedule view serializable?

  `R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1`

- The conflicts are:

  `R2(C) W1(C)` implies that T2 precedes T1

  `W1(A) W2(A)` cannot be used

  `W2(A) W1(A)` implies that T2 precedes T1

- We checked that the schedule is view equivalent to the serial schedule in the order T2, T1.

Checking Conflict Serializability

- Construct the Transaction Conflict Graph
  - For each conflict between Opi(A) and Opj(A), draw a directed edge from Transaction i to Transaction j
  - The schedule is serializable if and only if the Transaction conflict graph is acyclic
- Any topological sort of the Transaction Conflict Graph gives a serial schedule that is equivalent to the original schedule
- This is much easier than checking view serializability

Conflict Serializable Example

- Is this schedule conflict serializable?

```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2,
W1(C), C1
```

- The conflicts are:

```
R2(C) W1(C)

W1(A) W2(A)

W2(A) W1(A)
```

- The transaction conflict graph is:

$$\begin{tikzcd}
T1 \arrow[r, bend left] & T2 \arrow[l, bend left]
\end{tikzcd}$$

Conflict Serializable Example

- Is this schedule conflict serializable?

```
R2(B), R2(C), R1(C), W1(A), W2(A), W1(A),
C2, W1(C), C1
```

- The transaction conflict graph has a cycle, so the schedule is **not** serializable

View vs Conflict Serializability

- A view serializable schedule is obviously correct
  - The effect of the schedule on read operations and on the database is the same as for a serial schedule
- Why bother with conflict serializability?
- Conflict serializability implies view serializability
- Conflict serializability is much easier to check and to enforce
  - Checking view serializability is an NP-complete problem
  - Checking conflict serializability is polynomial time

Recoverability Criteria

Recoverability Criteria

- Strictness is concerned with whether a schedule is *vulnerable* to failures of various kinds
- There are three vulnerability criteria
  - Recoverability
  - Cascadelessness
  - Strictness

Recoverable

- A schedule is *recoverable* if the database system can recover from a failure at any point in the schedule
- To check whether a schedule is recoverable one must look at the WR conflicts
- A schedule is *unrecoverable* if it has this pattern:

```
… Wi(X) … Rj(X) … Cj … Ci …
```

- A schedule is *recoverable* if this pattern never occurs
- The pattern is called a *violation of recoverability*

Importance of Recoverability

- Why is recoverability important?
- If there is a violation of recoverability

  `… Wi(X) … Rj(X) … Cj … Ci …`

  Then the database is vulnerable between `Cj` and `Ci` because a failure between these two operations will cause transaction i to be rolled back but transaction j used a value of the data item `X` that was rolled back (i.e., never happened)
- This scenario is a **disaster**
  - Transaction j is invalid but cannot be rolled back
  - The database is now corrupted and cannot be easily corrected

Cascadeless

- A schedule is *cascadeless* if the failure of one transaction cannot cause another one to fail
- To check whether a schedule is casecadeless one must look at the WR conflicts
- A schedule allows cascading rollback if it has this pattern:

```
… Wi(X) … Rj(X) … Ci …
```

- A schedule is *cascadeless* if this pattern never occurs
- The pattern is called a *violation of cascadelessness*
- If a schedule is cascadeless then it is recoverable

Importance of Cascadelessness

- Why is cascadelessness important?
- If there is a violation of cascadelessness

```
… Wi(X) … Rj(X) … Ci …
```

  Then the database is vulnerable between `Rj(X)` and `Ci` because a failure of transaction i between these two operations will cause both transaction i and transaction j to be rolled back
- In other words, the failure of Transaction i *cascades* to a failure of Transaction j, which could, in turn, cause still other transactions to fail
- While this may not be a disaster, it does have a significant impact on performance

Strict

- A schedule is *strict* if it has no violations of strictness
- A *violation of strictness* is one of the following patterns:

```
… Wi(X) … Rj(X) … Ci …
… Wi(X) … Wj(X) … Ci …
```

- A strict schedule is cascadeless and recoverable
- The importance of strictness is that it is much easier to enforce strictness than to enforce cascadelessness and recoverability

Checking Recoverability

- Consider the earlier example:

  `R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1`

- There are no WR conflicts, so the schedule is recoverable and cascadeless
- There are two WW conflicts

  `R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1`

  violates strictness

  `R2(B), R2(C), R1(C), W1(A), W2(A), W1(A), C2, W1(C), C1`

  violates strictness

Example Schedule

R3(C), R3(B), R1(C), W1(C), W2(A), C1, R2(A), R2(C), W3(A), C3, C2

Determine whether the schedule is view-serializable, conflict-serializable, recoverable, cascadeless and/or strict. Note: you must state *separately* whether each one of these holds.

For each of the 5 answers in the first part above, explain why or why not. In the case of the two notions of serializability, you must either give an equivalent serial schedule or a reason why the schedule is not serializable. For the other three concepts, you must give an example of a violation if the schedule does not have the property.

Example Schedule

R2(A), W2(A), R3(C), R2(A), W2(B), R2(B), W3(A), W2(A), R1(C), C2, R1(B), W3(A), C1, C3

Determine whether the schedule is view-serializable, conflict-serializable, recoverable, cascadeless and/or strict. Note: you must state *separately* whether each one of these holds.

For each of the 5 answers in the first part above, explain why or why not. In the case of the two notions of serializability, you must either give an equivalent serial schedule or a reason why the schedule is not serializable. For the other three concepts, you must give an example of a violation if the schedule does not have the property.

Guaranteeing Serializability

Guaranteeing serializability

- There are two main techniques for ensuring serializability
  - Blocking Protocols
  - Nonblocking Protocols
- There are also techniques that combine these two, having both blocking and nonblocking features
- Blocking protocols make use of locks
- Nonblocking protocols make use of timestamping

Locks

- A *lock* is an option to perform an operation on a data item
- Before any operation is performed, a transaction must acquire the right to perform it
- A lock manager is a server that is responsible for giving transactions the rights necessary for their operations
- If a transaction requests a lock that conflicts with the existing locks on the data item, then the transaction must wait
  - This is why using locks is called “blocking”

Separation of Concerns

- Lock management is an example of separation of concerns
- The transactions can perform their work without any concern about other transactions that might be using the same data items
- The lock manager knows nothing about what transactions are doing or about the structure of a data item
  - The lock manager is only concerned with granting and releasing locks

Lock Types

- There are many lock types, and it is easy to add more of them
- The main lock types are Shared (S) and Exclusive (X)
- An S lock gives a transaction the right to read a data item
- An X lock gives a transaction the right to both read and write a data item
- If a transaction has an X lock then no other transaction has a lock on the data item of any kind
- More than one transaction can have an S lock on a data item

Deadlocks

- Blocking protocols are vulnerable to wait cycles, better known as deadlocks
- When a deadlock occurs, one of the transactions must be rolled back to allow the other transactions in the wait cycle to continue
- One can detect deadlocks by maintaining the waits-for graph and searching for cycles
- More commonly, deadlocks are handled by specifying a timeout period: the maximum amount of time that a transaction will wait for a lock

Two Phase Locking

- This is a protocol for how locks are acquired and released
- To read a data item, a transaction must acquire an S lock
- To write a data item, a transaction must acquire an X lock
- If a lock of any kind is released, then no other locks may be acquired
- The transaction is in two phases: lock acquisition and lock release
- If the two-phase locking protocol is followed, then the schedule is serializable

Strong Strict Two Phase Locking

- Two Phase Locking ensures serializability but it does not ensure recoverability or cascadelessness
- Strong Strict Two Phase Locking does not release any locks until the transaction commits
  - All locks are released as part of commit processing
  - Ensures strictness and so also recoverability and cascadelessness

Nonblocking Protocols

- There are many nonblocking protocols
- No locks are set
  - Sometimes called optimistic concurrency control
- Every data item is timestamped
  - Time of most recent access (read)
  - Time of most recent update (write)
- At the commit, the timestamps are examined to determine if serializability is violated
- If there is a violation, then the transaction is rolled back

Blocking versus Nonblocking

- Blocking requires a lock manager
  - The lock manager can be a bottleneck
- Both require rollbacks
  - Blocking requires rollbacks when deadlocks occur
  - More frequent for nonblocking protocols
- Blocking protocols are much more easily extended
  - Add another lock type
  - Nonblocking protocols must be redeveloped
- Blocking protocols are more commonly used

Specifying Transactions

Isolation Levels

- Not all transactions require the same level of isolation from other transactions
- Read Uncommitted. Lowest isolation level. Writes are not allowed, and no locks are set
- Read Committed. Next higher level. X locks are held until commit time. S locks are set but released after the read is complete
- Repeatable Read. Both X and S locks are held until commit time
- Serializable. Full serializability

Phantoms

- The abstract model used for concurrency control assumes that the set of data items is fixed
- In fact, data items can be created and destroyed
- Consider this example

  One transaction is computing the average of a column of a table

  Another transaction adds new row to the table concurrently with the first transaction

  The first transaction locked the rows in the table when the transaction started

  The row being added did not exist when the

Phantom Example

- Transaction 1 is computing the average grade of a student
- Transaction 2 adds a new grade concurrently
- Transaction 1 locked the rows in the table when the transaction started
  - But the new grade was not locked because it did not exist
- The new grade is a *phantom*
- Protecting against phantoms is necessary for full serializability

Phantom Protection

- Requires new kinds of locks
- Range locks
  - Locks all rows in a range
  - Requires a B-tree index
  - Special kind of predicate lock
- Hierarchical locking
  - Allows locking at different levels of granularity
  - Locking a single record requires several lock manager requests

Other Lock Modes

- Update locks
  - Indicates that one is going to read but may be updating
- Intention locks
  - Gives permission to request another lock
  - Used by hierarchical locking protocols
  - IS Intention to lock in S mode at finer granularity
  - IX Intention to lock in X mode at finer granularity
  - SIX S mode + IX mode

Lock Compatibility Matrix

	S	X	U	IS	SIX	IX
S	yes	no	yes	yes	no	no
X	no	no	no	no	no	no
U	yes	no	no	yes	no	no
IS	yes	no	yes	yes	yes	yes
SIX	no	no	no	yes	no	no
IX	no	no	no	yes	no	yes

Lock Compatibility Matrix

- The current mode is in the left column
- The requested mode is the mode in the first row
- The entry in the matrix says whether to allow the requested mode
- One can add a new lock mode by adding another row and column to the lock compatibility matrix

Preventing Deadlocks

- The most common deadlock
  - One data item
  - Two transactions are attempting to promote from S to X
  - This is what happens in the “Lost Update” scenario
- The U lock prevents this deadlock
  - It is like the S lock in only giving permission to read
  - However, it is incompatible with another U lock
- Specify the U lock by adding “for update” as the last clause of your SQL query

Savepoints

- A *savepoint* is a mark in the sequence of operations in the transaction used for limiting a rollback
  - One can rollback to a savepoint rather than the entire transaction
- The `setSavepoint` method sets a savepoint
- The rollback method can specify a savepoint

Metadata Management

Purpose

- Metadata is data about data
- The data in a repository or database is called the “content” to distinguish it from the metadata
- Metadata is also data but at a higher level than content
- Metadata can also have metadata
  - There can be any number of levels
  - Each level is a *model* or *schema* for the level below it
- The purpose of metadata is to specify the structure and constraints of the content
  - The most powerful schemas are called *ontologies*

Data Dictionary

- The metadata for a relational database is called its data dictionary
- The information about the tables, their columns and their relationships is modeled like any other data
  - For example, a table is a row in the Table table, a column is a row in the Column table, etc.
- For MySQL the information_schema has the metadata for all of the databases of the database system
  - It is easy to examine the information_schema using the mysql client or a MySQL workbench

MySQL Data Dictionary

```
CHARACTER_SETS
COLLATIONS
COLLATION_CHARACTER_SET_APPLICABILITY
COLUMNS
COLUMN_PRIVILEGES
ENGINES
EVENTS
FILES
GLOBAL_STATUS
GLOBAL_VARIABLES
KEY_COLUMN_USAGE
OPTIMIZER_TRACE
PARAMETERS
PARTITIONS
PLUGINS
```

MySQL Data Dictionary

```
PROCESSLIST
PROFILING
REFERENTIAL_CONSTRAINTS
ROUTINES
SCHEMATA
SCHEMA_PRIVILEGES
SESSION_STATUS
SESSION_VARIABLES
STATISTICS
TABLES
TABLESPACES
TABLE_CONSTRAINTS
TABLE_PRIVILEGES
TRIGGERS
USER_PRIVILEGES
VIEWS
```

MySQL Tables Table

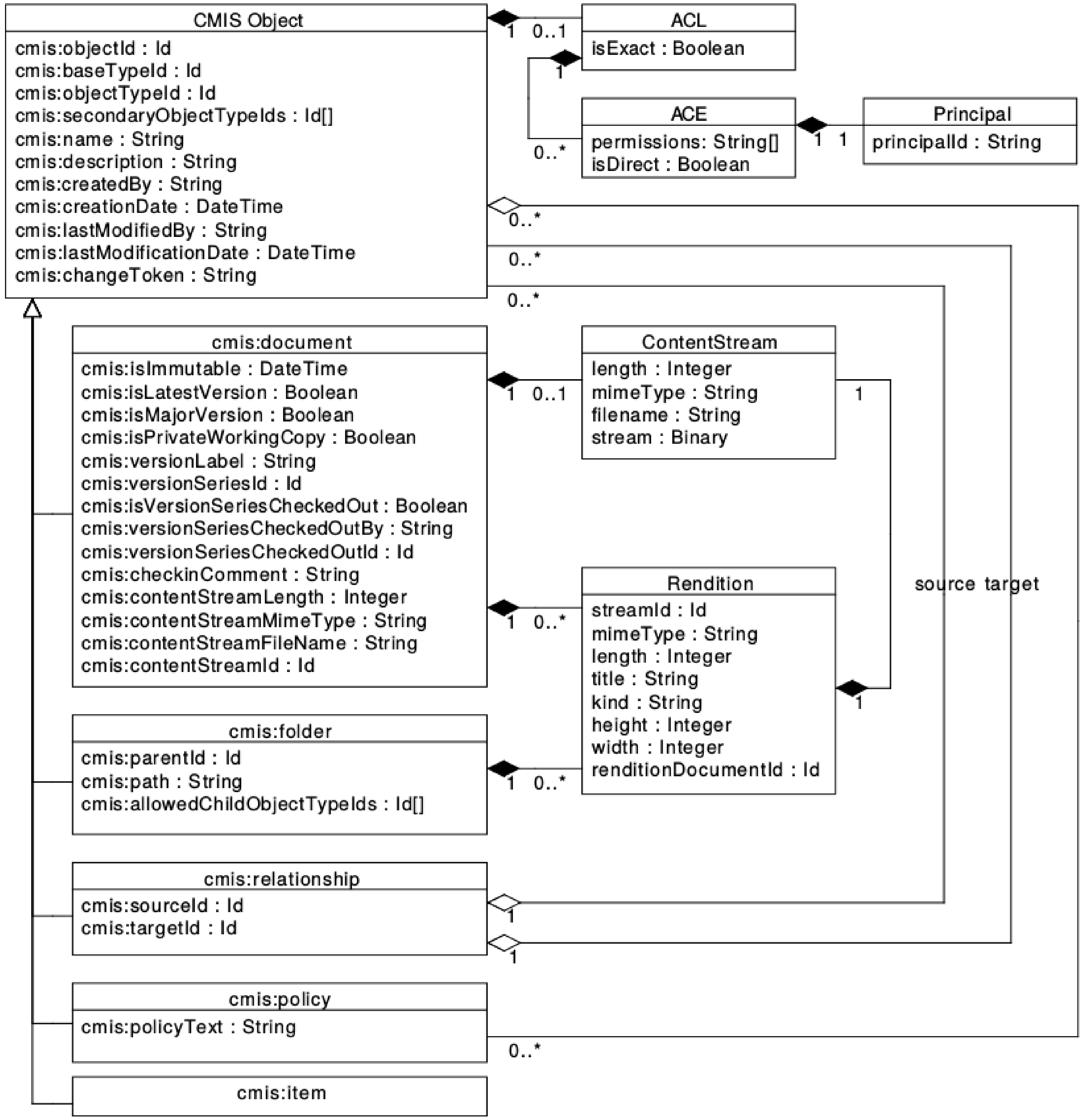
```
TABLE_CATALOG  varchar(512)   NO
TABLE_SCHEMA   varchar(64) NO
TABLE_NAME varchar(64) NO
TABLE_TYPE varchar(64) NO
ENGINE varchar(64) YES    NULL
VERSION bigint(21) unsigned   YES    NULL
ROW_FORMAT varchar(10) YES    NULL
TABLE_ROWS bigint(21) unsigned   YES    NULL
AVG_ROW_LENGTH bigint(21) unsigned   YES    NULL
DATA_LENGTH bigint(21) unsigned   YES    NULL
MAX_DATA_LENGTH    bigint(21) unsigned   YES    NULL
INDEX_LENGTH   bigint(21) unsigned   YES    NULL
DATA_FREE  bigint(21) unsigned   YES    NULL
AUTO_INCREMENT bigint(21) unsigned   YES    NULL
CREATE_TIME datetime   YES    NULL
UPDATE_TIME datetime   YES    NULL
CHECK_TIME datetime   YES    NULL
TABLE_COLLATION    varchar(32) YES    NULL
CHECKSUM   bigint(21) unsigned   YES    NULL
CREATE_OPTIONS varchar(255)   YES    NULL
TABLE_COMMENT  varchar(2048)  NO
```

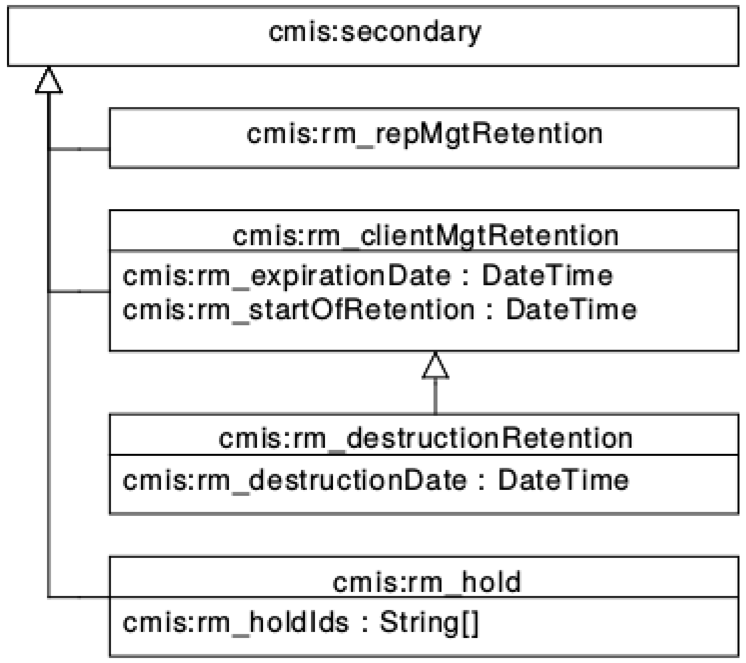
MySQL Columns Table

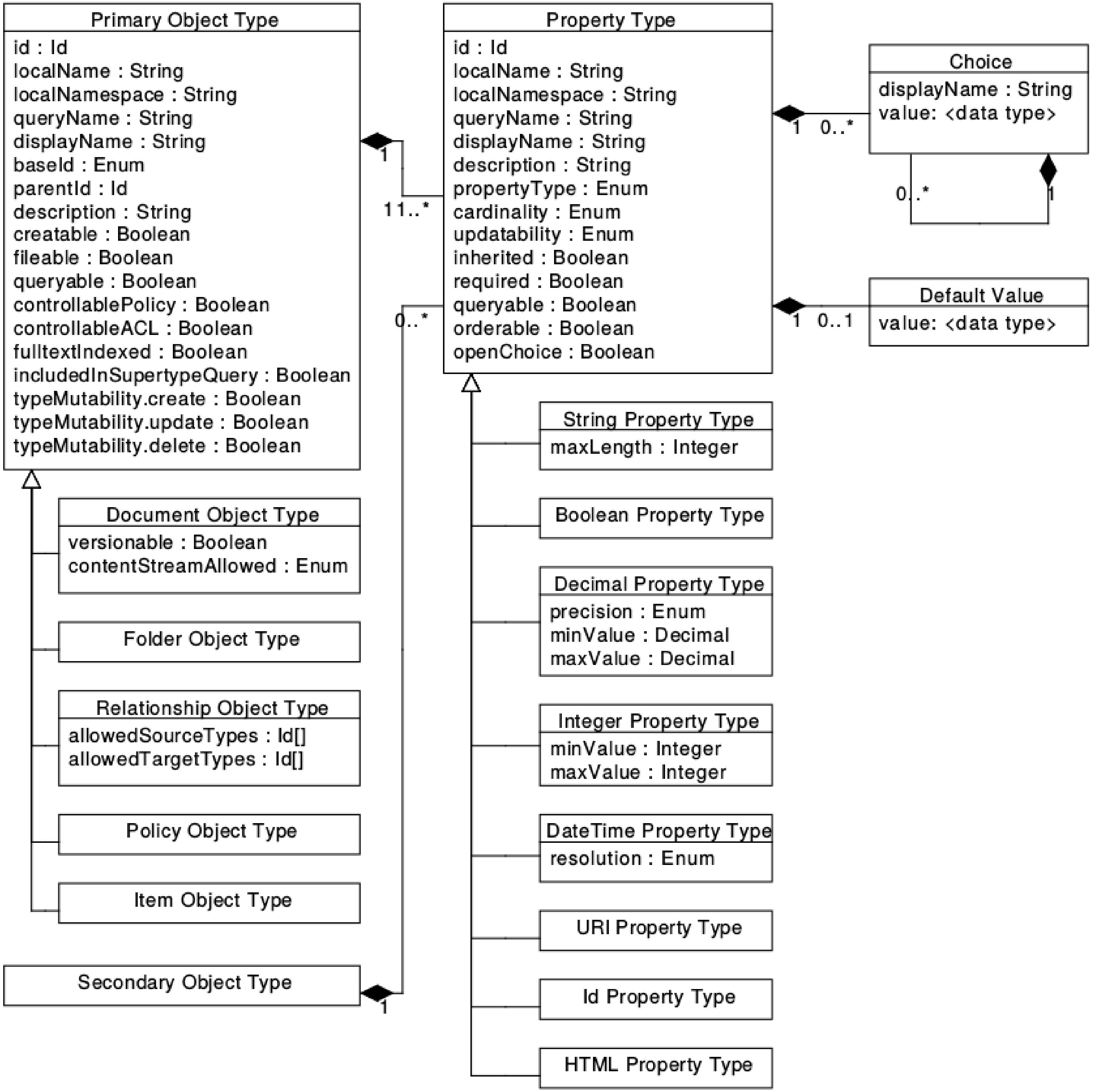
```
TABLE_CATALOG   varchar(512)   NO
TABLE_SCHEMA    varchar(64)NO
TABLE_NAME  varchar(64)NO
COLUMN_NAMEvarchar(64)NO
ORDINAL_POSITION   bigint(21) unsigned    NO      0
COLUMN_DEFAULT longtext    YES     NULL
IS_NULLABLEvarchar(3) NO
DATA_TYPE   varchar(64)NO
CHARACTER_MAXIMUM_LENGTH   bigint(21) unsigned    YES     NULL
CHARACTER_OCTET_LENGTHbigint(21) unsigned    YES     NULL
NUMERIC_PRECISION  bigint(21) unsigned    YES     NULL
NUMERIC_SCALE  bigint(21) unsigned    YES     NULL
DATETIME_PRECISION bigint(21) unsigned    YES     NULL
CHARACTER_SET_NAME varchar(32)YES     NULL
COLLATION_NAME varchar(32)YES     NULL
COLUMN_TYPElongtext    NO      NULL
COLUMN_KEY  varchar(3) NO
EXTRA   varchar(30)NO
PRIVILEGES  varchar(80)NO
COLUMN_COMMENT varchar(1024)  NO
GENERATION_EXPRESSION  longtext    NO      NULL
```

Content Management Systems

- CMIS has a metadata level
  - Serves as a data dictionary for a content management system
  - Allows one to extend the data model
- Customization of content management systems


CMIS Object
cmis:objectId : Id
cmis:baseTypeId : Id
cmis:objectTypeId : Id
cmis:secondaryObjectTypeIds : Id[]
cmis:name : String
cmis:description : String
cmis:createdBy : String
cmis:creationDate : DateTime
cmis:lastModifiedBy : String
cmis:lastModificationDate : DateTime
cmis:changeToken : String
ACL
isExact : Boolean
ACE
permissions: String[]
isDirect : Boolean
Principal
principalId : String
1
0..1
1
0..*
1
1
0..*
0..*
0..*
cmis:document
cmis:isImmutable : DateTime
cmis:isLatestVersion : Boolean
cmis:isMajorVersion : Boolean
cmis:isPrivateWorkingCopy : Boolean
cmis:versionLabel : String
cmis:versionSeriesId : Id
cmis:isVersionSeriesCheckedOut : Boolean
cmis:versionSeriesCheckedOutBy : String
cmis:versionSeriesCheckedOutId : Id
cmis:checkinComment : String
cmis:contentStreamLength : Integer
cmis:contentStreamMimeType : String
cmis:contentStreamFileName : String
cmis:contentStreamId : Id
ContentStream
length : Integer
mimeType : String
filename : String
stream : Binary
1
0..1
1
Rendition
streamId : Id
mimeType : String
length : Integer
title : String
kind : String
height : Integer
width : Integer
renditionDocumentId : Id
1
0..*
1
source
target
cmis:folder
cmis:parentId : Id
cmis:path : String
cmis:allowedChildObjectTypeIds : Id[]
1
0..*
cmis:relationship
cmis:sourceId : Id
cmis:targetId : Id
1
1
cmis:policy
cmis:policyText : String
0..*
cmis:item


cmis:secondary
cmis:rm_repMgtRetention
cmis:rm_clientMgtRetention
cmis:rm_expirationDate : DateTime
cmis:rm_startOfRetention : DateTime
cmis:rm_destructionRetention
cmis:rm_destructionDate : DateTime
cmis:rm_hold
cmis:rm_holdIds : String[]



Primary Object Type
id : Id
localName : String
localNamespace : String
queryName : String
displayName : String
baseId : Enum
parentId : Id
description : String
creatable : Boolean
fileable : Boolean
queryable : Boolean
controllablePolicy : Boolean
controllableACL : Boolean
fulltextIndexed : Boolean
includedInSupertypeQuery : Boolean
typeMutability.create : Boolean
typeMutability.update : Boolean
typeMutability.delete : Boolean
Document Object Type
versionable : Boolean
contentStreamAllowed : Enum
Folder Object Type
Relationship Object Type
allowedSourceTypes : Id[]
allowedTargetTypes : Id[]
Policy Object Type
Item Object Type
Secondary Object Type
1
1
1..*
0..*
1
Property Type
id : Id
localName : String
localNamespace : String
queryName : String
displayName : String
description : String
propertyType : Enum
cardinality : Enum
updatability : Enum
inherited : Boolean
required : Boolean
queryable : Boolean
orderable : Boolean
openChoice : Boolean
Choice
displayName : String
value: <data type>
1
0..*
0..*
1
Default Value
value: <data type>
1
0..1
String Property Type
maxLength : Integer
Boolean Property Type
Decimal Property Type
precision : Enum
minValue : Decimal
maxValue : Decimal
Integer Property Type
minValue : Integer
maxValue : Integer
DateTime Property Type
resolution : Enum
URI Property Type
Id Property Type
HTML Property Type

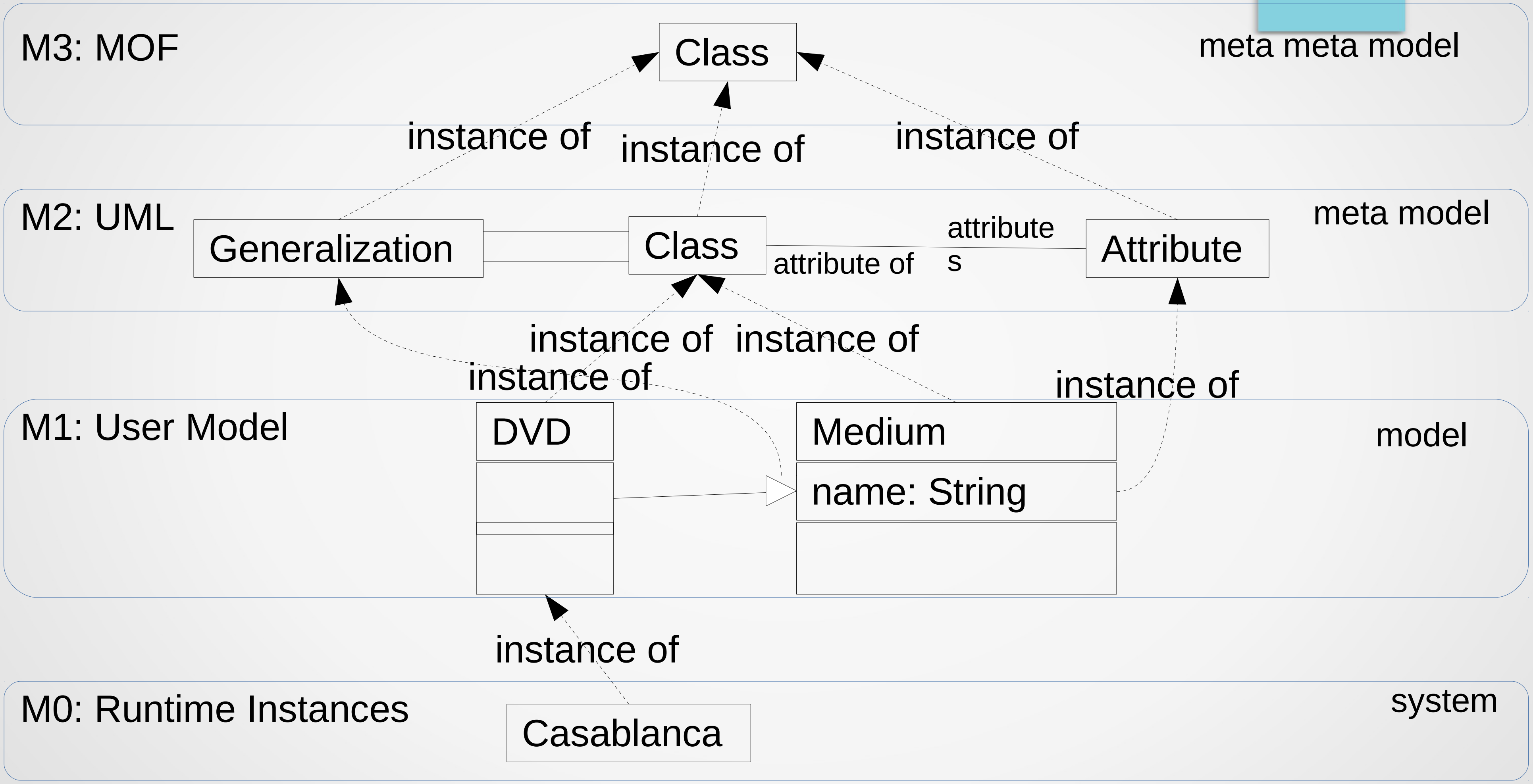
UML

- UML metadata is modeled by the Meta-Object Facility (MOF)
- MOF has four layers
  - More than other data modeling languages
- One can model other modeling languages in MOF
- Used by the Model-Driven Architecture (MDA)
  - Popular technique for developing software that is easily customizable for various platforms and languages

MOF Layers

- M3 is the top layer and only has MOF
- M2 is the language layer
  - UML is specified on this layer
  - The relational model is specified on this layer
- M1 is the model layer
  - A specific UML diagram is on this layer
  - A specific SQL schema is on this layer
- M0 is the data layer
  - Content and data is on this layer
- The relationship between layers is *instance of*

MOF Layers


M3: MOF
Class
meta meta model
instance of
instance of
instance of
M2: UML
Generalization
Class
attribute
s
attribute of
Attribute
meta model
instance of
instance of
instance of
instance of
M1: User Model
DVD
Medium
name: String
model
instance of
M0: Runtime Instances
Casablanca
system

Metadata using JDBC

- DatabaseMetadata
  - Metadata about the database server
  - Many methods
  - Some of them return a ResultSet
    - For example, getTables
- Database systems are highly customizable
  - DatabaseMetadata shows the customized values

Metadata using JDBC

- ResultSetMetadata
  - Metadata about the columns of a result set
  - Many methods
  - Should not be needed if you programmed the query
- ParameterMetadata
  - Metadata about the parameters of a prepared statement
  - Should not be needed if you programmed the query

Assignment #9