Concurrency Control / Serializability Theory

Correctness of a program can be characterized by invariants

- **Invariants over state**

```
Linked list:
  For all items I where I.prev != NULL: I.prev.next == I
  For all items I where I.next != NULL: I.next.prev == I
```

- **Invariants over operations**

```
Stack:
  Stack.size after push == Stack.size before push + 1
  Stack.size after pop  == Max (Stack.size before pop - 1, 0)
```

Consistency can be defined as validity of invariants

Non atomic actions temporarily violate invariants, hence program moves from consistent to consistent state through potentially inconsistent states

Running programs in parallel means inconsistent states can be observed and acted upon, which is what we need to prevent. This is the purpose of concurrency control.

Modeling Transactions

Transaction is a sequence of read and write actions on data …

- **A transaction T_i is a partial order with ordering relation $<_i$ where**
  - **$T_i \subseteq \{ r_i[x], w_i[x] \mid x$ is a data item $\} \cup \{ a_i, c_i \}$;**
  - **$a_i \in T_i$ iff $c_i \notin T_i$;**
  - **if t is c_i or a_i, $t \in T_i$, for any other operation $p \in T_i$, $p < t$;**
  - **if $r_i[x], w_i[x] \in T_i$ then either $r_i[x] < w_i[x]$ or $w_i[x] < r_i[x]$.**
- **Uninterpreted features – we do not make any assumptions**
- **Transactions are drawn as direct acyclic graphs (DAGs)**

Execution History

Let's model execution history from transactions, but we need to define conflicting operations

- **Conflicting operations operate upon the same data and at least one of them is write**
- **Conflicting transactions contain conflicting operations**

Execution history

- **$T = \{T_1, T_2, \ldots, T_n\}$ is a set of transactions**
- **Complete history is a partial order with ordering relation $<_H$ where**
  1. **$H = \bigcup_n^{i=1} T_i$**
  2. **$<_H \supseteq \bigcup_n^{i=1} <_i$**
  3. **for any two conflicting operations $p, q \in H$, either $p <_H q$ or $q <_H p$**
- **History is simply a prefix of complete history**

As well as with particular transactions, we draw history as a DAG
We don't draw all arrows implied by transitivity

Committed, aborted, and active transactions in history H

- **Committed projection C(H) of H contains only operations of committed transactions**

Serializable Histories

History is serializable if it is equivalent to a serial history

- **H and H' equivalent if**
  - **they have the same transactions and operations**
  - **they order conflicting operations of non-aborted transactions in the same way**

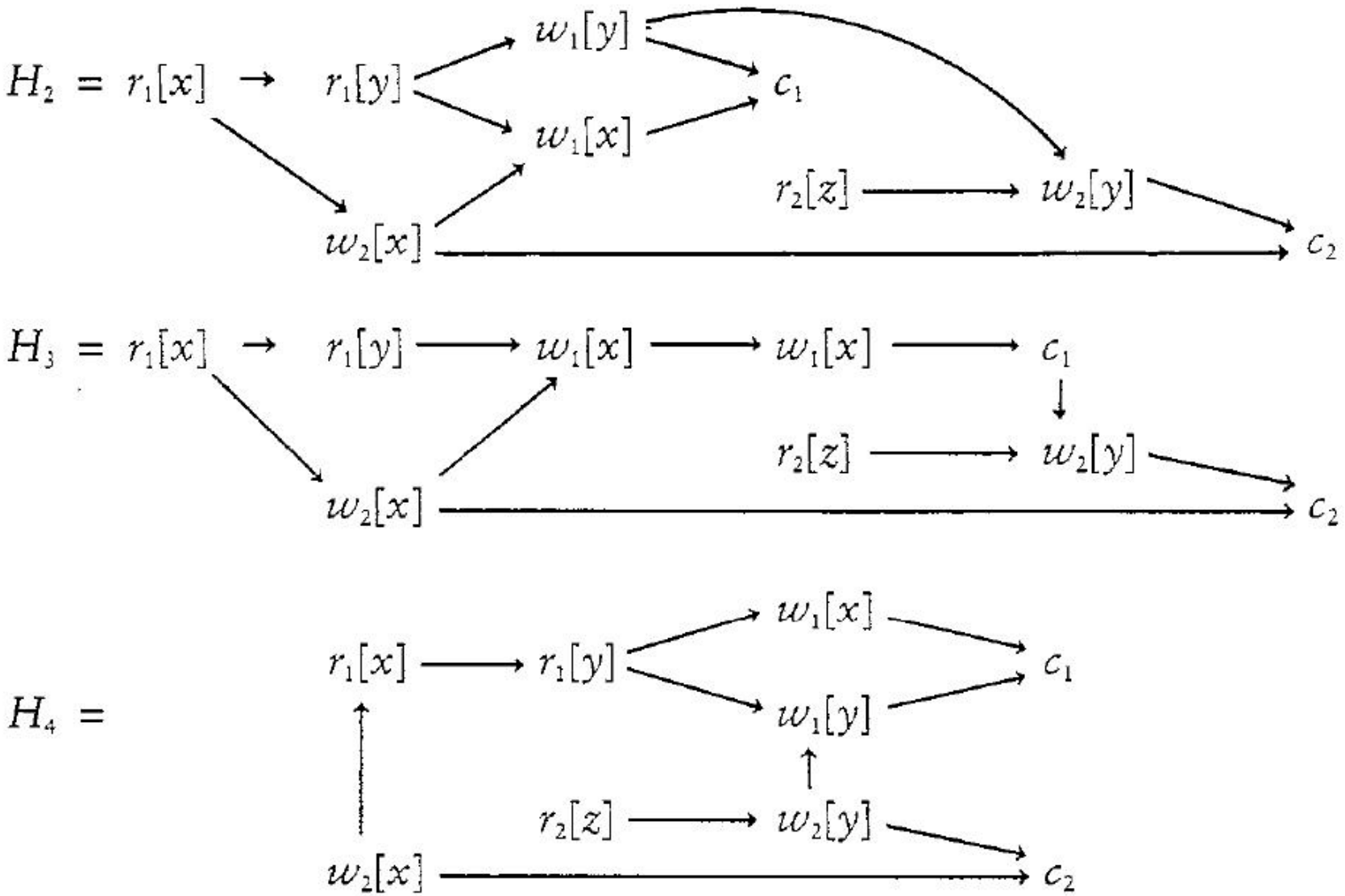
The outcome of concurrent execution of transactions depends on the relative ordering of conflicting operations
Executing two non-conflicting operations in either order has the same computational effect

- **Complete history H is serial if for every two transactions T_i, $T_j \in$ H all operations of T_i are before operations of T_j or vice versa**

We require completeness of history since incomplete transactions do not preserve data consistency

- **History is serializable (SR) if its committed projection is equivalent to a serial history**

Only execution of committed transactions is guaranteed


$H_2 = r_1[x] \rightarrow r_1[y]$
$w_1[y]$
$w_1[x]$
c_1
$r_2[z] \rightarrow w_2[y]$
$w_2[x]$
c_2
$H_3 = r_1[x] \rightarrow r_1[y] \rightarrow w_1[x] \rightarrow w_1[x] \rightarrow c_1$
$r_2[z] \rightarrow w_2[y]$
$w_2[x]$
c_2
$H_4 =$
$r_1[x] \rightarrow r_1[y]$
$w_1[x]$
$w_1[y]$
c_1
$r_2[z] \rightarrow w_2[y]$
$w_2[x]$
c_2

Serializability Theorem

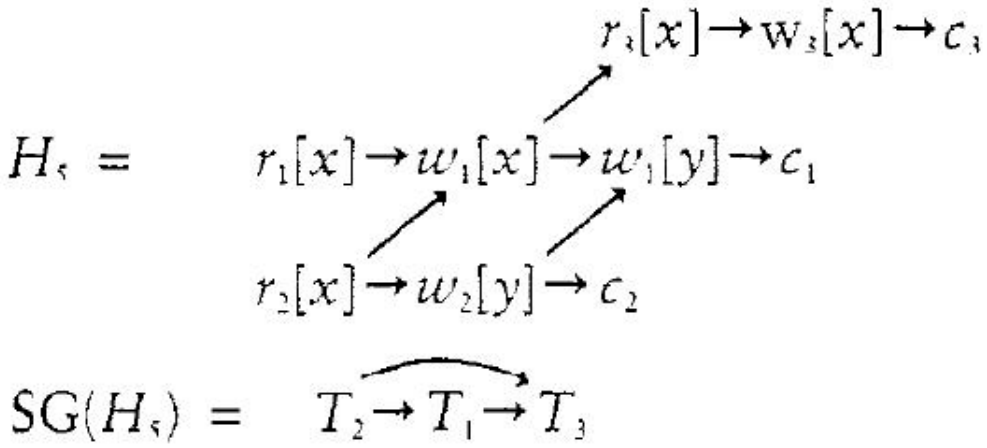
How we recognize serializable history

- **For H a history over T = {T_1, T_2, …, T_n }, serialization graph SG(H) is directed graph**
  - **nodes are transactions in T that are committed in H**
  - **edges are $T_i \rightarrow T_j$ if one of T_i's operations conflict and precedes with one of T_j's operations in H**

Single edge for more than one pair of conflicting operations
No transitivity
Edges between transactions implies ordering in a serial history
We can find an equivalent serial history if SG(H) is acyclic

Serializability theorem

- **A history H is serializable iff SG(H) is acyclic**


$r_3[x] \rightarrow w_3[x] \rightarrow c_3$
$H_5 =$
$r_1[x] \rightarrow w_1[x] \rightarrow w_1[y] \rightarrow c_1$
$r_2[x] \rightarrow w_2[y] \rightarrow c_2$
$SG(H_5) =$
$T_2 \rightarrow T_1 \rightarrow T_3$

Recoverable Histories

To ensure correctness in the presence of failures, the scheduler must produce SR histories that are recoverable

- **T_i reads x from T_j if**
  - **$w_j[x] < r_i[x]$**
  - **a_j does not precede $r_i[x]$ in the partial order**
  - **if there is some $w_k[x]$ such that $w_j[x] < w_k[x] < r_i[x]$ then $a_k < r_i[x]$**

- **T_i reads from T_j if there is an x such that T_i reads x from T_j**

  A transaction can read a data item from itself

- **H is recoverable (RC) if, whenever T_i reads from T_j ($i \neq j$) in H and $c_i \in$ H, then $c_j < c_i$**

  Intuitively, if T_i reads x from T_j, it has to wait whether x will not be invalidated by a_j

- **H avoids cascading aborts (ACA) if, whenever T_i reads x from T_j ($i \neq j$) in H, then $c_j < r_i[x]$**

  A transaction may read only those values that are written by committed transactions or by itself

- **H is strict (ST) if, whenever $w_j[x] < o_i[x]$ ($i \neq j$), either $a_j < o_i[x]$ or $c_j < o_i[x]$ where $o_i[x]$ is $r_i[x]$ or $w_i[x]$**

  No data item may be read or overwritten until the transaction that previously wrote into it terminates

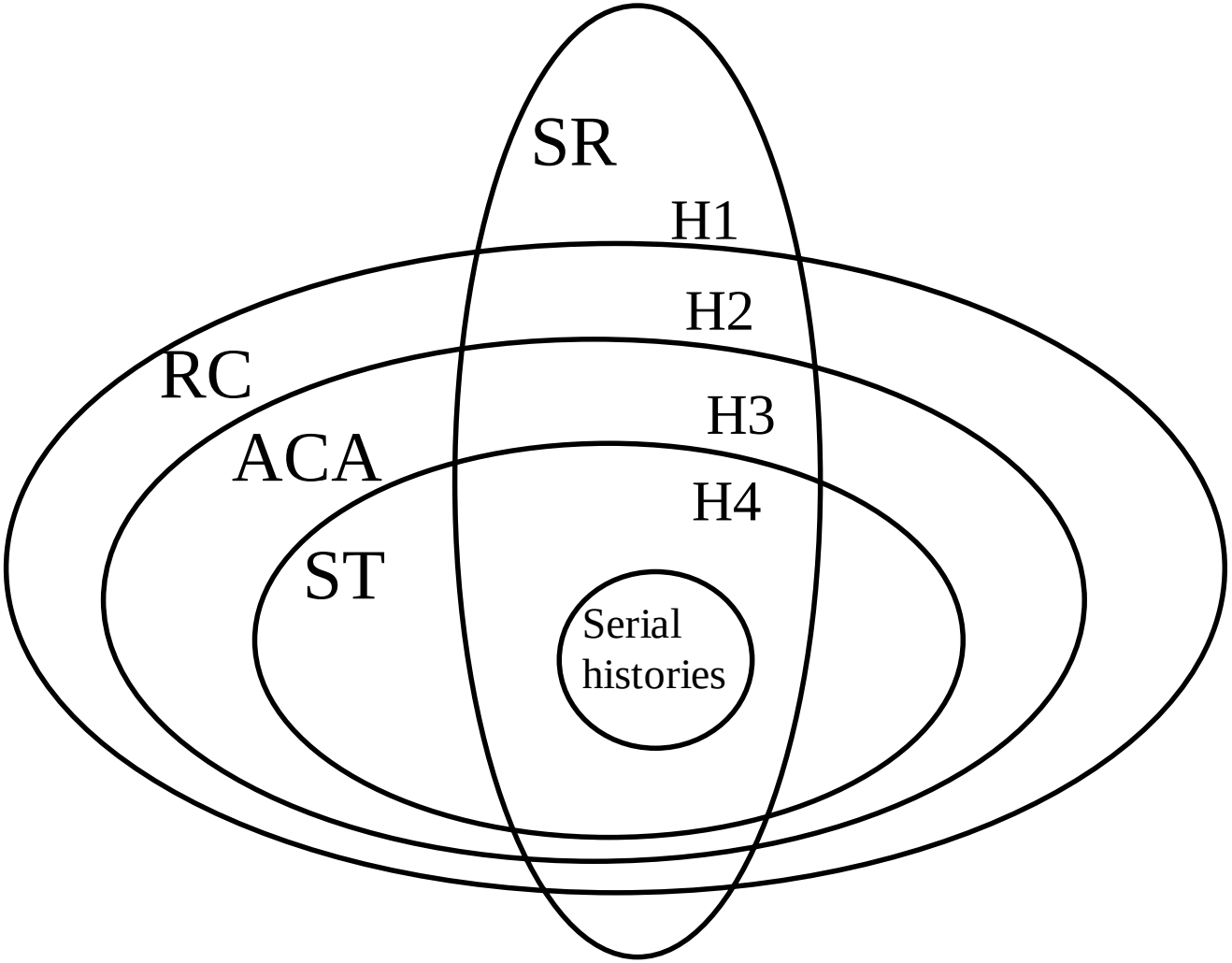
- **Examples**

```
T1 = w1[x] w1[y] w1[z] c1      T2 = r2[u] w2[x] r2[y] w2[y] c2

H1 = w1[x] w1[y] r2[u] w2[x] r2[y] w2[y] c2 w1[z] c1
H2 = w1[x] w1[y] r2[u] w2[x] r2[y] w2[y] w1[z] c1 c2
H3 = w1[x] w1[y] r2[u] w2[x] w1[z] c1 r2[y] w2[y] c2
H4 = w1[x] w1[y] r2[u] w1[x] c1 w2[x] r2[y] w2[y] c2
```

  H1 is not RC (T2 reads y from T1 and c2 < c1)
  H2 is RC but not ACA (T2 reads y from T1 before T1 is committed)
  H3 is ACA but not ST (T2 overwrites value written to x by T1 before T1 terminates)
  H4 is ST

- **ST $\subset$ ACA $\subset$ RC**


SR
H1
H2
RC
H3
ACA
H4
ST
Serial
histories

Prefix Commit-Closed Properties

If a scheduler produces a "correct" history H, then any prefix H' of H should be also correct with respect to committed transactions

In case of DBS fail after H', the database reflects C(H') which should be correct

- **A property is prefix commit-closed if, whenever the property is true of history H, it is also true of history C (H'), for any prefix H' of H**
  - **SR is prefix commit-closed**
  - **RC, ACA, and ST are prefix commit-closed**

Operations Beyond Reads and Writes

Let's consider other set of operations

- **New definition of conflict**
  - **Compatibility matrix for all operations**

- **SG definition unchanged**
- **Serialization theorem unchanged**

We use read and write operations for simplicity and practical usability in databases
Serializability theory works correctly with different operations

View Equivalence

Let's consider other criterion for history equivalence

- **We don't know what computation f(x) is between r[x] and w[x]**
  - **We know that it is some function of all reads**
  - **Thus, if all reads are the same in two histories, then all writes should also be the same**

Consequence:

- **If each transaction reads each of its data items from the same writes in both histories, then all writes write the same values in both histories**
- **If for each x, the final w[x] is the same in both histories, then the final value of all data is the same in both histories**

If both histories leave database in the same final state, then the histories have to be considered equivalent

View Equivalence

More formally:

- **Final write of x in H is $w_i[x] \in H$, such that $a_i \notin H$ and for any $w_j[x] \in H$ ($j \neq i$) either $w_j[x] < w_i[x]$ or $a_j \in H$**

- **Two histories H, H' are equivalent if**
  - **they are over the same set of transactions and have the same operations**
  - **for any T_i, T_j such that a_i, $a_j \notin H$ (hence a_i, $a_i \notin H'$) and for any x, if T_i reads x from T_j in H then T_i reads x from T_j in H'**
  - **for each x if $w_i[x]$ is the final write of x in H then it is also the final write of x in H'**

Call this equivalence view equivalence

- **Used for concurrency control algorithms for multicopy data**
  - **Multiversion concurrency control**
  - **Replicated data concurrency control**

Call the old definition conflict equivalence (conflicting operations of unaborted transactions appear in the same order on both histories)

View Serializability

Recall Conflict Serializability (CSR)

- **H is (conflict) serializable if its committed projection C(H) is (conflict) equivalent to some serial history**

View Serializability (VSR)

- **H is view serializable if for any prefix H' of H, C(H') is view equivalent to some serial history**
  - **We should emphasize "for any prefix" to ensure prefix commit-closed property**

VSR is a (strictly) more inclusive concept then CSR

- **If H is CSR then it is VSR. The converse is not, generally, true**

All practical concurrency control algorithms are conflict-based

An efficient scheduler that produces exactly the set of all view serializable histories can exist only if P=NP
(This would imply that a wide variety of notoriously difficult combinatorial problems would be solvable by efficient algorithms ☺)