Correctness Criterion, Serializability, and Concurrency Control

1. We had left of our discussion of transactions with:
   (a) definition of the page model
   (b) transaction syntax – partial and full orderings
   (c) shuffle histories
   (d) Missing 2 steps in our 5 step process for the page model
      i. Correctness of histories (schedules) – this section
      ii. Algorithms and protocols for implementing correct schedules (next section)

2. General approach to correctness
   (a) fact: transactions executed alone, obey the ACID properties
   (b) implication: transactions executed serially obey the ACID properties
   (c) approach: consider parallel and interleaved schedules based on correlating the effects of a parallel schedule
      with that of a serial schedule

3. What is a scheduler?
   (a) the component of a TP system that generates schedules
   (b) not a time division OS scheduler or a real-time schedule
   (c) orders that atomic actions (reads/writes in the page model) of transactions

4. What does the scheduler order?
   (a) Data operations: reads $r_i(x)$, writes $w_i(x)$
   (b) Termination conditions: abort $a_i$ and commit $c_i$

5. Sidebar on schedules and histories
   (a) histories refer to the retrospective (offline) ordering of transactions
   (b) schedules are prefixes of history and reflect current (on-line) operations orderings
   (c) Histories are useful in analysis, but not in transaction processing itself
   (d) Transaction processing is an online task (WV says highly dynamic) in which decisions need to be made
      immediately; i.e. TP systems don’t look at whole histories in practice, just schedules.
   (e) Histories – termination condition and future data operations known
   (f) Schedule – termination condition and future data operations unknown
   (g) History = “complete” schedule

6. Definition: schedules and histories
   (a) Let $T = \{t_1, \ldots, t_n\}$ be a set of transactions in which each $t_i \in T$ has the form $t_i = (op_i, <_i)$ and with
      $op_i$ denoting the set of operations of $t_i$ and $<_i$ denoting their order
   (b) A history for $T$ is a par $s = (op(s), <_s)$ such that:
      i. $op(s) \subseteq \bigcup_{i=1}^n op_i$ and $\bigcup_{i=1}^n \{a_i, c_i\}$ and $\bigcup_{i=1}^n op_i \subseteq op(s)$
      ii. $(\forall i, 1 \leq i \leq n) c_i \in op(s) \iff a \notin op(s)$
      iii. $\bigcup_{i=1}^n <_i \subseteq <_s$
      iv. $(\forall i, 1 \leq i \leq n)(\forall p \in op_i) p <_s a_i$ or $p <_s c_i$
      v. $(\forall p \in op_s)(\forall q \in op_s)$ if $p = w(x)$ and $q \in \{w(x), r(x)\}$ then either $p <_s q$ or $q <_s p$.  

1
(c) or in English

i. a history applies to all data operations of the component transactions and some outcomes
ii. each transaction has a commit or abort, but not both
iii. An orderings in the component transactions are preserved in the history
iv. the outcome is always the last operation in a transaction
v. all accesses to the same data item, with one being a write, are ordered

(d) A schedule is a prefix of a history

7. A history is serial if for any two transactions \( t_i \) and \( t_j \) with \( i \neq j \), all operations in \( t_i \) are ordered in \( s \) prior to all operations in \( t_j \) or vice versa.

8. The easiest way to visualize schedules (resp. histories) is as a DAG.

(a) Figures 3.1 and 3.2 in WV
(b) Each transaction is a partial order
(c) Partial orders are combined into the \( s \) dag according to the rules of schedules
(d) The \( s \) SAG may still be a partial order

9. Schedulers take partial orders and turn them into total orders – a little notation

(a) \( T = \{t_1, \ldots, t_n\} \) set of transactions then \( \text{shuffle}(T) \) denotes the shuffle product of \( T \) is the set of all sequences s.t. \( t_i \in T \) occurs as a subsequence
(b) a schedule \( s \in \text{shuffle}(T) \)
(c) a history \( s' \) consists of a schedule \( s \in \text{shuffle}(T) \) augmented so that \( \forall t_i \in T \) either \( a_i \in op'_s \) or \( c_i \in op'_s \)
(d) a serial schedule \( s \in \text{shuffle}(T) \) takes the form \( s = t_{\rho(1)} \ldots t_{\rho(n)} \) where \( \rho \) is a permutation of \( 1, \ldots, n \)

10. Correctness criteria

(a) intuition: some conditions that determine which schedules are good or bad. Semantically we want to identify the set of schedules that preserve ACIDity
(b) At the most abstract level for the set of all schedules \( S \) (frequently equal to \( \text{shuffle}(T) \), a correctness criteria is a function \( \alpha : S \leftarrow \{0, 1\} \)
(c) Correct schedules \( \text{correct}(S) := \{s \in S|\alpha(s) = 1\} \)

11. What makes a good correctness criteria

(a) \( \text{correct}(S) \neq \) – there are some legal/correct schedules
(b) \( s \in \text{correct}(S) \) can be decided efficiently, e.g. in P as a minimum requirement
(c) \( \text{correct}(S) \) gives a scheduler a lot of options for parallelism (important quality which is widely considered)

12. Serial equivalence (the fundamental idea behind concurrency control)

(a) We know, by induction on the properties of an individual transaction, that a serial history is correct
(b) Identify “equivalent” schedules that are also correct

13. Skipping view serializability and Herbrand semantics. Getting only to what matters.

(a) Herbrand semantics
   i. allow all operations in a transaction depend on all previous operations in those transaction.
   ii. Operations between transactions are dependent only if one is a write and they manipulate the same data item
iii. Formalization of the arbitrary $f$ function we talked about during transaction definition

(b) View serializability is the natural equivalence class of schedules derived from Herbrand semantics
   i. Solves inconsistent reads and lost updates (dirty writes are a recovery problem)
   ii. Large class – highly concurrent schedules exist
   iii. Computationally infeasible (NP)

(c) General approach will be to find computationally feasible subsets of View serializability
   i. Subset property ensures correctness
   ii. Will be inherently less concurrent

14. Conflict serializability
   (a) Builds on the notion of conflicts that we have been using between writes to the same data item
   (b) Is entirely a syntactic construct (sweet). No data semantics are necessary.
   (c) Is a subset of view serializability (less concurrent, but correct)

15. Conflict relations
   (a) Schedule $s$ with transactions $t, t' \in trans(s)$, $trans(s)$ are all transactions in $s$
      i. two data operations $p \in t$ and $q \in t'$ are in conflict if they access the same data item $x$ and one of
         them is $w(x)$ i.e. $(p = r(x) \land q = w(x)) \lor (p = w(x) \land q = r(x)) \lor (p = w(x) \land q = w(x))$
      ii. $conf(s) : = \{(p, q) | p, q$ are in conflict in $s$ and $p <_s q$ is called the conflict relation of $s$
   (b) Some comments
      i. only data items are in conflict
      ii. does not consider aborts

16. Conflict equivalence
   (a) $s, s'$ are conflict equivalent schedules, denoted $s \approx_c s'$ if they $op(s) = op(s')$ and $conf(s) = conf(s')$
   (b) i.e. all conflicting steps from different transactions are ordered in the same manner

17. Graphical representation of conflict equivalence
   (a) Let $D(s) = \{V, E\}$ where $V = op(s)$, $E = conf(s)$ then $s \approx_c s' \iff D(s) = D(s')$

18. Conflict serializability
   (a) A history $s$ is conflict serializable if there exists a serial history $s'$ s.t. $s \approx_c s'$

19. Example 3.12 – schedule $s$, graph $D(s)$, equivalent serial schedule $s'$ and $D(s')$

20. Testing for conflict serializability
   (a) Construct a serialization graph $G(s) = \{V, E\}$ where $V = commit(s)$ and $(t, t') \in E \iff t \neq t' \land (\exists P \in t)(\exists q \in t')(p, q) \in conf(s)$
   (b) $s \in CSRiffG(s)$ – TODO should we prove this?? maybe
   (c) What this says is that if the commit ordering is consistent with the operation ordering, then the schedule is
       correct
   (d) How do you test for conflict serializability and how long does it take? DFS, $O(V + E) = O(|T|)^2$

21. Commutativity – Another construction for conflict serialization
   (a) Schedules may be incrementally transformed so that they are serially equivalent based on commutativity
       rules.
(b) Idea: any operations not in conflict can be switched in a schedule to make an equivalent schedule
   
i. Rule C1: \( r_i(x)r_j(y) \sim r_j(y)r_i(x) \) for \( i \neq j \)
   
ii. Rule C2: \( r_i(x)w_j(y) \sim w_j(y)r_i(x) \) for \( i \neq j, x \neq y \)
   
iii. Rule C3: \( w_i(x)w_j(y) \sim w_j(y)w_i(x) \) for \( i \neq j, x \neq y \)
   
iv. Applied to adjacent operations only

(c) Constructing schedules with commutativity
   
i. Start with a serial schedule
   
ii. Commute operations of the schedule to achieve concurrent execution

(d) For homework, we will prove that commutativity is serial equivalent, i.e. \( s \approx_x s' \) if \( s \sim s' \) or Commute \( \equiv \) CSR
   
i. Prove containment by induction on incremental transformations are legal, Commute \( \subseteq \) CSR
   
ii. Prove CSR \( \subseteq \) Commute by showing that all transformations not allowed in Commute are conflicts in CSR.