COMP90020: Distributed Algorithms

11. Optimistic Concurrency Control

Trust Everyone to be Nice

Miquel Ramirez

THE UNIVERSITY OF MELBOURNE

Semester 1, 2019

Agenda

Agenda

1. Cascading Aborts and Recoverability
2. Optimistic Concurrency Control
3. Summary
4. Biblio & Reading

Revisiting Drawbacks of Locking

With locking we get the problem of deadlocks

- Preempting and breaking deadlocks reduces concurrency,
- lock timeouts waste CPU time when transactions are aborted prematurely,
- implementation of intelligent lock manager non trivial,
- the manager itself is a single point of failure,
- "Groundhog Day": fixing deadlocks can trigger recurrent cascading aborts .

Cascading Aborts

	Transaction t	Transaction s	Shared
$Time$	Operation	Operation	a
1	$lock(a)$, $x \leftarrow a.getBalance()$	–	25
2	$x \leftarrow x + 100$	–	
3	$a.setBalance(x)$	–	125
4	$lock(b)$, $unlock(a)$	–	
5	not committed yet	$lock(a)$, $x \leftarrow a.getBalance()$	
6	–	$x \leftarrow 2\,x$	
7	–	$a.setBalance(x)$	250

Transaction s is reading dirty data at $t = 5$:

$\rightarrow$ If t aborts, then s will have to be aborted too.

Why cascading aborts happen?

Because transactions are allowed to read dirty data.

Avoiding Schedules that Enable Cascading Aborts

Locking not useful to avoid dirty reads

- The pursuit of serial equivalence *and* throughput maximization can make cascading aborts unavoidable.

Avoid Cascading Abort (ACA) Schedules

We will say that a schedule is an ACA schedule if:

- Every read operation by a transaction *must access* values written by already committed transactions.

Note on Terminology:

→ Aborts are also referred to as rollbacks, and the above is then an Avoid Cascading Rollback (ACR) schedule.

→ Coulouris refers to these schedules as strict executions.

Example #1: Non ACA Schedule

	Transaction t	Transaction s	Shared
$Time$	Operation	Operation	a
k	$a.deposit(10)$	–	
$k+1$	$b.deposit(10)$	–	
...	–	–	
l	–	$a.deposit(24)$	
$l+1$	–	$x \leftarrow b.getBalance()$	
$l+2$	–	$c.deposit(0.1x)$	
$l+3$	–	commit	
$l+4$	commit/abort	–	

If transaction t aborts, then s has to abort as well.

Question!

Why does s need to abort?

(A): Because local variable x

(B): Because of writes on a and c

Example #1: Non ACA Schedule

	Transaction t	Transaction s	Shared
$Time$	Operation	Operation	a
k	$a.deposit(10)$	–	
$k+1$	$b.deposit(10)$	–	
...	–	–	
l	–	$a.deposit(24)$	
$l+1$	–	$x \leftarrow b.getBalance()$	
$l+2$	–	$c.deposit(0.1x)$	
$l+3$	–	commit	
$l+4$	commit/abort	–	

If transaction t aborts, then s has to abort as well.

Question!

Why does s need to abort?

(A): Because local variable x

(B): Because of writes on a and c

→ (A): x can be used later on by the process hosting s

→ (B): Without the Coordinator intervening, s writing on a will be a *premature write*.

Example #2: ACA Schedule

	Transaction t	Transaction s	Shared
$Time$	Operation	Operation	a
k	$a.deposit(10)$	–	
$k+1$	$b.deposit(10)$	–	
...	–	–	
l	commit	$a.deposit(24)$	
$l+1$	–	$x \leftarrow b.getBalance()$	
$l+2$	–	commit	

Transaction s reads at $time = l + 1$ data guaranteed by t commit at $time = l$.

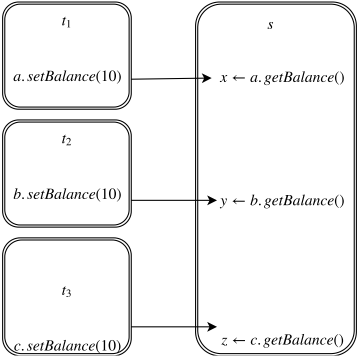
$\rightarrow$ At $time = l + 1$ we know for sure that changes to b will persist

Recoverable Schedules and Avoiding Cascading Aborts

Recoverable Schedule

A schedule is recoverable if each transaction from which it has *read* data has committed. Every ACA schedule is recoverable too.

Dependency graph



Rule for Recoverability

s can only *commit* if and only if t_1, t_2 and t_3 have committed already.

Guarantee

Any design that allows to undo, redo or both will restore shared objects to consistent state.

Question: Avoiding Cascading Aborts versus Serializability

Transactions t and s, with ACA schedule below

	Transaction t	Transaction s
$Time$	Operation	Operation
k	–	$a.setBalance(42)$
$k+1$	$b.setBalance(20)$	–
$k+2$	$a.setBalance(66)$	–
$k+3$	–	$x \leftarrow b.getBalance()$
$k+4$	–	$c.deposit(0.1x)$
$k+4$	$commit$	–
$k+6$	–	$commit$

HINT: a_k, b_k and c_k shared object at time$=k$

Question!

Is the schedule above equivalent to schedules $t \prec s$ or $s \prec t$?

(A): Yes

(B): No

Question: Avoiding Cascading Aborts versus Serializability

Transactions t and s, with ACA schedule below

	Transaction t	Transaction s
$Time$	Operation	Operation
k	–	$a.setBalance(42)$
$k+1$	$b.setBalance(20)$	–
$k+2$	$a.setBalance(66)$	–
$k+3$	–	$x \leftarrow b.getBalance()$
$k+4$	–	$c.deposit(0.1x)$
$k+4$	$commit$	–
$k+6$	–	$commit$

HINT: a_k, b_k and c_k shared object at time$=k$

Question!

Is the schedule above equivalent to schedules $t \prec s$ or $s \prec t$?

(A): Yes (B): No

$\rightarrow$ (B): For schedule above final state is $a = 66, b = 20, c = c_k + 2$. For $t \prec s$, $a = 42$, $b = 20$, $c = c_k + 2$. For $s \prec t$, $a = 66, b = 20, c = c_k + 0.1b_k$

Serial Equivalence and Recoverability

Important fact

The concepts of serial equivalence and recoverability are orthogonal.

A schedule can be:

Serial Equivalence and Recoverability

Important fact

The concepts of serial equivalence and recoverability are orthogonal.

A schedule can be:

1. Not serializable and not recoverable
2. Not serializable and recoverable
3. Serializable and non recoverable
4. Serializable and recoverable

Goal

Guarantee producing serializable and recoverable (ACA) schedules.

Agenda

Recall the Coordinator Interface

Coordinator interface

1. start(), returns unique identifier id
   - Client proc uses id with each request in the transaction
2. close(id), finalize transaction id, returns status (*commit* or *abort*)
   - commit indicates transaction successful, coordinator guarantees shared objects saved,
   - abort flags that it has not been completed.
3. abort(id), cancel transaction id
   - Coordinator guarantees *temporary effects* invisible to other transactions

An Optimistic Scheme

Idea

No Waiting → No Deadlocks

- Assumes that the likelihood of two transactions conflicting is low
- Transactions proceed without restriction until close() is invoked
- If transaction commits check if operations conflict with those of ongoing transactions
- In that case, abort and restart *some ongoing* transaction

Tentative Versions, Read and Write Sets

Transactions required to keep local copies of any shared object

- At any given time multiple versions of same object may exist,
- allows transactions to *abort* and avoid side effects

Tentative Versions, Read and Write Sets

Transactions required to keep local copies of any shared object

- At any given time multiple versions of same object may exist,
- allows transactions to *abort* and avoid side effects

Important

Tentative versions must be copies of the most recently committed version of the object.

Local copies held by transaction t arranged in two sets

- Read set $I(t)$ – set of shared objects data is read from
- Write set $O(t)$ – set of shared objects data is written to
- Same shared object can apper in both sets

Overview

Optimistic Currency Control (OCC) organised into three phases

1. Working Phase
   - Read operations are performed immediately on $I(t)$.
   - Write operations performed over $O(t)$,
   - these are considered tentative values, invisible to other transactions.
2. Validation Phase

Overview

Optimistic Currency Control (OCC) organised into three phases

1. Working Phase
   - Read operations are performed immediately on $I(t)$.
   - Write operations performed over $O(t)$,
   - these are considered tentative values, invisible to other transactions.
2. Validation Phase
   - When transaction closed, *Coordinator* validates it,
   - if successful, transaction commits,
   - otherwise, *either* or *both* transactions involved in conflict aborted.
3. Update Phase

Overview

Optimistic Currency Control (OCC) organised into three phases

1. Working Phase
   - Read operations are performed immediately on $I(t)$.
   - Write operations performed over $O(t)$,
   - these are considered tentative values, invisible to other transactions.
2. Validation Phase
   - When transaction closed, *Coordinator* validates it,
   - if successful, transaction commits,
   - otherwise, *either* or *both* transactions involved in conflict aborted.
3. Update Phase
   - If validated, then tentative values in $O(t)$ are made permanent
   - Transactions with read-only operations can commit immediately.

Guarantees of Optimistic Concurrency Control

Premature writes, dirty reads and inconsistent retrievals are precluded.

1. writes aren't rolled back by aborted transactions,
2. reads guaranteed to return a consistent set of committed values.

Guarantees of Optimistic Concurrency Control

Premature writes, dirty reads and inconsistent retrievals are precluded.

1. writes aren't rolled back by aborted transactions,
2. reads guaranteed to return a consistent set of committed values.

Lost updates can't occur due to validation

- Transactions assigned sequence number *when* entering validation
- If validated then sequence number is retained as version number for changes,
- else if rejected then number is reused and reassigned.

Guarantees of Optimistic Concurrency Control

Premature writes, dirty reads and inconsistent retrievals are precluded.

1. writes aren't rolled back by aborted transactions,
2. reads guaranteed to return a consistent set of committed values.

Lost updates can't occur due to validation

- Transactions assigned sequence number *when* entering validation
- If validated then sequence number is retained as version number for changes,
- else if rejected then number is reused and reassigned.

Sequence Numbers are Logical Clocks

Transactions sequence numbers are integers assigned in ascending order $\rightarrow$ define position in time

Optimistic Example

Transaction t_1 and t_2 want to deposit \$10 and \$20 on *Account* a, $a^I \in I(t_i)$, $a^O \in O(t_i)$ local copies

→ Latest commit to a by transaction with sequence number k

	Transaction t_1			Transaction t_2		
T	Operation	a^I	a^O	Operation	a^I	a^O
0	$a^I \leftarrow a.getBalance()$	50	–	$a^I \leftarrow a.getBalance()$	50	–
1	$a^O \leftarrow a^I + 10$	50	60	$a^O \leftarrow a^I + 20$	50	70

1. t_1 and t_2 proceed to validation
2. t_1 gets assigned sequence number $k+1$, t_2 gets $k+2$
3. t_1 is validated successfully, and commits 60 to a
4. t_2 fails because t_1 wrote a value to a, invalidating the one read by t_2

    → Sequence numbers used to track if most recent committed value used

Validation of Transactions

Sequence numbers impose *total order* amongst transactions

→ A transaction with number t_i always precedes t_j with $j > i$

Validation of Transactions

Sequence numbers impose *total order* amongst transactions

→ A transaction with number t_i always precedes t_j with $j > i$

Let t_v be transaction being validated, t_i ongoing transaction,

→ test pairs of transactions (t_i, t_v) for compliance with rules below

t_v	t_i	Rule
write	read	$I(t_i) \cap O(t_v) = \emptyset$, t_i read and t_v write set disjoint
read	write	$I(t_v) \cap O(t_i) = \emptyset$, t_v read and t_i write set disjoint
write	write	$O(t_v) \cap O(t_i) = \emptyset$, no concurrent writes

Reminder

$X \cap Y$ denotes the intersection of sets X and Y i.e. the subset of common elements in X and Y.

Algorithms and Data Structures

Algorithm 1 Check if $X \cap Y$ is the empty set $\emptyset$

1: **for** $x \in X$ **do**
2:     **if** $x \in Y$ **then**
3:         **return** $\perp$
4:     **end if**
5: **end for**
6: **return** $\top$

Question!

What data structure we want to use to implement algorithm above?

- (A): A list
- (B): A hashtable
- (C): Depends on input sizes
- (D): Depends on access and data statistics

Algorithms and Data Structures

Algorithm 2 Check if $X \cap Y$ is the empty set $\emptyset$

for $x \in X$ **do**
  if $x \in Y$ **then**
    return $\perp$
  end if
end for
return $\top$

Question!

What data structure we want to use to implement algorithm above?

(A): A list

(B): A hashtable

(C): Depends on input sizes

(D): Depends on access and data statistics

$\rightarrow$ (C & D): hashtable has overheads (setup, hashing) you need to be sure you amortize those computations.

Optimizations on Validation

Performance Alert (N=number of operations in longest transaction)

Naïve algorithm to check compliance with rules is $O(N^3)$

t_v	t_i	Rule
write	read	$I(t_i) \cap O(t_v) = \emptyset$, t_i read and t_v write set disjoint
read	write	$I(t_v) \cap O(t_i) = \emptyset$, t_v read and t_i write set disjoint
write	write	$O(t_v) \cap O(t_i) = \emptyset$, no concurrent writes

Optimization

Optimizations on Validation

Performance Alert (N=number of operations in longest transaction)

Naïve algorithm to check compliance with rules is $O(N^3)$

t_v	t_i	Rule
write	read	$I(t_i) \cap O(t_v) = \emptyset$, t_i read and t_v write set disjoint
read	write	$I(t_v) \cap O(t_i) = \emptyset$, t_v read and t_i write set disjoint
write	write	$O(t_v) \cap O(t_i) = \emptyset$, no concurrent writes

Optimization

Allow only one transaction to be in the validation and update phase.

- No concurrent writes come for "free",
- and dirty reads precluded, all schedules are ACA
- But we need to implement Validation & Update phases within a critical section

Transaction Sequence Numbers

Facts about Sequence Numbers

- Assigned when transaction enters validation phase
- Act as pseudo-clock that ticks when transactions completes successfully

Question!

What would happen if sequence numbers were assigned when transactions entered the working phase?

(A): Throughput would be reduced

(B): Fast, short transactions would be penalized

(C): We would avoid Cascading Aborts entirely

(D): Nothing weird

Transaction Sequence Numbers

Facts about Sequence Numbers

- Assigned when transaction enters validation phase
- Act as pseudo-clock that ticks when transactions completes successfully

Question!

What would happen if sequence numbers were assigned when transactions entered the working phase?

(A): Throughput would be reduced

(B): Fast, short transactions would be penalized

(C): We would avoid Cascading Aborts entirely

(D): Nothing weird

$\rightarrow$ (A & B): (Fast) transactions t_i reaching end of working phase, would have to wait for slower t_j with lower sequence numbers ($j < i$).

Backward and Forward Validation

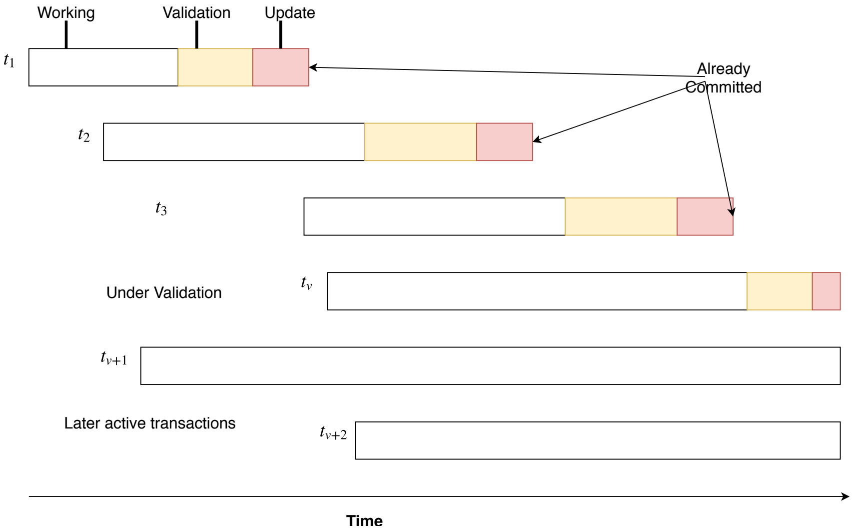
t_v	t_i	Rule
write	read	$I(t_i) \cap O(t_v) = \emptyset$, t_i read and t_v write set disjoint
read	write	$I(t_v) \cap O(t_i) = \emptyset$, t_v read and t_i write set disjoint
write	write	$O(t_v) \cap O(t_i) = \emptyset$, no concurrent writes

Still read/write conflicts need to be checked between pairs (t_v, t_i)

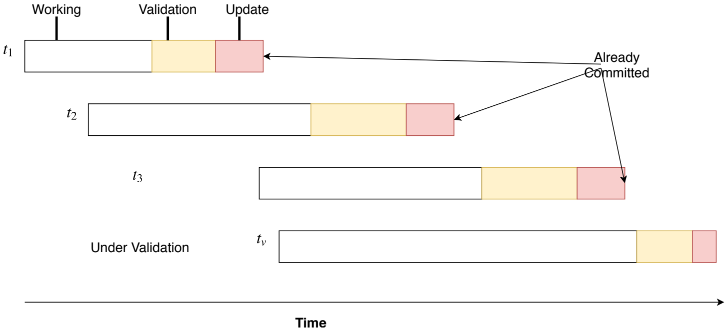
Two strategies

- Backward – t_i precedes t_v, and already validated & committed.
- Forward – t_i is later transaction, still in their working phase.

Illustration: Concurrent Transactions

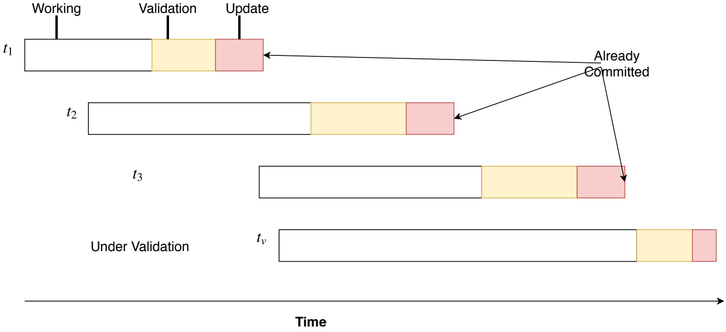


Backwards Validation - I



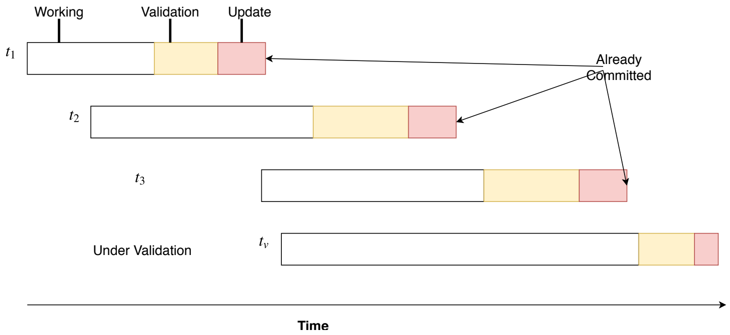
- t_v writes cannot interfere with read operations by t_1, t_2, t_3 i.e. $I(t_i)$ and $O(t_v)$ may overlap, but no longer relevant.
- Validation of t_v checks if $I(t_v) \cap O(t_i)$ empty, otherwise t_v validation will fail

Backwards Validation - II



- since t_i have already committed, only way to resolve conflict is to abort t_v,
- we need to check for emptiness $O(t_2) \cap I(t_v)$ and $O(t_3) \cap I(t_v)$,
- transactions t_i with $O(t_i) = \emptyset$ do not need to be checked.

Backwards Validation - III



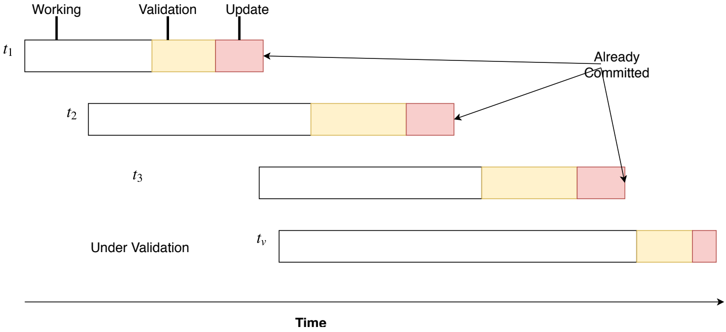
Question!

Do we need to check for emptiness the intersection between $O(t_1)$, write set for t_1, and $I(t_v)$ the read set of t_v?

(A): Yes

(B): No

Backwards Validation - III



Question!

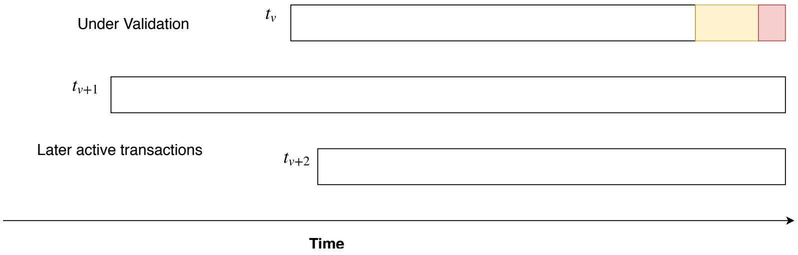
Do we need to check for emptiness the intersection between $O(t_1)$, write set for t_1, and $I(t_v)$ the read set of t_v?

(A): Yes

(B): No

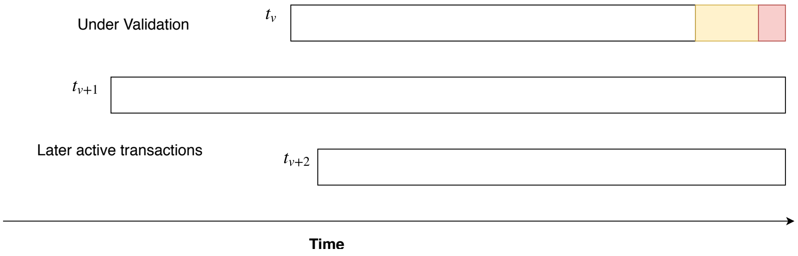
→ (B): $O(t_1) \cap I(t_v)$ does not matter since t_1 already committed when t_v started

Forward Validation - I



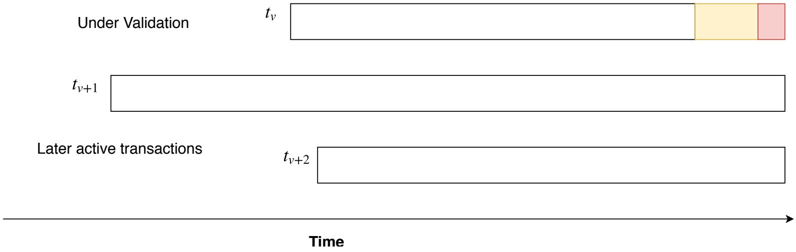
- We need to check emptiness for $O(t_v) \cap I(t_{v+i})$, as these are in their working phase,
- and there is no need to check $O(t_{v+i}) \cap I(t_v)$, since transactions t_{v+i} will not write until t_v has completed.

Forward Validation - II



- Read-only transactions t_{v+i} always pass the test,
- if $O(t_v) \cap I(t_{v+i}) \neq \emptyset$ then several strategies are possible, to handle conflict

Forward Validation - Conflict Resolution



- abort t_{v+i} and commit t_v,
- defer validation until t_{v+i} finished,
- abort t_v, simplest but t_{v+i} may abort anyways

Forward versus Backward Validation

Regarding conflict resolution

- Forward validation more flexible than backwards validation,
- backward validation only allows to abort t_v.

Backward validation requires to store $O(t)$ until all overlapping transactions have finished

- That can be a long time

In general sets $I(t)$ bigger than $O(t)$

→ This has important implications on optimal data structures to implement $I(t_i) \cap O(t_j) = \emptyset$,

→ i.e. amortize runtime to construct hash table etc.

Validation and Starvation

Restarting Transactions does not Guarantee Starvation-freeness

There is no guarantee that a transaction t_i will ever pass validation checks

- t_i may be aborted and then be in conflict with a new ongoing, faster transaction t_j.

Extra responsibilities for *Coordinator*

- ensure that transactions from process p do not get aborted too often,
- if so, make p to be privileged using locking.

So we get deadlocks through a back door?

- Less likely than starvation because locks make transactions wait
- But, distributed deadlock detection is very hard to implement!

Agenda

Pessimism versus Optimism

Pessimistic approach (detect conflicts as they arise)

- Locking decides serialization order *dynamically*
- Locking is good for transactions where writes $>>$ reads
- Can get deadlock

Optimistic methods

- All transactions proceed, but may need to abort at the end
- Efficient when there are few write conflicts
- aborts lead to repeated work (Question: does this always matter?)

Summary of Transactions so Far

The analysis of conflicts amongs transactions operations forms basis to derive concurrency control protocols

- Protocols ensure serial equivalence of schedules
- allow for recovery by using ACA schedules

Three alternative strategies are possible in scheduling an operation in a transaction

- (1) to execute it immediately, (2) to delay it, or (3) to abort it
- Strict two-phase locking (2PL) uses (1) and (2), aborting in the case of deadlock
  - Ordering depends on timing of accesses to shared objects
- Optimistic concurrency control allows transactions to proceed without any form of checking until they are completed
  - Validation is carried out; starvation can occur

Agenda

Further Reading

Weikum & Vossen *Transactional Information Systems*, Morgan Kauffman, 2002

- Formalizes the notion of recoverability, available at the Library

Coulouris et al. *Distributed Systems: Concepts & Design*

- See Chapter 16.2 for strict executions/ACA schedules.
- See Chapter 16.5 for Optimistic Concurrency Control.
- Chapter 16.7 compares Locking with Optimistic Concurrency Control.

Terry et al. *No compromises: distributed transactions with consistency, availability, and performance* In ACM Symposium in Operating Systems Principles, 2015

- State–of–the–art Optimistic Concurrency Control (FaRM)

Louridas *Real-World Algorithms: A Beginner's Guide*, MIT Press, 2017

- See Chapter 11.3 "The Matthew Effect and Power Laws"