On Optimistic Methods for Concurrency Control

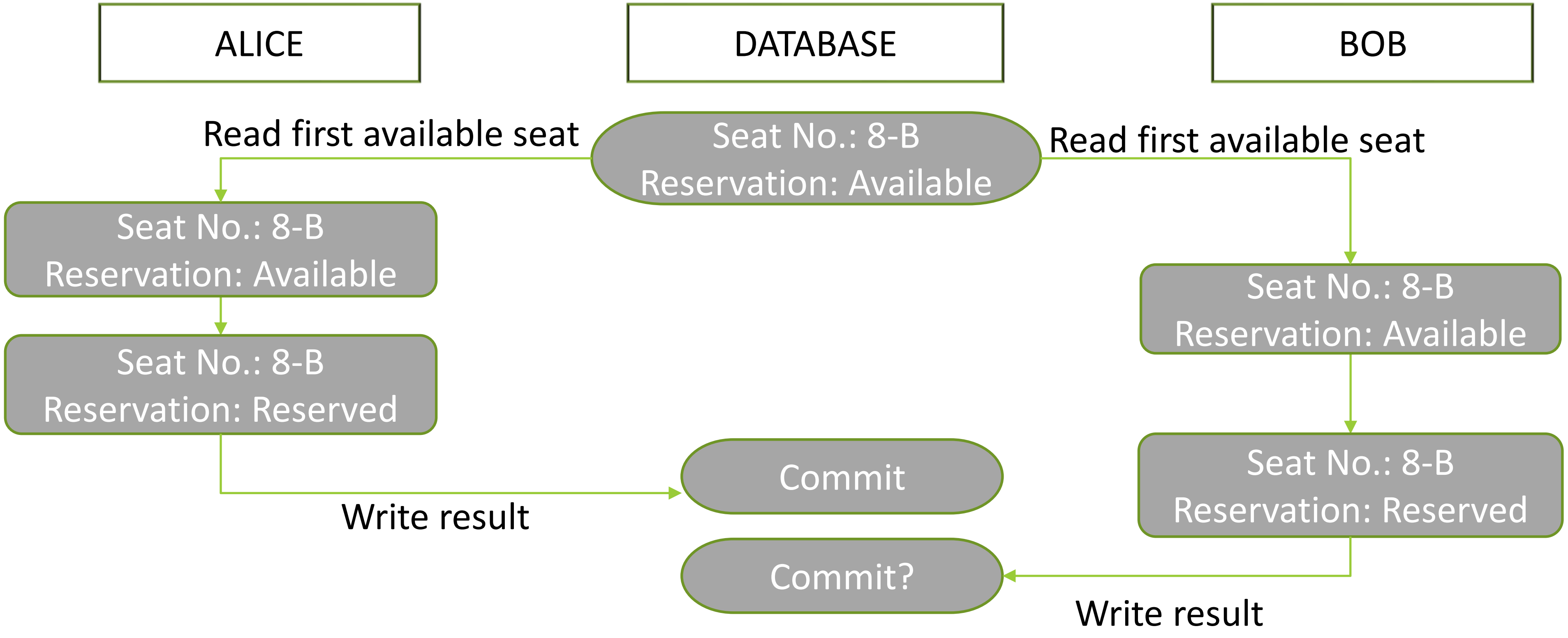
AUTHORS: H.T. KUNG and JOHN T. ROBINSON

PRESENTATION BY:
DHIVYA SIVARAMAKRISHNAN

Introduction

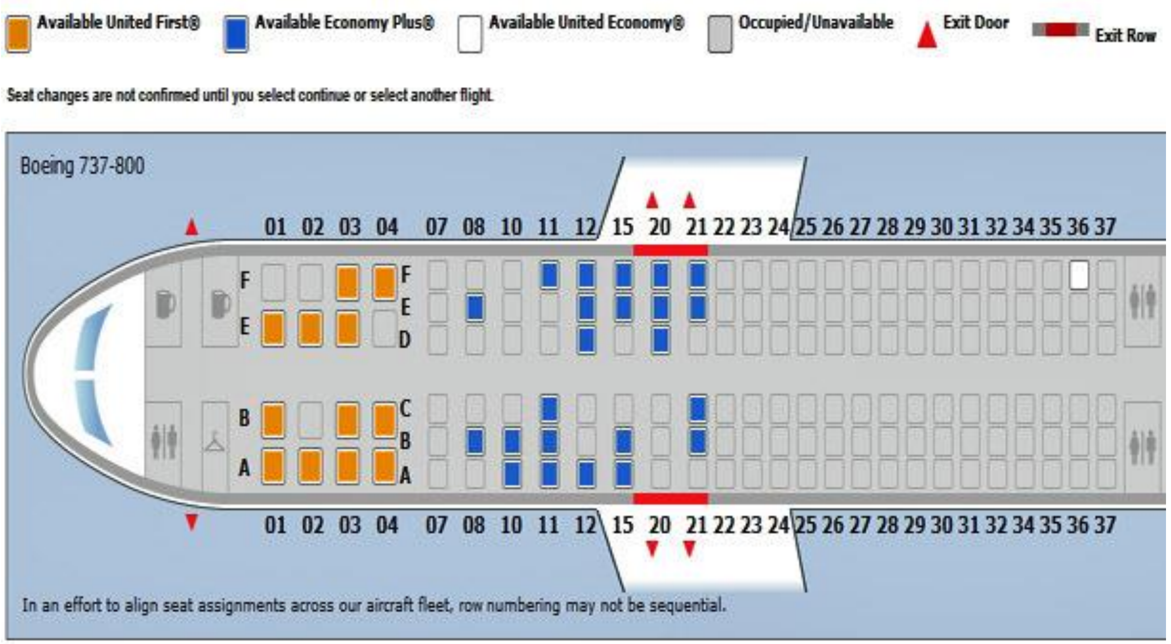
- What is a Transaction?
- What is concurrent access?
- Why is it so desirable?
- Preserving database integrity while allowing concurrency

Lost Update


ALICE
DATABASE
BOB
Read first available seat
Seat No.: 8-B
Reservation: Available
Read first available seat
Seat No.: 8-B
Reservation: Available
Seat No.: 8-B
Reservation: Reserved
Seat No.: 8-B
Reservation: Available
Seat No.: 8-B
Reservation: Reserved
Commit
Write result
Commit?
Write result

Locking Insights

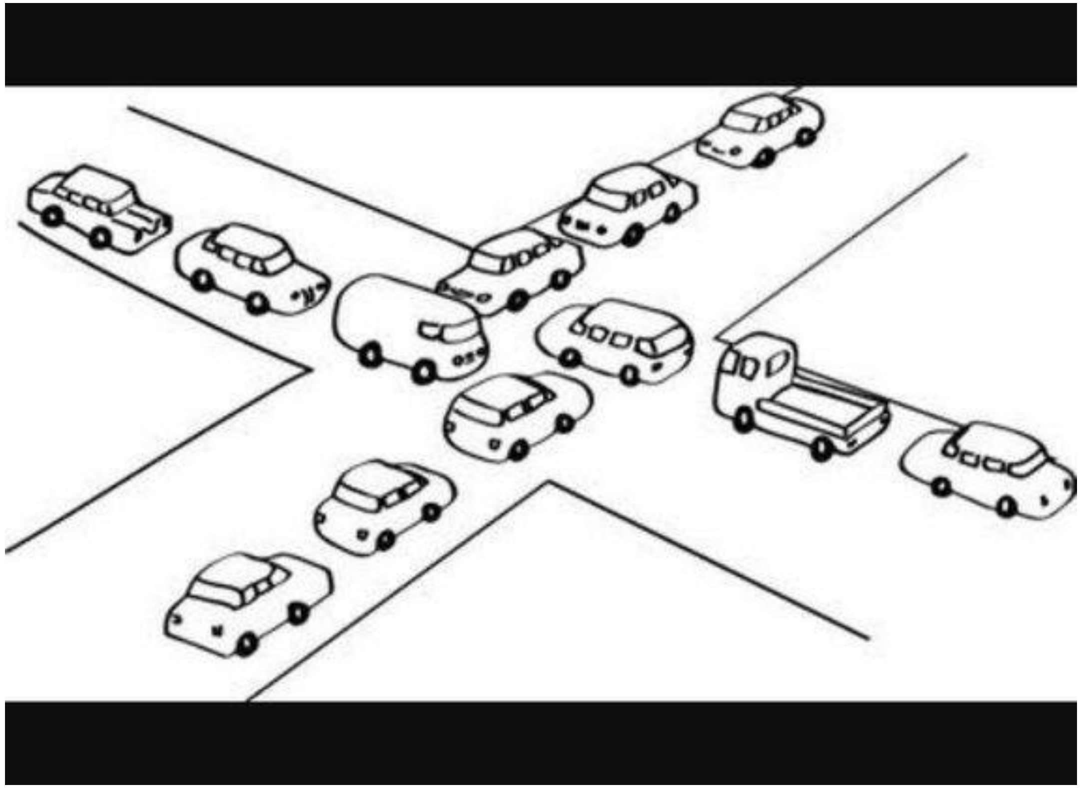
- Alice and Bob access the database concurrently
- Locking - exclusive access to the resource
- Other attempts will be invalidated
- No process will act upon obsolete or work-in-progress information



Trade-offs with Locking

- Lock maintenance overheads
- Impacts on concurrency, especially for aborted transactions
- Ensuring availability of congested nodes
- Secondary memory swaps on locked resources
- Deadlocks can occur

Deadlocking Explained



Focus of this Paper

- **Locking**
  - Disadvantages are identified
- **Transaction Backup**
  - An “optimistic” methodology
  - Avoids the pitfalls of locking

Foundation for “Optimism”

Locking may be necessary only in the worst case

General cases:

- Very high number of total resources compared to those being accessed
- Probability of modifying a congested resource is less
- Access conflicts will not happen among transactions

Three Phases

Read
- Unrestricted - not a “modify“ and cannot affect database integrity

Validation
- Determines if transaction causes any loss of integrity

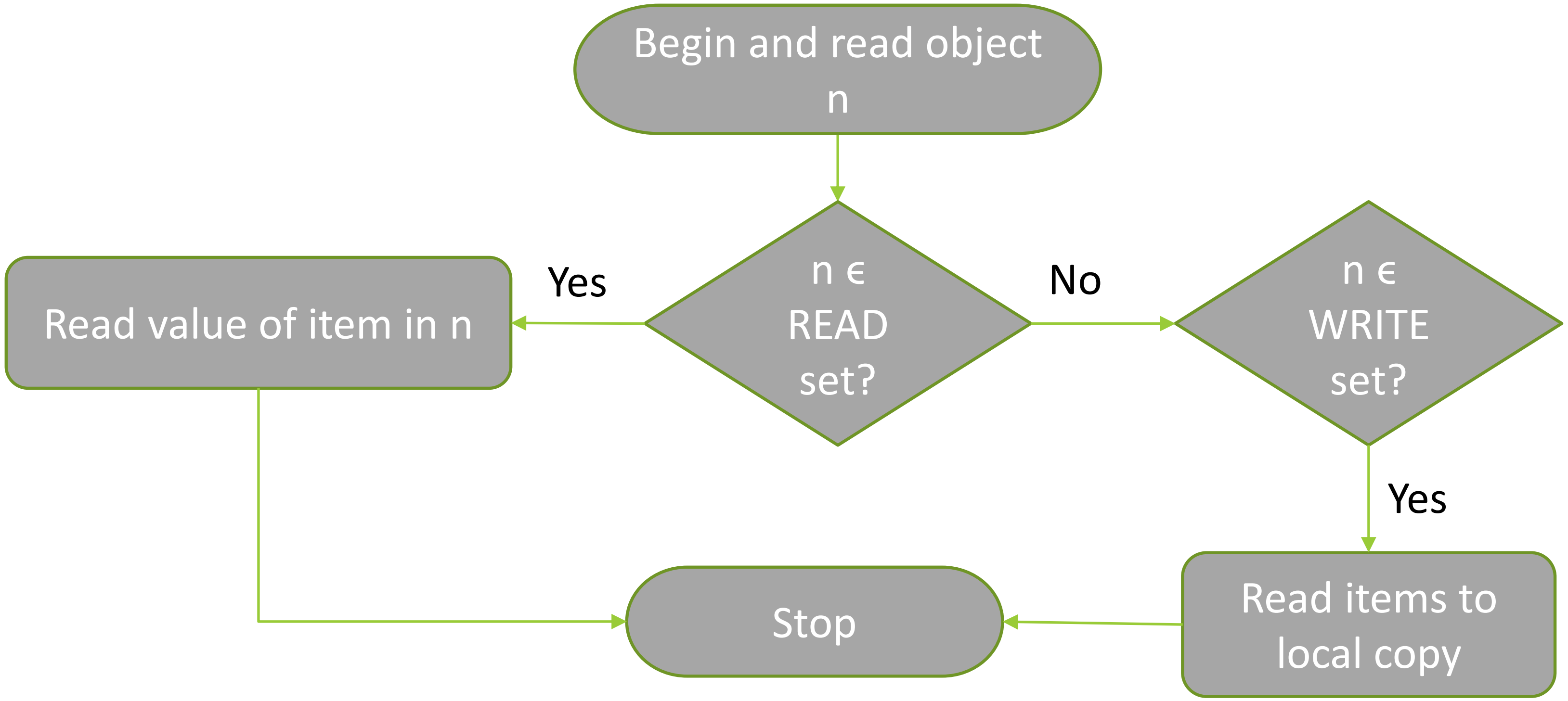
Write
- Stringent restrictions. Writes happen only if validation succeeds

What to Know

- A set of homogeneous objects of type A
- Concurrency control mechanism maintains OBJECT NAMES used by every transaction
- Assumed to be an empty set at the very beginning
- Every transaction has two copies of objects used – “read” set, “write” set

CREATE	Create a new object and return its name
READ	Read an item of an object
WRITE	Write a value to an item of an object
DELETE	Delete an object
COPY, EXCHANGE	Create copies of an object, swap values

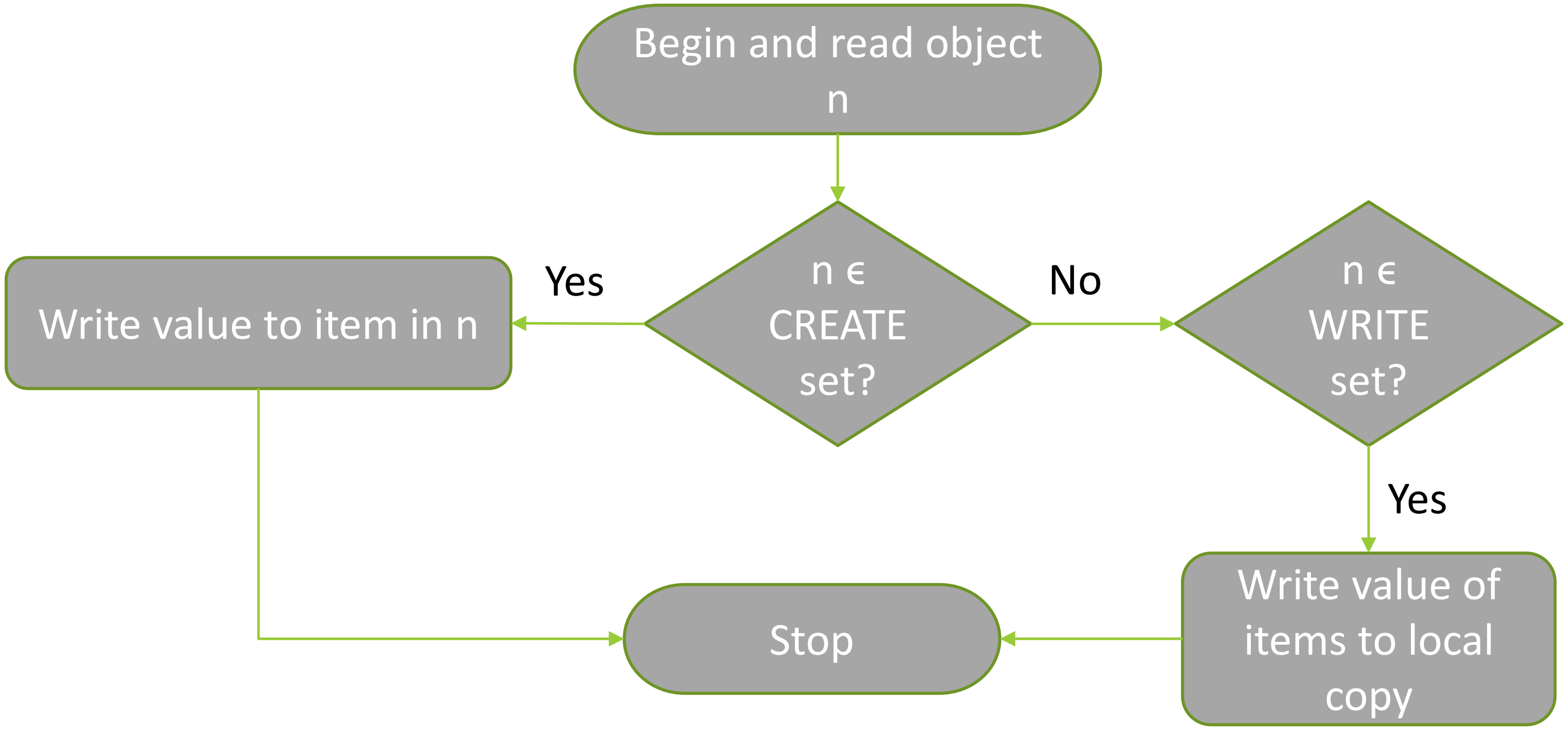
READ WRITE VALIDATE



READ

WRITE

VALIDATE

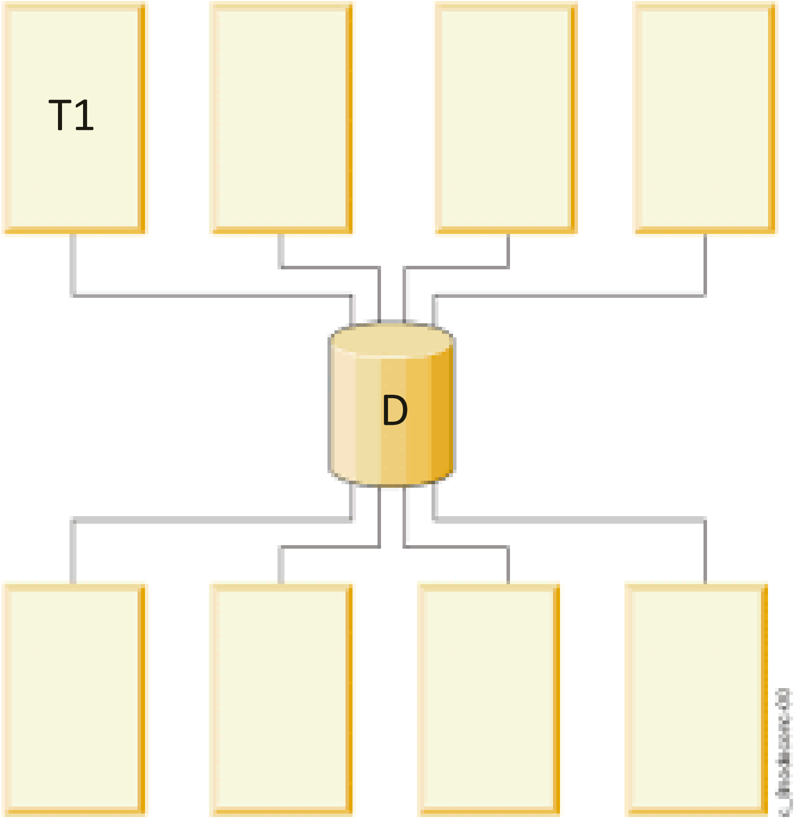


Integrity Preservation

- No root node can be created without writing new pointers to access it
- Root node deletions must clean up dangling pointers
- At transaction completion –
  - Created nodes become accessible
  - Deleted nodes become inaccessible
- Cleanup also happens after a transaction is aborted
- At the end of READ, all changes to ‘n’ are known

READ WRITE VALIDATE

- Every transaction aims to preserve integrity of this shared data structure, D
- Check if D has been updated by any other transaction since the start of T1
- How do we verify the correctness of this concurrent execution?



Serial Equivalence

- 'n' transactions concurrently access a database resource
- Two instances of transaction interleaving

Complete Schedule	
T_1	T_2
R(A)	
W(A)	
Commit	
	R(B)
	W(B)
	Abort

Complete Schedule	
T_1	T_2
R(A)	
	R(B)
W(A)	
	W(B)
Commit	
	Abort

- Same effect on database as if all the transactions ran one after the other

Why is Serial Equivalence important?

- An easy way to validate that every transaction preserves integrity
- Easier to verify serial equivalence than check integrity after every interleaving of concurrent transactions
- Preserves the basic property for consistency – every transaction is atomic in nature
- Any amount of interleaving is possible, but the end result is the same – a consistent state

Validating Serial Equivalence

- PROBLEM: Prove that database state remains same after any interleaving
- Find a permutation such that serial equivalence holds
- Assign transaction numbers t(tname)

$$\boxed{t(i) < t(j)}$$

Transaction Numbers

Each transaction given a unique number	• Indicates its position in time
Number assigned through counters	• End of READ
Transactions that complete WRITE	• Number retained
Aborted transactions	• Number recycled

Three Validation Conditions

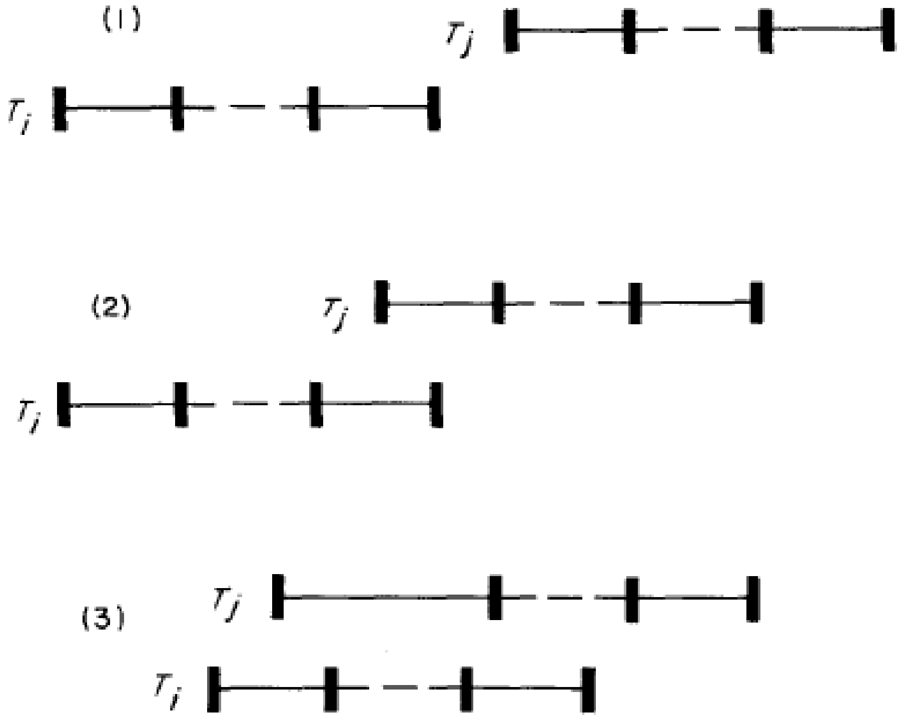


Fig. 2. Possible interleaving of two transactions.

Serial Validation

- First of a family of concurrency controls
- Utilizes validation conditions 1 and 2 - sequential WRITEs
- Record “read” and “write” sets to local copy
- Tid, validation and subsequent write are all in a critical section

```
tbegin = (
    create set := empty;
    read set := empty;
    write set := empty;
    delete set := empty;
    start tn := tnc)

tend = (
    (finish tn := tnc;
     valid := true;
     for t from start tn + 1 to finish tn do
         if (write set of transaction with transaction number t intersects read set)
            then valid := false;
     if valid
         then ((write phase); tnc := tnc + 1; tn := tnc));
     if valid
         then (cleanup)
         else (backup)).
```

Parallel Validation

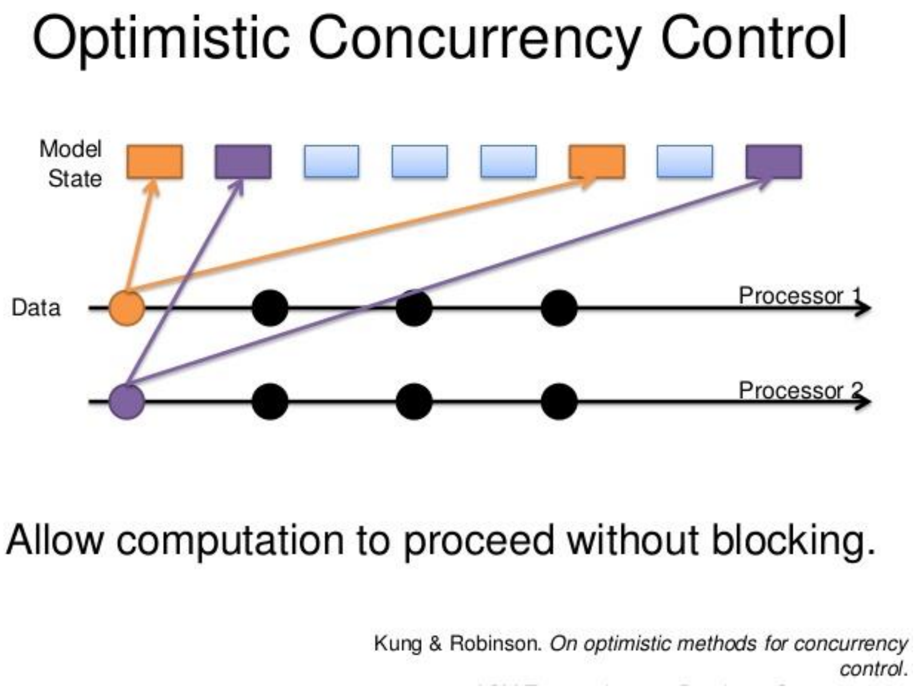
- Another family of concurrency control
- Uses all three validation conditions
- Multiple transactions may be in the validation phase at once
- Provides optimization similar to Serial Validation

Parallel Validation - Procedure

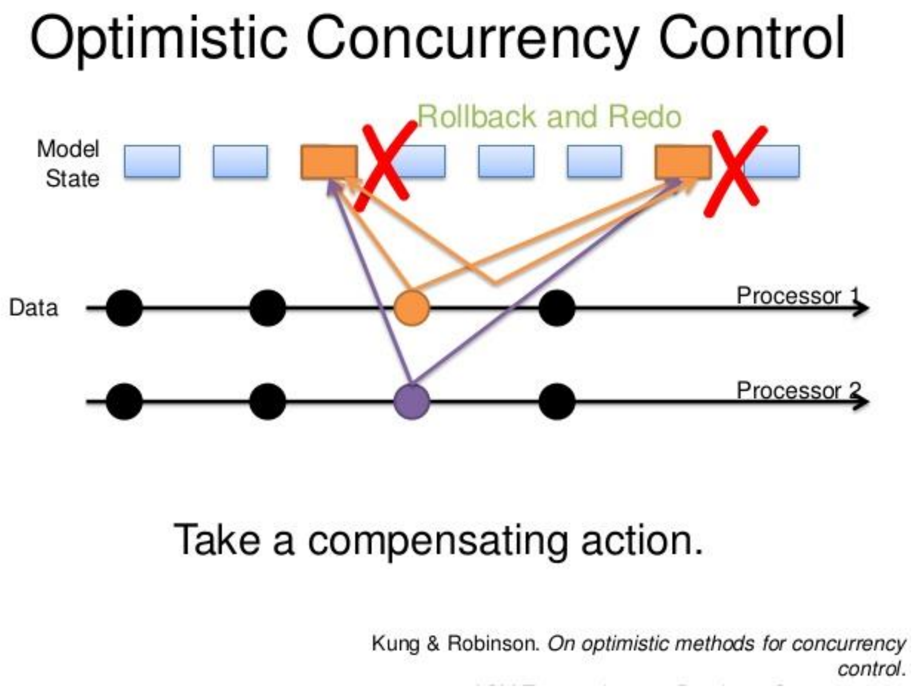
- Save active transactions – finished READ
- Validate against conditions 1 and 2
- Validate against 3 for all transactions in “active” set
- If no conflicts, remove self from active and assign T(id)
- Else, abort

A Comparison

NO CONFLICTS



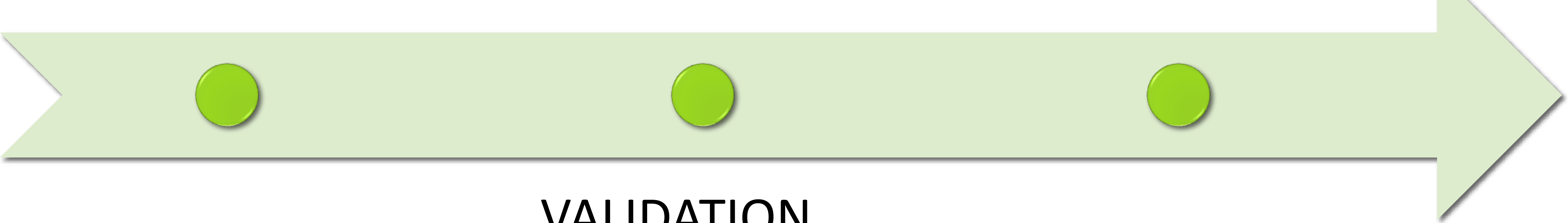
CONFLICT



A Quick Recap

READ

- Create local copies



VALIDATION

- Ensure database consistency

WRITE

- Make local copies global

Merits

- Locking overheads avoided - good access throughput
- Conflicts are assessed pretty early - at the end of READ
- Maximized parallelism
- Cost of rollbacks is lesser than deadlock resolution cost
- Negligible concurrency control overhead if more READs

Major Demerits

- Relies solely on the belief that the likelihood of two transactions conflicting is low
- Conflicting transactions need to be aborted and restarted
- Too much redundancy if many transactions are aborted
- With heavy concurrency, heavy load and failure probabilities
- Starvation when same transactions are aborted

Real-Time Users of OCC

MediaWiki
Because ideas want to be free.

RAILS

Microsoft .NET
Entity Framework

elastic

Google app engine

Conclusion

- Two branches of concurrency control
- Locking => resource-waiting
- Optimistic methods => all transactions to proceed and conflicting ones are aborted
- How to choose –
  - Locking – when chances of users updating same objects at once are high
  - Optimistic – if resources are many but transactions are fewer; more READs
- Unified goals – more throughput, less turnaround time

References (In Order of Slides)

4 - http://sharetraveler.com/get-good-seat-economy-class/

6 - http://csunplugged.org/routing-and-deadlock/

11 - https://slidehunter.com/powerpoint-templates/setting-smart-objectives-powerpoint-template/

15 - https://www.ibm.com/support/knowledgecenter/SSPHQG_7.2.0/com.ibm.powerha.concepts/ha_concepts_8node_takeover.htm

16 - https://gradeup.co/transactions-and-concurrency-control-i-4c5d9b27-c5a7-11e5-bcc4-bc86a005f7ba

20, 22 - H. T. Kung, John T. Robinson, ‘On Optimistic Methods for Concurrency Control’, June 1981

25 - Concurrency Control for Parallel Machine Learning - http://www.slideshare.net/jeykottalam/concurrency-control-for-machine-learning

29 - https://en.wikipedia.org

31 - Google Images for “Questions”