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

Read first available seat

Seat No.: 8-B
Reservation: Available

Seat No.: 8-B
Reservation: Reserved

Write result

DATABASE

Seat No.: 8-B
Reservation: Available

Commit

BOB

Read first available seat

Seat No.: 8-B
Reservation: Available

Seat No.: 8-B
Reservation: Reserved

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
<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE</td>
<td>Create a new object and return its name</td>
</tr>
<tr>
<td>READ</td>
<td>Read an item of an object</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write a value to an item of an object</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete an object</td>
</tr>
<tr>
<td>COPY, EXCHANGE</td>
<td>Create copies of an object, swap values</td>
</tr>
</tbody>
</table>
**READ**

Begin and read object

n

Yes

Read value of item in n

n ∈ READ set?

Yes

Stop

No

WRITE

n ∈ WRITE set?

Yes

Read items to local copy

VALIDATE

n ∈ set?
Begin and read object n

Write value to item in n

n ∈ CREATE set?

Yes

Write value of items to local copy

n ∈ WRITE set?

No

Stop

Yes

No

CMPT 843 - SPRING 2017
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
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

- 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 $t(t\text{name})$

$t(i) < t(j)$
Transaction Numbers

- Each transaction given a unique number
- Number assigned through counters
- Transactions that complete WRITE
- Aborted transactions

• Indicates its position in time
• End of READ
• Number retained
• Number recycled
Three Validation Conditions

Fig. 2. Feasible 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
\texttt{thebegin} = ( \\
\quad \text{create set} := \text{empty}; \\
\quad \text{read set} := \text{empty}; \\
\quad \text{write set} := \text{empty}; \\
\quad \text{delete set} := \text{empty}; \\
\quad \text{start tn} := \text{tnc}) \\
\texttt{tend} = ( \\
\quad \langle \text{finish tn} := \text{tnc}; \ \\
\quad \text{valid} := \text{true}; \ \\
\quad \text{for } t \text{ from start } tn + 1 \text{ to finish } tn \text{ do} \ \\
\quad \quad \text{if (write set of transaction with transaction number } t \text{ intersects read set) } \\
\quad \quad \quad \text{then valid} := \text{false}; \ \\
\quad \text{if valid} \ \\
\quad \quad \text{then } ((\text{write phase}); \text{tnc} := \text{tnc} + 1; \text{tn} := \text{tnc}); \ \\
\quad \text{if valid} \ \\
\quad \quad \text{then (cleanup)} \ \\
\quad \text{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

Optimistic Concurrency Control

Allow computation to proceed without blocking.

Kung & Robinson. On optimistic methods for concurrency control.

CONFLICT

Optimistic Concurrency Control

Take a compensating action.

Kung & Robinson. On optimistic methods for concurrency control.
A Quick Recap

**READ**
- Create local copies

**WRITE**
- Make local copies global

**VALIDATION**
- Ensure database consistency
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
- RAILS
- .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/
16 - https://gradeup.co/transactions-and-concurrency-control-i-4c5d9b27-c5a7-11e5-bcc4-bc86a005f7ba
29 - https://en.wikipedia.org
31 - Google Images for “Questions”