Today’s Papers

- **The Notions of Consistency and Predicate Locks in a Database System**
  

- **Key Range Locking Strategies for Improved Concurrency**
  

Thoughts?

Overview

- Serializability
- The Phantom Issue
- Predicate Locking
- Key-Range Locks
- Next-Key Locking techniques
- Index Management and Transactions
- Multi-level reasoning

Theory and reality

- Traditional serializability theory treats database as a set of items (Eswaran et al. ’76 says “entities”) which are read and written
  
  Two phase locking is proved correct in this model
  
  - We now say “serializable”

- But, database has a richer set of operations than just read/write
  
  - Declarative selects
  
  - Insert
  
  - Delete
**Review: Goals of Transaction Scheduling**

- Maximize system utilization, i.e., concurrency
  - Interleave operations from different transactions

- Preserve transaction semantics
  - Semantically equivalent to a serial schedule, i.e., one transaction runs at a time


  Serial schedule (T1, then T2):

  Serial schedule (T2, then T1):

**Two Key Questions**

1) Is a given schedule equivalent to a serial execution of transactions?


\[ \equiv ? \]

\[ \text{Serial schedule (T1, then T2):} \]

\[ \equiv ? \]

\[ \text{Serial schedule (T2, then T1):} \]

2) How do you come up with a schedule equivalent to a serial schedule?

**Transaction Scheduling**

- **Serial schedule**: A schedule that does not interleave the operations of different transactions
  - Transactions run serially (one at a time)

- **Equivalent schedules**: For any storage/database state, the effect (on storage/database) and output of executing the first schedule is identical to the effect of executing the second schedule

- **Serializable schedule**: A schedule that is equivalent to some serial execution of the transactions
  - Intuitively: with a serializable schedule you only see things that could happen in situations where you were running transactions one-at-a-time

**Anomalies with Interleaved Execution**

- May violate transaction semantics, e.g., some data read by the transaction changes before committing

- Inconsistent database state, e.g., some updates are lost

- Anomalies always involves a “write”; Why?
Anomalies with Interleaved Execution

• Read-Write conflict (Unrepeatable reads)

\[
\begin{align*}
T1: & \ R(A), \quad R(A), W(A) \\
T2: & \quad R(A), W(A)
\end{align*}
\]

• Violates transaction semantics

• Example: Mary and John want to buy a TV set on Amazon but there is only one left in stock
  – (T1) John logs first, but waits…
  – (T2) Mary logs second and buys the TV set right away
  – (T1) John decides to buy, but it is too late…


Anomalies with Interleaved Execution

• Write-read conflict (reading uncommitted data)

\[
\begin{align*}
T1: & \ R(A), W(A), \quad W(A) \\
T2: & \quad R(A), \quad ...
\end{align*}
\]

• Example:
  – (T1) A user updates value of A in two steps
  – (T2) Another user reads the intermediate value of A, which can be inconsistent
  – Violates transaction semantics since T2 is not supposed to see intermediate state of T1

Anomalies with Interleaved Execution

• Write-write conflict (overwriting uncommitted data)

\[
\begin{align*}
T1: & \ W(A), \quad W(B) \\
T2: & \quad W(A), W(B)
\end{align*}
\]

• Get T1’s update of B and T2’s update of A

• Violates transaction serializability

• If transactions were serial, you’d get either:
  – T1’s updates of A and B
  – T2’s updates of A and B

Conflict Serializable Schedules

• Two operations conflict if they
  – Belong to different transactions
  – Are on the same data
  – At least one of them is a write

• Two schedules are conflict equivalent iff:
  – Involve same operations of same transactions
  – Every pair of conflicting operations is ordered the same way

• Schedule S is conflict serializable if S is conflict equivalent to some serial schedule
Conflict Equivalence – Intuition

- If you can transform an interleaved schedule by swapping consecutive non-conflicting operations of different transactions into a serial schedule, then the original schedule is conflict serializable

Example:

\[
\begin{align*}
T1: & \text{R(A), W(A), R(B), W(B)} \\
T2: & \text{R(A), W(A), R(B), W(B)} \\
T1: & \text{R(A), W(A), R(B), W(B)} \\
T2: & \text{R(A), W(A), R(B), W(B)} \\
T1: & \text{R(A), W(A), R(B), W(B)} \\
T2: & \text{R(A), W(A), R(B), W(B)} \\
\end{align*}
\]

Conflict Equivalence – Intuition (cont’d)

- If you can transform an interleaved schedule by swapping consecutive non-conflicting operations of different transactions into a serial schedule, then the original schedule is conflict serializable

Example:

\[
\begin{align*}
T1: & \text{R(A), W(A), R(B), W(B)} \\
T2: & \text{R(A), W(A), R(B), W(B)} \\
T1: & \text{R(A), W(A), R(B), W(B)} \\
T2: & \text{R(A), W(A), R(B), W(B)} \\
T1: & \text{R(A), W(A), R(B), W(B)} \\
T2: & \text{R(A), W(A), R(B), W(B)} \\
\end{align*}
\]

Dependence Graph

- Dependency graph:
  - Transactions represented as nodes
  - Edge from Ti to Tj:
    - an operation of Ti conflicts with an operation of Tj
    - Ti appears earlier than Tj in the schedule

- Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic
Example

• Conflict serializable schedule:

- No cycle!

```
T1: R(A), W(A),          R(B), W(B)
T2:          R(A), W(A),         R(B), W(B)
```

Example

• Conflict that is *not* serializable:

- Cycle: The output of T1 depends on T2, and vice-versa

```
T1: R(A), W(A),                   R(B), W(B)
T2:          R(A), W(A),R(B), W(B)
```

Notes on Conflict Serializability

• Conflict Serializability doesn’t allow all schedules that you would consider correct
  – This is because it is strictly *syntactic* - it doesn’t consider the meanings of the operations or the data

• In practice, Conflict Serializability is what gets used, because it can be done efficiently
  – Note: in order to allow more concurrency, some special cases do get implemented, such as for travel reservations, ...

• Two-phase locking (2PL) is how we implement it

Serializability ≠ Conflict Serializability

• Following schedule is *not* conflict serializable

```
T1: R(A), W(A),
T2:     W(A),
T3:                WA
```

• However, the schedule is serializable since its output is equivalent with the following serial schedule

```
T1: R(A), W(A),
T2:     W(A),
T3:                WA
```

• Note: deciding whether a schedule is serializable (not conflict-serializable) is NP-complete
Locks (Simplistic View)

- Use *locks* to control access to data

- Two types of locks:
  - shared (S) lock – multiple concurrent transactions allowed to operate on data
  - exclusive (X) lock – only one transaction can operate on data at a time

<table>
<thead>
<tr>
<th>Lock Compatibility Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>X</td>
</tr>
</tbody>
</table>

Two-Phase Locking (2PL)

1) Each transaction must obtain:
   - S (shared) or X (exclusive) lock on data before reading,
   - X (exclusive) lock on data before writing

2) A transaction cannot request additional locks once it releases any locks

Thus, each transaction has a “growing phase” followed by a “shrinking phase”

Avoid deadlock by acquiring locks in some lexicographic order

Two-Phase Locking (2PL)

- 2PL guarantees conflict serializability
- Doesn’t allow dependency cycles. Why?
- Answer: a dependency cycle leads to deadlock
  - Assume there is a cycle between Ti and Tj
  - Edge from Ti to Tj: Ti acquires lock first and Tj needs to wait
  - Edge from Tj to Ti: Tj acquires lock first and Ti needs to wait
  - Thus, both Ti and Tj wait for each other
  - Since with 2PL neither Ti nor Tj release locks before acquiring all locks they need → deadlock

  - Schedule of conflicting transactions is conflict equivalent to a serial schedule ordered by “lock point”

Example

- T1 transfers $50 from account A to account B
  
  \[
  T1: \text{Read}(A), A := A - 50, \text{Write}(A), \text{Read}(B), B := B + 50, \text{Write}(B)
  \]

- T2 outputs the total of accounts A and B
  
  \[
  T2: \text{Read}(A), \text{Read}(B), \text{PRINT}(A+B)
  \]

- Initially, A = $1000 and B = $2000

- What are the possible output values?
  
  - 3000, 2950, 3050
Is this a 2PL Schedule?

1. Lock_X(A) <granted>
2. Read(A)  
3. A := A-50 
4. Write(A)  
5. Unlock(A) <granted>
6. Read(A) 
7. Unlock(A) 
8. Lock_S(B) <granted>
9. Lock_X(B)  
10. Read(B) 
11. <granted> Unlock(B) 
12. PRINT(A+B) 
13. Read(B) 
14. B := B +50 
15. Write(B) 
16. Unlock(B) 

No, and it is not serializable

Cascading Aborts

• Example: T1 aborts
  – Note: this is a 2PL schedule

  T1: R(A), W(A),  R(B), W(B), Abort
  T2:       R(A), W(A)

• Rollback of T1 requires rollback of T2, since T2 reads a value written by T1

• Solution: **Strict Two-phase Locking (Strict 2PL)**: same as 2PL except
  – All locks held by a transaction are released only when the transaction completes

Strict 2PL (cont’d)

• All locks held by a transaction are released only when the transaction completes

• In effect, “shrinking phase” is delayed until:
  a) Transaction has committed (commit log record on disk), or
  b) Decision has been made to abort the transaction (then locks can be released after rollback)
Is this a Strict 2PL schedule?

1. Lock_X(A) <granted>
2. Read(A)
3. A:= A-50
4. Write(A)
5. Lock_X(B) <granted>
6. Unlock(A) <granted>
7. Read(A)
8. Lock_S(B)
9. Read(B)
10. B:= B +50
11. Write(B)
12. Unlock(B) <granted>
13. Unlock(A)
14. Read(B)
15. Unlock(B)
16. PRINT(A+B)

No: Cascading Abort
Possible

Overview

• Serializability
• The Phantom Issue
• Predicate Locking
• Key-Range Locks
• Next-Key Locking techniques
• Index Management and Transactions
• Multi-level reasoning

Phantom

T1
Select count(*)
where dept = “Acct”
//find and S-lock (“Sue”, “Acct”, 3500) and (“Tim”, “Acct”, 2400)

T2
Insert (“Joe”, “Acct”, 2000)
//X-lock the new record
Commit
//release locks

Select sum(salary)
where dept = “Acct”
//find and S-lock (“Sue”, “Acct”, 3500) and (“Tim”, “Acct”, 2400) and (“Joe”, “Acct”, 2000)
Phantoms and Commutativity

- A predicate-based select doesn’t commute with the insert of a record that meets the select’s where clause
- We need to have some lock to protect the correctness of the result of the where clause
  - Not just the records that are the result!
  - Eswaran et al ‘76 describe (conceptually) locking the records that might exist but don’t do so yet

Page-level locking

- The traditional concurrency control in the 1970s was page-level locking
- If all locks are at page granularity or above, phantoms can’t arise
  - Lock every page read or written (even when page is scanned and no records are found/returned)
  - There are no queries to find a set of pages
- But performance is often poor
  - Lots of false conflicts, low concurrency obtained

Overview

- Serializability
- The Phantom Issue
- Predicate Locking
- Key-Range Locks
- Next-Key Locking techniques
- Index Management and Transactions
- Multi-level reasoning

Predicate Locking

- Solution proposed by Eswaran et al in the 1976 journal paper where they identified and explained the phantom issue
  - And also gave a proof of correctness of 2PL!
  - Context: transactions and serializability were new ideas!
- Never implemented in any system I know of
**Locking Predicates**

- S-Lock the predicate in a where-clause of a SELECT
  - Or a simpler predicate that “covers” this
- X-lock the predicate in a where clause of an UPDATE, INSERT or DELETE

**Conflict decision**

- A lock can’t be granted if a conflicting lock is held already
- For predicates, a Lock on P by T conflicts with Lock on Q by U if
  - Locks are not both S-mode
  - T different from U
  - P and Q are mutually satisfiable
  
  » Some record r could exist in the schema such that P(r) and Q(r)

**An Effective Test for Conflict**

- In general, satisfiability of predicates is undecidable
- Eswaran et al suggest using covering predicates that are boolean combinations of atomic equality/inequalities
  
  \[
  P = \text{(Location = 'Napa' \lor Location = 'Santa Rosa')} \\
  \land (\text{Balance < 500} \land \text{Balance > 10}) \\
  P' = \text{Location = 'Napa' \land Balance = 700}.
  \]
  
  Then the disjunctive normal form of \( P \lor P' \) is
  
  Location = ('Napa' \land \text{Balance < 500} \land \text{Balance > 10} \land \text{Balance = 700})
  
  \lor (\text{Location = 'Santa Rosa'} \land \text{Location = 'Napa'} \land \text{Balance < 500} \land \text{Balance > 10} \land \text{Balance = 700}).

- Satisfiability is a decidable problem, but not efficient

**Implementation Issues**

- Note the contrast to traditional lock manager implementations
  - Conflict is only on lock for same lockname
  - Can be tested by quick hashtable lookup!
Overview

• Serializability
• The Phantom Issue
• Predicate Locking
• Key-Range Locks
• Next-Key Locking techniques
• Index Management and Transactions
• Multi-level reasoning

CS262a Project Proposals

• Two People from this class
  – Projects can overlap with other classes
  – Exceptions to the two person requirement need to be OK’d
• Should be a miniature research project
  – State of the art (can’t redo something that others have done)
  – Should be “systems related”, i.e. dealing with large numbers of elements, big data, parallelism, etc…
  – Should be publishable work (but won’t quite polish it off by end of term)
  – Must have solid methodology!
• Metric of success/base case for measurements
  – Figure out what your “metrics of success” are going to be…
  – What is the base case you are measuring against?
• Project proposals due Friday at midnight – should have:
  – Motivation and problem domain
  – Description of what you are going to do and what is new about it
  – How you are going to do the evaluation (what is methodology, base case, etc.)
  – If you need resources, you need to tell us NOW exactly what they are…
  – List of ALL participants

Key-Range Locks (Lomet’93)

• A collection of varying algorithms/implementation ideas for dealing with phantoms with a lock manager which only considers conflicts on the same named lock
  – Some variants use traditional Multi-Granularity Locking (MGL) modes: IX, IS, SIX, etc.
  – Other dimensions of variation: whether to merge locks on keys, ranges, records
    » Are deleted records removed, or just marked deleted
    » Are keys unique, or duplicatable
Main Ideas

- Avoid phantoms by checking for conflicts on dynamically chosen ranges in key space
  - Each range is from one key that appears in the relation, to the next that appears
- Define lock modes so conflict table will capture commutativity of the operations available
- Conservative approximations: simpler set of modes, that may conflict more often

Range

- If $k_0$ is one key and $k$ is the next, that appear in the relation contents
  - $(k_0, k]$ is the semi-open interval that starts immediately above $k_0$ and then includes $k$
- Name this range by something connected to $k$ (but distinguish it from the key lock for $k$)
  - Example: $k$ with marker for range
  - Or use $k$ for range, Record ID for key itself
- Note: insert or delete will change the set of ranges!

Operations of the storage layer

- Read at $k$
- Update at $k$
- Insert
- Delete
- Scan from $k$ to $k'$ (or fetch next after $k$, as far as $k'$)
  - Note that higher query processing converts complex predicates into operations like these
  » Locks on scan ranges will automatically cover the predicate in the query

Overview

- Serializability
- The Phantom Issue
- Predicate Locking
- Key-Range Locks
- Next-Key Locking techniques
- Index Management and Transactions
- Multi-level reasoning
Current Practice

- Implementations do not use the full flexibility of Lomet’s modes
- Common practice is to use MGL modes, and to merge lock on range with lock on upper key
  - A S-lock on key $k$ implicitly is also locking the range $(k_0,k]$ where $k_0$ is the previous key
  - This is basis of ARIES/KVL

Insertion

- As well as locking the new record’s key, take instant duration IX lock on the next key
  - Make sure no scan has happened that would have showed the non-existence of key just being inserted
  - No need to prevent future scans of this range, because they will see the new record!

Gap Locks

- A refinement S-locks a range $(k_0,k]$ by S-locking the key $k$, and separately it gets a lock on $k$ with a special mode G, that represents the gap – the open interval $(k_0,k)$
- This is used in InnoDB

Overview

- Serializability
- The Phantom Issue
- Predicate Locking
- Key-Range Locks
- Next-Key Locking techniques
- Index Management and Transactions
- Multi-level reasoning
Indices

• Primary index
  – Leaves contain all records with data from table
  – Higher levels contain some records that point to leaf pages or other index pages, with keys to work out which pointer to follow

• Secondary index
  – Leaves contain value of some attribute, and some way to access the records of the data that contain that value in the attribute
    » Eg primary key value, rowid, etc

Problems

• Suppose we don’t do concurrency control on the index structure, but just on the data records (in the leaves)

• Two problems can arise
  – Impossible structure
    » Transaction executes an operation that sees a structure that violates data structure properties
  – Phantom: query with where clause sees the wrong set of values
    » Access through an index must protect against insertion of future matching data record

Mangled Data Structure

Logical Locks and Physical Latches

<table>
<thead>
<tr>
<th></th>
<th>Locks</th>
<th>Latches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separate …</td>
<td>User transactions</td>
<td>Threads</td>
</tr>
<tr>
<td>Protect …</td>
<td>Database contents</td>
<td>In-memory data structures</td>
</tr>
<tr>
<td>During …</td>
<td>Entire transactions</td>
<td>Critical sections</td>
</tr>
<tr>
<td>Modes …</td>
<td>Shared, exclusive, update,</td>
<td>Read, writes,</td>
</tr>
<tr>
<td></td>
<td>intention, escrow, schema, etc.</td>
<td>(perhaps) update</td>
</tr>
<tr>
<td>Deadlock …</td>
<td>Detection &amp; resolution</td>
<td>Avoidance</td>
</tr>
<tr>
<td>… by …</td>
<td>Analysis of the waits-for graph,</td>
<td>Coding discipline,</td>
</tr>
<tr>
<td></td>
<td>timeout, transaction abort,</td>
<td>“lock leveling”</td>
</tr>
<tr>
<td></td>
<td>partial rollback, lock de-escalation</td>
<td></td>
</tr>
<tr>
<td>Kept in …</td>
<td>Lock manager’s hash table</td>
<td>Protected data structure</td>
</tr>
</tbody>
</table>

From Graefe, TODS 35(3):16

Lock: logical level, held for transaction duration
Latch: physical level, held for operation duration
**Latch Coupling**

- When descending a tree
  - Hold latch on parent until after latch on child is obtained
- Exception: if child is not in buffer (it must be fetched from disk)
  - Release latch on parent
  - Return to root, traverse tree again

**Avoiding Undos for Structural Modifications**

- Use System Transactions
  - To ensure recoverability, but avoid lots of unneeded data movement during transaction rollback
- Perform structure modification as separate transaction, outside the scope of the user transaction that caused it
  - Structure modification is logical no-op
  - Eg insert is done by system transaction that splits page; then record is inserted by user transaction into the now-available space

**Overview**

- Serializability
- The Phantom Issue
- Predicate Locking
- Key-Range Locks
- Next-Key Locking techniques
- Index Management and Transactions
- Multi-level reasoning

**Abstraction**

- Data structures can be considered as abstract data types with mathematical values, or as a complex arrangement of objects-with-references
- Example: compare a hash table abstractly as a Map (relating keys and values), or concretely as an array of linked lists
Abstraction

• An operation that changes the logical abstract content is realized by a complex sequence of changes to the objects and references

• The same abstract state can be represented by many different detailed arrangements

Abstraction

• Both concurrency control and recovery can be designed in different ways, depending on what level of abstraction is being considered

• For a DBMS, we can think of a relational table in different levels

Logical View

• Treat the relation as a set of records
• Order not important
• Layout not important

• Example:
  – We log that we executed INSERT (7, fred) into Table57

Physical View

• Treat the relation as a collection of pages whose bits are described

• Example:
  – We log that bytes 18 to 32 in page 17, and bytes 4 to 64 in page 19, were changed as follows…
Physiological View

- Treat the relation as a collection of pages each of which contains a set of records
- Example:
  - We log that in page 17 record (7, fred) was inserted
- “Logical within a page, but physical pages are noticed”
- Enables placing the LSN of relevant log entry into each page

Multi-level Execution

- Top level is a set of transactions
- Next level shows how each transaction is made of logical operations on relations
- Then we see how each logical operation is made up of page changes, each described physiologically
- Lowest level shows operations, each of which has physical changes on the bits of a page

Multi-level Execution

- Lowest level operations are in a total order of real-time
- Higher levels may have concurrency between the operations
  - Deduce this from whether their lowest-level descendants form overlapping ranges in time

Lowest level operations happen in time order as shown
Multi-level Reasoning

- Each level can be rearranged to separate completely the operations of the level above, provided appropriate policies are used
  - Once rearranged, forget there was a lower layer
- If an operation contains a set of children whose combined effect is no-op (at that level), then remove the operation entirely

Multilevel Transaction Management

- Obtain a suitable-mode lock when performing an operation at a level
  - Hold the lock until the parent operation completes
- To abort an operation that is in-progress, perform (and log) compensating operations for each completed child operation, in reverse order

Necessary Properties

- Lock modes
  - If operations at a level are not commutative, then their lock-modes must conflict
- Recovery
  - Performing an operation from a log record must be idempotent
    » Use LSNs etc to restrict whether changes will occur
- Compensators
  - Compensator for an operation must act as its inverse

Defined Properties

- Commutativity
  - O1 and O2 commute if their effect is the same in either order
- Idempotence
  - O1 is idempotent if O1 followed by O1 has the same effect as O1 by itself
- Inverse
  - Q1 is inverse to O1 if (O1 then Q1) has no effect
Lowest level operations happen in time order as shown

Rearrange lowest level, to make next level non-concurrent
Then remove lowest level, and think about level above as single steps

Were these good papers?
- What were the authors’ goals?
- What about the evaluation / metrics?
- Did they convince you that this was a good system /approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the “Test of Time” challenge?
- How would you review this paper today?

References and Further Reading
- Transactional Information Systems, by G. Weikum and G. Vossen, 2002