Concurrent Control in Distributed Databases

In distributed DB

In centralized (Local) DB

Example:

T1: \( \text{Read}(A) \)  \( \text{Write}(A) \)  \( \text{Read}(B) \)  \( \text{Write}(B) \)
\( A \leftarrow A + 100 \)
\( A \leftarrow A \times 2 \)
\( B \leftarrow B + 100 \)
\( B \leftarrow B \times 2 \)
Constraint:  \( A = B \)

Schedule A: Serial Schedule

<table>
<thead>
<tr>
<th></th>
<th>( A )</th>
<th>( B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Read(A); A ← A+100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(A); Read(B); B ← B+100; Write(B);</td>
<td>125</td>
<td>250</td>
</tr>
<tr>
<td>Read(A); A ← A×2; Write(A); Read(B); B ← B×2; Write(B);</td>
<td>125</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>250</td>
</tr>
</tbody>
</table>

Schedule B

<table>
<thead>
<tr>
<th></th>
<th>( A )</th>
<th>( B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Read(A); A ← A+100</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(A); Read(B); B ← B+100; Write(B);</td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A×2; Write(A);</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B×2; Write(B);</td>
<td>250</td>
<td></td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>250</td>
</tr>
</tbody>
</table>
What are good schedules?

- Want schedules that are “good”, regardless of:
  - initial state and
  - transaction semantics
- Only look at order of reads and writes

Example:

\[ S_B = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B) \]

Example (Cont)

\[ S_c = r_1(A)w_1(A)r_2(A)w_2(A)r_2(B)w_2(B)r_1(B)w_1(B) \]

Example (Cont)

\[ S_c = r_1(A)w_1(A)r_2(A)w_2(A) r_2(B)w_2(B)r_1(B)w_1(B) \]

Example (Cont)

\[ S_c = r_1(A)w_1(A)r_2(A)w_2(A) r_2(B)w_2(B)r_1(B)w_1(B) \]

T₁ → T₂
Also, T₂ → T₁

\[ S_c \text{ cannot be rearranged into a serial schedule} \]
\[ S_c \text{ is not “equivalent” to any serial schedule} \]
\[ S_c \text{ is “bad”} \]
**Concepts**

Transaction: sequence of r(x), w(x) actions

Conflicting actions:
- r1(A) w1(A)
- w2(A) r2(A)

Schedule: represents chronological order in which actions are executed

Serial schedule: no interleaving of actions or transactions

Serializable schedule: a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule

**Definition**

S1, S2 are conflict equivalent schedules if S1 can be transformed into S2 by a series of swaps on non-conflicting actions.

A schedule is conflict serializable if it is conflict equivalent to some serial schedule.

**Precedence graph P(S) (S is schedule)**

Nodes: transactions in S

Arcs: Ti → Tj whenever
- p_i(A), q_j(A) are actions in S
- p_i(A) ≤_S q_j(A)
- at least one of p_i, q_j is a write

**Theorem**

P(S1) acyclic ⇐⇒ S1 conflict serializable

**Distributed Transactions**

A distributed transaction T is initiated at one site and spawned subtransactions at several other sites. We distinguish the subtransaction at home site by calling it the coordinator, while the other subtransactions are the participants.
### Distributed Transactions

T₁ in site A, denoted as T₁<sup>A</sup>, is the coordinator.

T₁ in site B, denoted as T₁<sup>B</sup>, is the participant.

T₁<sup>A</sup> waits for T₁<sup>B</sup>, and T₁<sup>B</sup> waits for T₁<sup>A</sup>.

### Example

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Node 1</th>
<th>Node 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(T₁) a ← X</td>
<td>(T₂) c ← X</td>
</tr>
<tr>
<td>2</td>
<td>(T₁) X ← a+100</td>
<td>(T₂) X ← 2c</td>
</tr>
<tr>
<td>3</td>
<td>(T₁) b ← Y</td>
<td>(T₂) d ← Y</td>
</tr>
<tr>
<td>4</td>
<td>(T₁) Y ← b+100</td>
<td>(T₂) Y ← 2d</td>
</tr>
</tbody>
</table>

### Serializability in Distributed DBMS

- Somewhat more involved. Two types of schedules have to be considered:
  - local schedules
  - global schedule

- For global transactions (i.e., global schedule) to be serializable, two conditions are necessary:
  - Each local schedule should be serializable.
  - All sub-transactions of global transactions appear in the same order in the equivalent serial schedule at ALL sites.

### Global Non-serializability

Consider 2 sites and one data item x that is duplicated in both sites

T₁: Read(x) → Write(x) → Commit

T₂: Read(x) → Write(x) → Commit

The following two local schedules are individually serializable (in fact serial), but the two transactions are not globally serializable.

LH₁=⟨R₁(x),W₁(x),R₂(x),W₂(x)⟩
LH₂=⟨R₂(x),W₂(x),R₁(x),W₁(x)⟩

Assume x=1 initially. At site 1, x=60 after LH₁.
At site 2, x=30 after LH₂. Violates the mutual consistency.
How to enforce serializable schedules?
prevent P(S) cycles from occurring

T1  T2  ……  Tn
Scheduler
DB

Classification of Concurrency control Mechanisms

A locking protocol
Two new actions:
lock (exclusive):
unlock:

T1  T2
scheduler
lock

lock table

Locking Rules in centralized db
(2-phase locking)

• Well-formed transactions
• Legal schedulers
• Two-phase transactions
• These rules guarantee serializable schedules

Exercise:

Rule #1: Well-formed transactions
T1: … li(A) … pi(A) … ui(A) …

Rule #2: Legal scheduler
S = ……… li(A) ………… ui(A) ………
no lj(A)

What schedules are legal?
What transactions are well-formed?
S1 = li(A)ri(B)wi(B)li(B)ui(A)ui(B)
ri(B)wi(B)ui(B)
S2 = li(A)ri(B)wi(B)ui(A)ui(B)
lz(B)ri(B)wi(B)ui(B)
S3 = li(A)ri(A)ui(A)ri(B)wi(B)ui(B)
lz(B)ri(B)wi(B)ui(B)

Exercise:

- What schedules are legal?
- What transactions are well-formed?

S1 = l1(A)r1(B)w1(A)l2(B)u1(A)u1(B)  
   r2(B)w2(B)u2(B)l3(B)r3(B)u3(B)

S2 = l1(A)r1(A)w1(A)l2(B)u1(A)u1(B)  
   l2(B)r2(B)w2(B)r3(B)u3(B)

S3 = l1(A)r1(A)u1(A)l1(B)w1(B)u1(B)  
   l2(B)r2(B)w2(B)l3(B)r3(B)u3(B)

Schedule F

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>l1(A);Read(A)</td>
<td>A ← A + 100;Write(A);u1(A)</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>l2(A);Read(A)</td>
<td>B ← Bx2;Write(B);u1(B)</td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>l1(B);Read(B)</td>
<td></td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>B ← B + 100;Write(B);u1(B)</td>
<td></td>
<td>150</td>
<td></td>
</tr>
</tbody>
</table>

Two phase locking (2PL)

for transactions

Schedule G

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l1(A);Read(A)</td>
<td>l1(B);Read(B)</td>
</tr>
<tr>
<td>A ← A + 100;Write(A)</td>
<td>B ← B + 100;Write(B);u1(B)</td>
</tr>
<tr>
<td>l2(A);Read(B)</td>
<td>B ← Bx2;Write(B);u1(B);delayed</td>
</tr>
</tbody>
</table>

Schedule H (T2 reversed)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l1(A);Read(A)</td>
<td>l1(B);Read(B)</td>
</tr>
<tr>
<td>A ← A + 100;Write(A)</td>
<td>B ← Bx2;Write(B);delayed</td>
</tr>
</tbody>
</table>

Deadlocked transactions are rolled back

Theorem

2PL ⇒ conflict serializable schedule
Strict 2PL

Locking-Based Algorithms
- That’s all that is needed to ensure serializability!
- Beyond this is to improve concurrency
  - Locks are either read lock (also called shared lock) or write lock (also called exclusive lock)
  - Read locks and write locks are conflicting (or incompatible)

<table>
<thead>
<tr>
<th></th>
<th>rlock</th>
<th>wlock</th>
</tr>
</thead>
<tbody>
<tr>
<td>rlock</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>wlock</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Beyond this is to improve concurrency.

Read locks and write locks are conflicting (or incompatible).

Locks are either read lock (also called shared lock) or write lock (also called exclusive lock).

Distributed Locking
- Look at three schemes:
  - Centralized locking
  - Primary Copy 2PL
  - Distributed 2PL

Centralized Locking
- Single site that maintains all locking information.
- One lock manager for whole of DDBMS.
- Local transaction managers involved in global transaction request and release locks from lock manager.
- Or transaction coordinator can make all locking requests on behalf of local transaction managers.

Communication Structure of Centralised 2PL

Data processors at participating sites → Coordinating TM → Central site LM

1. lock request
2. lock granted
3. operation
4. end of operation
5. release lock
Primary Copy 2PL
- Lock managers distributed to a number of sites.
- Each lock manager responsible for managing locks for set of data items.
- For replicated data item, one copy is chosen as primary copy, others are slave copies.
- Only need to write-lock primary copy of data item that is to be updated.
- Once primary copy has been updated, change can be propagated to slaves.

Distributed 2PL
- 2PL schedulers are placed at each site.
- Each scheduler handles lock requests for data at that site.
- A transaction may read any of the replicated copies of item x, by obtaining a read lock on one of the copies of x. Writing into x requires obtaining write locks for all copies of x.

Communication Structure of Distributed 2PL

Dealing with Multiple Copies of Data
Three schemes which guarantee that conflicts are discovered at least in one site.

1. Read-lock-one, write-lock-all (ROWA). To read a data item A, a transaction may obtain a read-lock on any copy of A. To write on a data item A, a transaction must obtain write-locks on all copies of A.

Dealing with Multiple Copies of Data

2. The majority locking strategy. To read a data item A, a transaction may obtain read-locks on a majority of the copies of A. To write on a data item A, a transaction must also obtain write-locks on a majority of the copies of A.

3. Primary Copy Locking. All locks for a data item A are requested at the site of the primary copy.

Deadlock and Wait for Graph
- A transaction is deadlocked if it is blocked and will remain blocked until there is intervention.
- Locking-based CC algorithms may cause deadlocks.
- Wait-for graph (WFG).
Local versus Global WFG

- Transaction T1 is initiated at site A and spawned subtransactions at site B. Transaction T2 is initiated at site B and spawned subtransactions at site A.
- Assume T1 waits for a lock held by T2 at site B, and T2 waits for a lock held by T1 at site A.

Site A (no cycle locally)  Site B (no cycle)

Local versus Global WFG

- Transaction T1 is initiated at site A and spawned subtransactions at site B. Transaction T2 is initiated at site B and spawned subtransactions at site A.
- Assume T1 waits for a lock held by T2 at site B, and T2 waits for a lock held by T1 at site A.

Site A  Site B

Deadlock Management

- Ignore
  - use timeout.
- Avoidance
  - detecting potential deadlocks in advance and taking action to ensure that deadlock will not occur.
- Detection and Recovery
  - Allowing deadlocks to form and then finding and breaking them.

Centralized Deadlock Detection

- Single site appointed deadlock detection coordinator (DDC).
- Each scheduler periodically sends its local WFG to the central site.
- DDC has responsibility of constructing and maintaining GWFG.
- If one or more cycles exist, DDC must break each cycle by selecting transactions to be rolled back and restarted.

Hierarchical Deadlock Detection

- Sites are organized into a hierarchy.
- Each site sends its LWFG to detection site above it in hierarchy.
- Reduces dependence on centralized detection site.
Hierarchical Deadlock Detection

- Sites cooperate in detection of deadlocks.
- The local WFGs are formed at each site and passed on to other sites.
- The WFGs are combined into a GWFGs.
- To save cost, only when a potential deadlock exists will the WFG be transmitted.

Distributed Deadlock Detection

- There is a potential global deadlock found in site A, because the WFG in site A has a cycle involving the external node EX.
- Similarly, it is true for site B.

Local versus Global WFG

- All potential global deadlock graphs discovered at site i are forwarded to the site j where there is a transaction T for which site i is waiting.
Distributed Deadlock Detection

Q: How to avoid the same deadlock being detected more than once?

Example

- r6(l), r2(y), r5(m), r1(z), r1(x), r4(b), r3(a), r6(n), r3(c), r5(a), r1(y), r3(x), r6(m), r2(b), r4(n)

Example

- r6(l), r2(y), r5(m), r1(z), r1(x), r4(b), r3(a), r6(n), r3(c), r5(a), r1(y), r3(x), r6(m), r2(b), r4(n)
Size of Data Items

The size or granularity of the data item that can be locked in a single operation has effect on the performance of the concurrency control method.

Granule size can be:
- Tuple
- Page (or bucket)
- Relation

Size of Data Items (Cont)

Consider a transaction which is simply updating a tuple of a relation:
- Tuple – the CC locks only that tuple
- Relation – the CC locks the whole relation

Consider a transaction which is updating many tuples of a relation:
- Tuple – the CC locks each individual tuple separately
- Relation – the CC locks the entire relation

Size of Data Items (Cont)

Ideally, the CC should support mixed granularity with tuple, page and relation level locking.

DBMS will automatically upgrade locks from tuple to page to relation if a particular transaction is locking more than a certain percentage of the tuples or pages in the relation.

Summary

- Data sharing has to be managed to present inconsistencies or other anomalies
- Locking is the most widely used mechanism
- Deadlock has to be managed