Validation

Transactions have 3 phases:

1) Read
   - all DB values read
   - writes to temporary storage
   - no locking

2) Validate
   - check if schedule so far is serializable

3) Write
   - if validate ok, write to DB

Key idea

- Make validation atomic
- If T1, T2, T3, ... is validation order, then resulting schedule will be conflict equivalent to Ss = T1 T2 T3...
To implement validation, system keeps two sets:
- **FIN** = transactions that have finished phase 3 (and are all done)
- **VAL** = transactions that have successfully finished phase 2 (validation)

Example of what validation must prevent:
\[
RS(T_2) = \{B\} \quad RS(T_3) = \{A, B\} \neq \emptyset \\
WS(T_2) = \{B, D\} \quad WS(T_3) = \{C\}
\]

Example of what validation must allow:
\[
RS(T_2) = \{B\} \quad RS(T_3) = \{A, B\} \neq \emptyset \\
WS(T_2) = \{B, D\} \quad WS(T_3) = \{C\}
\]
Another thing validation must prevent:

\[ \text{RS}(T_2) = \{A\} \quad \text{RS}(T_3) = \{A, B\} \]
\[ \text{WS}(T_2) = \{D, E\} \quad \text{WS}(T_3) = \{C, D\} \]

Validation rules for \( T_j \):

1. When \( T_j \) starts phase 1:
   \( \text{ignore}(T_j) \leftarrow \text{FIN} \)

2. at \( T_j \) Validation:
   \( \text{if check } (T_j) \text{ then } \)
   \[ \left[ \text{VAL} \leftarrow \text{VAL} \cup \{T_j\}; \right. \]
   \( \quad \text{do write phase; } \)
   \( \quad \text{FIN} \leftarrow \text{FIN} \cup \{T_j\} \]
Check (T)_i:\n    For Ti \in VAL - IGNORE (T)_i DO\n        IF [ WS(Ti) \cap RS(T_i) \neq \emptyset OR\n            Ti \notin FIN ] THEN RETURN false;\n        RETURN true;\n    Is this check too restrictive ?

Improving Check(T)_i\n    For Ti \in VAL - IGNORE (T)_i DO\n        IF [ WS(Ti) \cap RS(T_i) \neq \emptyset OR\n            (Ti \notin FIN AND WS(Ti) \cap WS(T_i) \neq \emptyset)]\n            THEN RETURN false;\n        RETURN true;

Exercise:

U: RS(U)={A,B} \hspace{1cm} W: RS(W)={A,D}\n    WS(U)={D} \hspace{1cm} WS(W)={A,C}\n
T: RS(T)={A,B} \hspace{1cm} V: RS(V)={B}\n    WS(T)={A,C} \hspace{1cm} WS(V)={D,E}
Validation (also called optimistic concurrency control) is useful in some cases:
- Conflicts rare
- System resources plentiful
- Have real time constraints

Summary

Have studied C.C. mechanisms used in practice
- 2 PL
- Multiple granularity
- Tree (index) protocols
- Validation

Chapter 10  More on transaction processing

Topics (which I doubt we’ll get to all of them):
• Cascading rollback, recoverable schedule
• Deadlocks
  – Prevention
  – Detection
• View serializability
• Long transactions (nested, compensation)
Concurrency control & recovery

Example:

\[ T_j \rightarrow T_i \]

\[ \text{Commit } T_i \]

\[ \text{Abort } T_j \]

- Schedule is conflict serializable
  - \( T_j \rightarrow T_i \)

- But not recoverable

- Need to make "final" decision for each transaction:
  - **commit decision** - system guarantees transaction will or has completed, no matter what
  - **abort decision** - system guarantees transaction will or has been rolled back (has no effect)

\( \text{Cascading rollback} \quad \text{(Bad!)} \)
To model this, two new actions:

- Ci - transaction Ti commits
- Ai - transaction Ti aborts

Back to example:

\[
\begin{array}{c|c|c}
T_i & T_j & W_j(A) \\
\hline
\vdots & \vdots & r(A) \\
\hline
\end{array}
\]

\[\text{Ci} \leftarrow \text{can we commit here?}\]

Definition

Ti reads from Tj in S (Tj \Rightarrow_s Ti) if

1. \(w_j(A) <_S r(A)\)
2. \(a_j <_S r(A)\) (\(<_S \) does not precede)
3. If \(w_j(A) <_S w_k(A) <_S r(A)\) then \(a_k <_S r(A)\)
Definition

Schedule S is recoverable if whenever $T_j \Rightarrow_S T_i$ and $j \neq i$ and $C_i \in S$ then $C_j <_S C_i$

Note: in transactions, reads and writes precede commit or abort

$\Rightarrow$ If $C_i \in T_i$, then $r_i(A) < C_i$

$wi(A) < C_i$

$\Rightarrow$ If $A_i \in T_i$, then $r_i(A) < A_i$

$wi(A) < A_i$

• Also, one of $C_i, A_i$ per transaction

How to achieve recoverable schedules?
With 2PL, hold write locks to commit (strict 2PL)

<table>
<thead>
<tr>
<th>Tj</th>
<th>Ti</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Wj(A)</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Cj</td>
<td>...</td>
</tr>
<tr>
<td>uj(A)</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>rj(A)</td>
</tr>
</tbody>
</table>

With validation, no change!

- S is **recoverable** if each transaction *commits* only after all transactions from which it read have committed.
- S avoids cascading rollback if each transaction may *read* only those values written by committed transactions.
• S is strict if each transaction may read and write only items previously written by committed transactions.

Where are serializable schedules?

Examples

• Recoverable:
  \(-w_1(A) \ w_1(B) \ w_2(A) \ r_2(B) \ c_1 \ c_2\)

• Avoids Cascading Rollback:
  \(-w_1(A) \ w_1(B) \ w_2(A) \ c_1 \ r_2(B) \ c_2\)

  Assumes \(w_2(A)\) is done without reading

• Strict:
  \(-w_1(A) \ w_1(B) \ c_1 \ w_2(A) \ r_2(B) \ c_2\)
Deadlocks
- Detection
  - Wait-for graph
- Prevention
  - Resource ordering
  - Timeout
  - Wait-die
  - Wound-wait

Deadlock Detection
- Build Wait-For graph
- Use lock table structures
- Build incrementally or periodically
- When cycle found, rollback victim

Resource Ordering
- Order all elements $A_1, A_2, ..., A_n$
- A transaction $T$ can lock $A_i$ after $A_j$ only if $i > j$

Problem: Ordered lock requests not realistic in most cases
Timeout

- If transaction waits more than $L$ sec., roll it back!
- Simple scheme
- Hard to select $L$

Wait-die

-Transactions given a timestamp when they arrive \( \ldots \) $ts(T_i)$
-\( T_i \) can only wait for $T_j$ if $ts(T_i) < ts(T_j)$ \( \ldots \) else die

Example:

\[ T_1 \quad (ts = 10) \quad \text{wait} \quad T_2 \quad (ts = 20) \quad \text{wait} \quad T_3 \quad (ts = 25) \]
Wound-wait

- Transactions given a timestamp when they arrive ... ts(T_i)
- Ti wounds T_j if ts(T_i) < ts(T_j)
  else Ti waits

"Wound": T_j rolls back and gives lock to T_i