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 $T_1, T_2, T_3, \ldots$ is validation order, then resulting schedule will be conflict equivalent to $S_S = T_1 T_2 T_3\ldots$
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:

\[
\begin{align*}
RS(T_2) &= \{B\} \\
WS(T_2) &= \{B,D\} \\
T_2 \text{ start} &\quad T_2 \text{ validated} \\
T_3 \text{ start} &\quad T_3 \text{ validated} \\
\end{align*}
\]

Example of what validation must prevent:

\[
\begin{align*}
RS(T_2) &= \{B\} \quad RS(T_3) = \{A,B\} \neq \emptyset \\
WS(T_2) &= \{B,D\} \quad WS(T_3) = \{C\} \\
T_2 \text{ start} &\quad T_2 \text{ validated} \\
T_3 \text{ start} &\quad T_3 \text{ validated} \\
\end{align*}
\]
Another thing validation must prevent:

\[
\begin{align*}
RS(T_2) &= \{A\} & RS(T_3) &= \{A, B\} \\
WS(T_2) &= \{D, E\} & WS(T_3) &= \{C, D\}
\end{align*}
\]

BAD: \( w_3(D) \)  \( w_2(D) \)

Validation rules for \( T_j \):

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

2. at \( T_j \) Validation:
   
   \[
   \begin{align*}
   \text{if check } (T_j) & \text{ then} \\
   & [ \text{ VAL } \leftarrow \text{ VAL } U \{T_j\}; \\
   & \text{ do write phase;} \\
   & \text{FIN } \leftarrow \text{FIN } U \{T_j\} ]
   \end{align*}
   \]
Check (T_j):
   For T_i ∈ VAL - IGNORE (T_j) DO
       IF [ WS(T_i) ∩ RS(T_j) ≠ ∅ OR T_i ∉ FIN ] THEN RETURN false;
       RETURN true;
   Is this check too restrictive?

Improving Check(T_j)
   For T_i ∈ VAL - IGNORE (T_j) DO
       IF [ WS(T_i) ∩ RS(T_j) ≠ ∅ OR (T_i ∉ FIN AND WS(T_i) ∩ WS(T_j) ≠ ∅) ] THEN RETURN false;
       RETURN true;

Exercise:
   U: RS(U)={B}  \(\triangleright\) start
   WS(U)={D}  \(\triangleright\) validate
   W: RS(W)={A,D}  \(\star\) finish
   WS(W)={A,C}
   T: RS(T)={A,B}  \(\triangleleft\) start
   WS(T)={A,C}
   V: RS(V)={A}  \(\triangleright\) validate
   WS(V)={B}  \(\star\) finish
   W: RS(W)={A,D}  \(\star\) finish
   WS(W)={A,C}
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: \( T_j \quad T_i \)

\[ \begin{align*}
& W_j(A) \quad W_i(A) \\
& r_j(A) \quad r_i(A) \\
& \text{Commit } T_i \\
& \text{Abort } T_j \\
\end{align*} \]

\( \rightarrow \) Cascading rollback (Bad!)

- 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)
To model this, two new actions:

- $C_i$ - transaction $T_i$ commits
- $A_i$ - transaction $T_i$ aborts

Back to example:

\[ \begin{array}{c|c|c}
T_i & T_i & \vdots \\
\hline
W_j(A) & r_i(A) & \vdots \\
& & C_i \leftarrow \text{can we commit here?}
\end{array} \]

Definition

$T_i$ reads from $T_j$ in $S$ ($T_j \Rightarrow_S T_i$) if

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

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

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

$\Rightarrow^*$ If $C_i \in T_i$, then $r(A) < C_i$
  $w(A) < C_i$

$\Rightarrow^*$ If $A_i \in T_i$, then $r(A) < A_i$
  $w(A) < A_i$

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

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

\[
\begin{array}{cccc}
T_j & T_i & \vdots & \\
W_j(A) & \vdots & \\
C_j & \vdots & \\
u_j(A) & \vdots & r_j(A)
\end{array}
\]

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:
  – w₁(A) w₂(B) w₂₁(A) r₂(B) c₁ c₂
• Avoids Cascading Rollback:
  – w₁(A) w₁(B) w₂₁(A) c₁ r₂(B) c₂
• Strict:
  – w₁(A) w₁(B) c₁ w₂₁(A) r₂(B) c₂
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 A₁, A₂, ..., Aₙ
- A transaction T can lock Aᵢ after Aⱼ 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.... ts(Ti)
- Ti can only wait for Tj if ts(Ti) < ts(Tj)
  ...else die

Example:

T1
(ts = 10)

T2
(ts = 20)

T3
(ts = 25)

wait

wait
Wound-wait

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

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

User/Program commands

Lots of variations, but in general
- Begin_work
- Commit_work
- Abort_work

Nested transactions

User program:
  :
  Begin_work;
  :
  :
  If results_ok, then commit work
  else abort_work
Nested transactions

User program:

\begin{verbatim}
Begin_work;
    Begin_work;
        If results_ok, then commit work
        else {abort_work; try something else...}
    If results_ok, then commit work
    else abort_work
\end{verbatim}

Parallel Nested Transactions

\begin{verbatim}
T_1: begin-work
parallel:
    T_{11}: begin_work
        commit_work
    T_{12}: begin_work
        commit_work
        commit_work
\end{verbatim}

Locking

\begin{verbatim}
What are we really locking?
\end{verbatim}
Example:

T₁
  :  
  Read record r₁
  :  
  Read record r₁  do record locking
  :  
  Modify record r₃
  :

But underneath:

If we lock all data involved in read of R₁, we may prevent an update to R₂ (which may require reorganization within block)

Solution: view DB at two levels

Top level: record actions
  record locks
  undo/redo actions — logical

  e.g., Insert record(X,Y,Z)
  Redo: insert(X,Y,Z)
  Undo: delete
Low level: deal with physical details
  latch page during action
  (release at end of action)

Note: undo does not return physical DB
to original state; only same logical state

e.g.,  Insert R3  Undo  (delete R3)

Related idea: Sagas
  • Long running activity: \( T_1, T_2, \ldots, T_n \)
  • Each step/transaction \( T_i \) has a compensating transaction \( T_i-1 \)
  • Semantic atomicity: execute one of
    - \( T_1, T_2, \ldots, T_n \)
    - \( T_1, T_2, \ldots, T_{n-1}, T_{n-1}^{-1}, T_{n-2}^{-1}, \ldots, T_1^{-1} \)
    - \( T_1, T_2, \ldots, T_{n-2}, T_{n-2}^{-1}, T_{n-3}^{-1}, \ldots, T_1^{-1} \)
    - \( T_1, T_{n-1}^{-1}, \ldots, T_1^{-1} \)
    - nothing
Summary

- Cascading rollback
  - Recoverable schedule
- Deadlock
  - Prevention
  - Detection
- Nested transactions
- Multi-level view