Distributed Optimistic Algorithm

- **Assumptions**
  1. Synchronized clocks
  2. MTD (Maximum Transition Delay) can be defined

- **Step 1:** Read

- **Step 2:** Compute

- **Step 3:** Transaction is broadcasted to all nodes at time $\pi(V_i)$ (time when computation finishes and $T_i$ is ready for validation)

- **Step 4:** At time $\pi(V_i) + \text{MTD}$, all nodes start validation of $T_i$. (Note $\pi(V_i)$ is attached to $T_i$) and if $T_i$ reaches before $\pi(V_i) + \text{MTD}$, it must wait
Distributed Optimistic Algorithm

- Step 5:
  IF validation succeeds, all nodes write $S(w_i)$
  ELSE all nodes except “X” ignore $T_i$
  At node X, $T_i$ is restarted and repeated until $T_i$ validates

THEOREM:
The dist. opt. algorithm produces only correct histories at each node and all histories are identical.

PROOF:
ONLY correct histories are produced. Because of Theorem 1
ELSE UPDATE $S(R_i)$ and repeat from step 2
Centralized Optimistic Algorithm

A node (C) is chosen as central node

**CASE 1: Validation takes place only at central node**

*When $T_i$ arrives at a node “X”*

1. Read $S(R_i)$
2. Execute (compute) and get $S(w_i)$

Note $S(w_i)$ is semantic write set (actual)

- $T_i$ goes to node C (if $X \neq C$)
- If $T_i$ succeeds, send write set to all nodes
CASE 2: Validation takes place at local node and then at central node

1. Same
2. Same
3. $T_i$ validates at X
4. IF successful, $T_i$ commits at X and is sent to C
5. ELSE UPDATE $S(R_i)$ and repeat from step 2
6. If successful at C, send write set to all nodes
   ELSE UPDATE $S(R_i)$ at C and execute at C and repeat validation until successful.
Centralized Optimistic

CASE 1: Validation takes place only at central node only

CASE 2: Validation takes place at local node and then central node

Distributed Optimistic
Validation takes place at all nodes after a delay of MTD (Max. transmission Delay)
When to synchronize (assert concurrency control)

- First access to an entity
  (locking, pessimistic validation)
- At each access
  (granularity level)
- After all accesses and before commitment
  (optimistic validation)
Information needed for synchronization

- Locks on database entities
  (system R, INGRES, Rosenkrantz, ...)
- Time stamps on database entities
  (Thomas, Reed, ...)
- Time stamps on transactions
  (Kung, SDD-1, Schlageter, ...)

OBSERVATIONS

- Time stamps more fundamental than locking
- Time stamps carry more information
- Time stamp checking costs more than checking locks
Distributed DBMS

Optimistic CC.

History

Serial

\[
\begin{align*}
T_1 & : X \leftarrow X + 1 \\
T_{11} : X & \leftarrow X + 1 \\
T_{12} : X & \leftarrow 2 \times X \\
T_2 & : X \leftarrow X + 1 \\
T_{21} : X & \leftarrow X + 1
\end{align*}
\]

\[f: \text{Herbrand fn.}\]

non serializable

\[
\begin{align*}
T_{11}, T_{21}, T_{12} & \\
f_{12}(f_{21}(f_{11}(x))) & \\
f_{21}(f_{12}(f_{11}(x)))
\end{align*}
\]

So given interpretation of \( f_{ij} \)'s allows us to include histories which are not allowed by SERIALIZABILITY and hence allows us higher concurrency.
Transaction (Ti) arrives

Obtain S(Ri)

I/O Q (Lo)

Read S(Ri)

CPUQ

Execute
Obtain S(Wi) and Validate
CPU

Local Validation Successful?

Send transaction to other sites

Global Validation Successful?

Yes

No

Transaction (Ti) Finishes

I/O Q (Hi)

Write S(Wi)

Figure 2
Obtain locks from lock table for S(Ri) and S(Wi)

Obtain locks

C.P.U.

Execute

Release locks

Write S(Wi)

I/O

Read S(Ri)

Done

Arrive

Obtain S(Ri)

S(Wi)

Locks granted

No

Yes

CPUQ (low)

CPUQ (med)

CPUQ (Hi)

IOQ (Hi)

IOQ (Hi)
Steps of a Transaction (Ti) Non-Locking Algorithm

1. The transaction (Ti) arrives in the system

2. The read $S'(R_i)$ and write $S'(W_i)$ set of the transaction is obtained. These sets are syntactic

3. The transaction goes to an I/O queue to obtain item values for read set $S(R_i)$

4. The transaction goes to CPU queue and completes execution to obtain write set values. Also actual read set $S(R_i)$ and write set $S(w_i)$ are determined. These sets represent semantic information

5. The transaction’s read sets are validated against other active transactions according consistency constraints (such as serializability)
Steps of a Transaction (Ti) ... (cont)

6. If validation fails due to conflict among transaction Ti and some other transaction Tj, then one of the transaction is required to repeat its execution. For example, if consistency constraint is “strongly serializable”, then the transaction that arrived later (let us say Ti) is selected for re-execution. Moreover the conflict among Ti and Tj is resolved and the values of S'(Ri) are updated with values from S(Wj) at the time of validation. This is useful because Ti does not have to go and do its I/O once again.

7. The transaction is sent to CPU queue to do its computation.

8. The transaction Ti’s write set is validated against write set of some transaction Tj (that has not completed but arrived before Ti). If conflict occurs, then Ti is delayed and writes after Tj writes in the database.
9. The transaction goes to an I/O queue and update its write set $S(W_i)$.

10. The transaction $T_i$ waits in memory for validation against transactions that arrived in the interval between its arrival time and validation time.
Performance Techniques

- Complexity
- Analytical
- Simulation
- Empirical
Parameters

1. Arrival rate
2. Base set (size of write set/read)
4. Size of database
5. Number of sets
6. Transmission delay
7. CPU time
8. I/O time
9. Retry delay
10. Read only trans/write & read trans ratio
11. Multiprogramming level
12. Degree of conflict