Variants of OCC

Dynamic Adjustment & TicToc

Dynamic Adjustment

- Based on the Forward Validation scheme i.e., validate against concurrent live transactions.
- Number of aborts is reduced by using dynamic adjustment of serialization order. Supported by the use of dynamic timestamp assignment scheme.
- Serialization order of committed transactions may be different from their commit order.

Validation scheme

- Suppose we have a validating transaction T_v and a set of active transactions T_j (j = 1, 2, …, n, j ≠ v).
- There are three possible types of data conflicts which can induce serialization order between T_v and T_j:
  - RS (T_v) ∩ WS (T_j) ≠ φ (write-read conflict)
    Write-read conflict can be resolved by adjusting the serialization order as T_v -> T_j so that the read of T_v cannot be affected by T_j's write. Forward adjustment.

  - WS (T_v) ∩ RS (T_j) ≠ φ (read-write conflict)
    Read-write conflict can be resolved by adjusting the serialization order as T_j -> T_v so that the read of T_j cannot be affected by T_v's write. Backward adjustment.

  - WS (T_v) ∩ WS (T_j) ≠ φ (write-write conflict)
    Write-write conflict can be resolved by adjusting the serialization order as T_v -> T_j so that the write of T_v cannot overwrite T_j's write. Forward adjustment.

Implementation details

- SOT (T_i)
  - To indicate the relative serialization order of the transactions.
  - Initial value = ∞ (INT_MAX)
  - If SOT (T_i) < ∞, it means it has been backward adjusted before a committed transaction.
- ATS (T_v)
  - When T_v comes to validation, the set of active transactions T_i such that SOT (T_v) >= SOT (T_i).
- BTS (T_v)
  - Set of active transactions T_j such that SOT (T_j) < SOT (T_v)
- TR (T_v, D_p) = WTS (D_p) when T_v read D_p.

Validation Part 1

- First part of Validation test is used only for those validating transactions which have been backward adjusted. It is to check whether:
  - All the read operations of T_v have been read from the committed transactions T_c whose SOT (T_c) < SOT (T_v)
  - Whether T_v's write is invalidated. This is done by comparing SOT (T_v) with WTS (D_p) and RTS (D_p) of the data items D_p in T_v's write set.

```
part_one(T_v)
{
    if  SOT(T_v) ≠ ∞  then
    {
        for  ∀ D_p ∈ RS(T_v)
        {
            if  TR(T_v, D_p) > SOT(T_v)  then
                restart( T_v );
        }

        for  ∀ D_p ∈ WS(T_v)
        {
            if  SOT(T_v) < RTS(D_p)  or  SOT(T_v) < WTS(D_p)  then
                restart( T_v );
        }
    }
}
```

Validation Part 2

- The purpose of part two is to detect read-write conflicts between active transactions and the validating transactions.
- Here we compare the WS (T_v) with the RS (T_i) where T_i ∈ ATS (T_v)
- The conflicting transactions T_i are added to BTlist (T_v) to indicate that T_j needs to be backward adjusted before T_v.

```
part_two(T_v)
{
    BTlist(T_v) = ∅ ;

    for ∀ T_j ∈ ATS(T_v)
    {
        for ∀ D_p ∈ WS(T_v)
        {
            if D_p ∈ RS(T_j) then
                BTlist(T_v) = BTlist(T_v) ∪ T_j ;
        }
    }
}
```

Validation Part 3

- The third part of the test is to detect whether a backward-adjusted transaction T_j also needs forward adjustment w.r.t T_v
- Compares the RS (T_v) with WS (T_j) where T_j ∈ BTS (T_v) or BTlist (T_v)
- Compares the WS (T_v) with WS (T_j) where T_j ∈ BTS (T_v) or BTlist (T_v)
- If either one of them is non-empty, T_j has serious conflict with T_v.
- Conflict resolution can be implemented to abort transactions based on priority.
- In our code, we abort the current validating transaction T_v.

```
part_tree(T_v)
{
    for  ∀ T_j ∈ BTS(T_v) ∪ BTlist(T_v)
    {
        for  ∀ D_p ∈ RS(T_v)
        {
            if  D_p ∈ WS(T_j)  then
                conflict_resolution(T_v,T_j);
        }

        for  ∀ D_p ∈ WS(T_v)
        {
            if  D_p ∈ WS(T_j)  then
                conflict_resolution(T_v,T_j);
        }
    }
}
```

Validation Part 4

- When the validating transaction reaches part 4, it is guaranteed to commit.
- The purpose is to assign a final commitment timestamp to the validating transaction and to update the necessary timestamps of the data items.

part_four(T_v)
{
  if $SOT(T_v) = \infty$ **then**
    $SOT(T_v) = validation\_time$;

  for $\forall\ T_j \in BTlist(T_v)$
    $SOT(T_j) = SOT(T_v) - \epsilon$; //infinitesimal quantity

  for $\forall\ D_p \in RS(T_v)$
    $RTS(D_p) = SOT(T_v)$;

  for $\forall\ D_p \in WS(T_v)$
    $WTS(D_p) = SOT(T_v)$;

  commit $WS(T_v)$ to database;
}

TicToc: Time Travelling OCC

- Traditional OCC algorithms assign static timestamps, essentially agreeing on a fixed sequential schedule -> eases conflict detection, but limits concurrency.
- TicToc does not allocate static timestamps -> does not restrict the set of potential orderings.
- Instead, a transaction’s timestamp is calculated lazily at its commit time in a distributed manner based on the tuples it accesses.
  - Distributed nature -> avoids all of the bottlenecks inherent in centralized timestamp allocation schemes -> highly scalable algorithm.
  - Laziness allows the DBMS to exploit more parallelism in the workload -> reduces aborts and improves performance.

Example

- Consider the following example involving two concurrent transactions, A and B, and two tuples, x and y. The transactions invoke the following sequence of operations:
  - A read (x)
  - B write (x)
  - B commits
  - A write (y)
- If ts (A) < ts (B), then A can commit since the interleaving of operations is consistent with the timestamp order.
- However if ts (A) > ts (B), A must eventually abort since committing it would violate the schedule imposed by timestamp order.

Lazy Timestamp Management

- To encode the serialization information in the tuples, each data version has a valid range of timestamps bounded by the write timestamp (wts) and the read timestamp (rts) -> a particular version is created at timestamp wts and is valid until timestamp rts.
- A version read by a transaction is valid iff that transaction's commit timestamp is in between the version's wts and rts.
- A write by a transaction is valid iff the transaction's commit timestamp is greater than the rts of the previous version.

$$
\begin{aligned}
&\exists\ commit\_ts, \\
&(\forall\ v \in \{versions\ read\ by\ T\}, v.wts \leq commit\_ts \leq v.rts) \\
&\wedge (\forall\ v \in \{versions\ written\ by\ T\}, v.tuple.rts < commit\_ts)
\end{aligned}
$$

Implementation details

- A read set (RS) and write set (WS) of tuples for each transaction.
- Each entry in the RS and WS is encoded as {tuple, data, wts, rts}
  - tuple is a pointer to the tuple in the database
  - data is the data value read by the transaction
  - wts and rts are the timestamps copied from the tuple when it was accessed by the transaction.
- The value and timestamps must be loaded atomically to guarantee that the value matches timestamps.

Algorithm 1: Read Phase

Data: read set RS, tuple t

$r = RS.get\_new\_entry()$

$r.tuple = t$

 # Atomically load wts, rts, and value

$< r.value = t.value,\ r.wts = t.wts,\ r.rts = t.rts >$

Validation Phase

- In the validation phase, TicTic uses the timestamps stored in the transaction’s read and write sets to compute its commit timestamp.
- Then, the algorithm checks whether the tuples in the transaction’s read set are valid based on this commit timestamp.
- Step 1: Lock all the tuples in the transaction’s WS in their primary key order -> no concurrent updates and no deadlocks.

Algorithm 2: Validation Phase

```
Data: read set RS, write set WS
# Step 1 – Lock Write Set
for w in sorted(WS) do
    lock(w.tuple)
end
# Step 2 – Compute the Commit Timestamp
commit_ts = 0
for e in WS ∪ RS do
    if e in WS then
        commit_ts = max(commit_ts, e.tuple.rts +1)
    else
        commit_ts = max(commit_ts, e.wts)
    end
end
#  Step 3 – Validate the Read Set
for r in RS do
    if r.rts < commit_ts then
        # Begin atomic section
        if r.wts ≠ r.tuple.wts or (r.tuple.rts ≤ commit_ts and
        isLocked(r.tuple) and r.tuple not in W) then
            abort()
        else
            r.tuple.rts = max(commit_ts, r.tuple.rts)
        end
        # End atomic section
    end
end
```

Validating the tuples in the Read Set

- If the transaction's commit_ts <= rts of the read set entry, then the invariant holds and no further action.
- If the entry's rts < commit_ts, it is not clear whether the local value is still valid or not at commit_ts.
  - If another transaction has modified the tuple at a logical time between the local rts and commit_ts -> the transaction has to abort. Otherwise, if no transaction has modified the tuple, rts can be extended to be greater than or equal to commit_ts, making the version valid at commit_ts. To check this, compare the local wts to the latest wts.
  - If wts matches, but the tuple is already locked by a different transaction, it is not possible to extend the rts either.
  - If the rts is extensible or if the version is already valid at commit_ts, the rts of the tuple can be extended to at least commit_ts.

Write Phase

Algorithm 3: Write Phase

Data: write set *WS*, commit timestamp *commit_ts*

for *w in WS* **do**

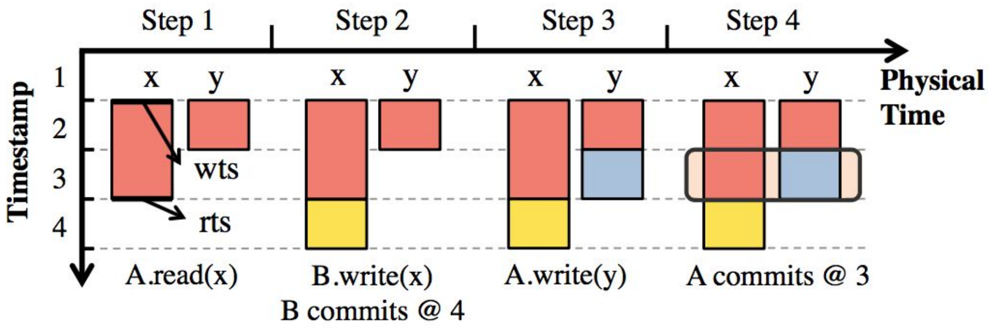
  write(w.tuple.value, w.value)

  w.tuple.wts = w.tuple.rts = commit_ts

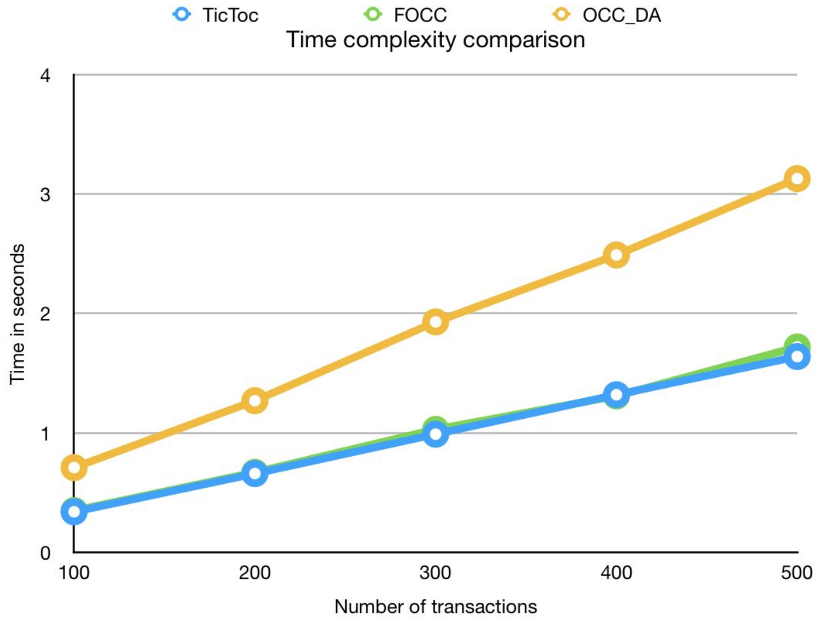
  unlock(w.tuple)

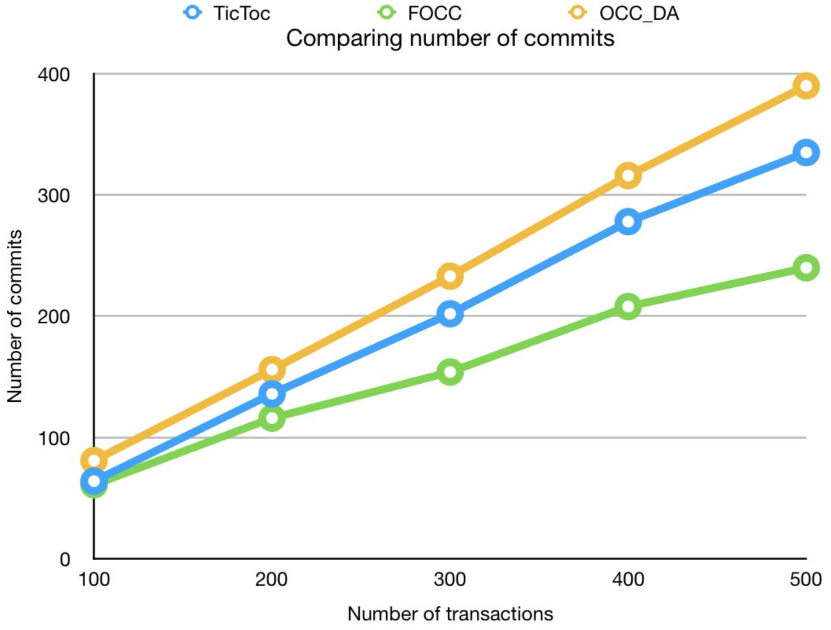
end

Example



Comparison with Vanilla FOCC





Thank You!