Today’s Plan

- Last week’s homework
- Basic Timestamp Ordering (BTO), Serialization Graph Testing (SGT)
- Backward- and forward-oriented validation (BOCC, FOCC)
- Multiversion Concurrency Control (MVCC)
- Homework
(1) Extending 2PL to avoid starvation (your ideas):

- count aborts, use as a first victim criterion
- remember and reserve locks for the starving transaction
- C2PL and a sorted lock order
- random victim criterion
- O2PL avoids at least some deadlocks (but not all!)

- when the victim criterion kills old transactions: reset the timestamp when a transaction is restarted
- when the victim criterion kills new transactions: keep the old timestamp when a transaction is restarted
BTO, SGT, BOCC, FOCC, MVCC

- Credits: Dr. Andrey Gubichev, 2013
(Basic) Timestamp ordering

- Non-locking algorithm (unlike previous ones)
- Transactions get timestamps $ts$ (time of beginning, number)
- TO rule: if $p_i(x)$ is in conflict with $q_j(x)$, then $p_i(x)$ is executed before $q_j(x)$ iff $ts(T_i) < ts(T_j)$
The scheduler executes the operations right away, except the operation is too late.

Operation $p_i(x)$ is too late, if the conflicting operation $q_j(x)$ of a later transaction has already been executed:

- $p_i(x)$ and $q_j(x)$ are in conflict
- $q_j(x)$ has been executed
- but $ts(T_j) > ts(T_i)$
- $p_i(x)$ is too late!
For every $x$ record

- $\text{max-r-scheduled}(x)$ – max timestamp of the read operation sent to data manager so far
- $\text{max-w-scheduled}(x)$ – max timestamp of the write operation sent to data manager so far
Example

- $ts(T_1) = 1$, $ts(T_2) = 2$, $ts(T_3) = 3$
- $s = r_1(x)w_2(x)r_3(y)w_2(y)w_3(z)c_3r_1(z)$
Example

- \( ts(T_1) = 1, \ ts(T_2) = 2, \ ts(T_3) = 3 \)
- \( s = r_1(x)w_2(x)r_3(y)w_2(y)a_2w_3(z)c_3r_1(z)a_1 \)
SGT: Overview

- Goal: keep the CG acyclic
- Maintain a graph in which nodes and edges are added dynamically
A new operation $p_i(x)$ arrives:

- create a node for $T_i$ if it does not exist
- insert edges $(T_j, T_i)$ for all transactions in conflict
  - There is a cycle. Reject $p_i(x)$, abort $T_i$
  - No cycle. Add the edge permanently
SGT: Discussion

- Graph tends to grow over time
- Can we remove information about old transactions (committed long time ago)?
Graph tends to grow over time

Can we remove information about old transactions (commited long time ago)?

\[ r_{k+1}(x) w_1(x) w_1(y_1) c_1 \underbrace{w_2(x) w_2(y_2) c_2 \ldots w_k(x) w_k(y_k) c_k}_{\text{all of } t_k} w_{k+1}(z) \ldots \]

Need to check whether \( z \in \{x, y_1, \ldots, y_k\} \)
SGT: Discussion

- Graph tends to grow over time
- Can we remove information about old transactions (committed long time ago)?

\[
\begin{align*}
    r_{k+1}(x) \cdot & w_1(x)w_1(y_1)c_1 \cdot w_2(x)w_2(y_2)c_2 \cdots \cdot w_k(x)w_k(y_k)c_k \cdot w_{k+1}(z) \cdots \notag
    \end{align*}
\]

- Need to check whether \( z \in \{x, y_1, \ldots, y_k\} \)
- if \( T_i \) is finished and is a source (no incoming edges), it can be safely removed (incoming edges are never added afterwards)
Optimistic protocols: main ideas

- Assume conflicts are rare
- Let the arriving operations simply pass, but sometimes check whether the schedule is serializable
- Checking (*validation*) is part of the transaction
- Transaction has three phases:
  - Read phase: all writes are applied to the private workspace (not to the database). No one else can see them
  - Validate phase: ready to commit. Was the execution OK (CSR?)
  - Write phase: workspace is transferred into the database
- Atomic *Val-Write* phase
Validation approaches

- **backward-oriented (BOCC):** A transaction under validation executes a conflict test against the already committed transactions.
- **forward-oriented (FOCC):** T is validated against the transactions that run in parallel, but are still in their read phase.
- Both guarantee that the CG is acyclic.
$T_j$ is under validation. For every committed $T_i$ check:

- $T_i$ ended before $T_j$ started.
- $RS(T_j) \cap WS(T_i) = \emptyset$ and Val-Write($T_i$) ended before Val-Write($T_j$) ($T_j$ does not read from $T_i$)

If not: abort $T_j$
BOCC Example

```
read phase
\[ t_1 \]
\[ r_1(x) \quad r_1(y) \quad \text{val.} \quad w_1(x) \]

write phase
\[ t_2 \]
\[ r_2(y) \quad r_2(z) \quad \text{val.} \quad w_2(z) \]

\[ t_3 \]
\[ r_3(x) \quad r_3(y) \quad \text{val.} \quad \text{abort} \]

\[ t_4 \]
\[ r_4(x) \quad \text{val.} \quad w_4(x) \]
```
$T_j$ is under validation at the time $n$. For every running $T_i$, $RS^n(T_i)$ is the read set of $T_i$ at the moment $n$. Check:

- For all $T_i$ reading at the current moment $n$:
  \[ WS(T_j) \cap RS^n(T_i) = \emptyset \]

If not:

- abort $T_i$
- abort $T_j$ for which the intersection is non-empty
- wait with validation
FOCC Example

\[ \begin{align*}
\text{read phase} & \quad \text{write phase} \\
& \quad \begin{array}{c}
\text{r}_1(x) \quad \text{r}_1(y) \\
\text{val.} \quad \text{w}_1(x)
\end{array}
\end{align*} \]

\[ \begin{align*}
t_1 & \quad \begin{array}{c}
\text{r}_2(y) \quad \text{r}_2(z) \quad \text{val.} \quad \text{w}_2(z)
\end{array}
\end{align*} \]

\[ \begin{align*}
t_3 & \quad \text{abort}
\end{align*} \]

\[ \begin{align*}
t_4 & \quad \begin{array}{c}
\text{r}_4(x) \quad \text{r}_4(y) \quad \text{val.} \quad \text{w}_4(y)
\end{array}
\end{align*} \]

\[ \begin{align*}
t_5 & \quad \begin{array}{c}
\text{r}_5(x) \quad \text{r}_5(y)
\end{array}
\end{align*} \]
Multiversion concurrency control: ideas

- Write operations are no longer “in place”. Each write operation creates a version of the data item
- Old values are always accessible
- Scheduler has to decide which version to read
BTO, SGT, BOCC, FOCC, MVCC

- Credits: Dr. Andrey Gubichev, 2013
Homework

- Already uploaded to our website.