Advanced Concurrency Control and Transaction Models
Nested Transactions

(a) Flat transaction

(b) Flat transaction

(c) Nested transactions
Decentralized execution is complex:

- Object A calls object B: object-B call is subtransaction. Object B now calls object C, and C calls again A. A must detect that it is the same transaction.

```
T

A.m1(XID) → Θ_A

m1{
    ...
    B.m2
    ...
}

B.m2(XID) → Θ_B

A.m4(XID) → C.m3(XID)

C.m3(XID) → Θ_C
```
A nested transaction is a tree of transactions
Transactions at the leaves are flat transactions
A transaction can be parent of many transactions and is the subtransaction (child-transaction) of its parent.
The main concepts behind nested transactions are in regard to atomicity. (will discuss later)
Execution semantics:

- Everything is executed serially:
  - Begin T, begin T1, begin/end T11, begin/end T12, end T1, begin T2, end T2, end T
Isolation with centralized execution

- Serial execution (parent waits until child finishes, siblings do not execute in parallel)
  - Parent gives child all its locks
  - Child returns all locks (old locks and newly acquired locks) to parent
  - Implementation if the underlying data source is relational database system supporting only flat transactions:
    - All subtransactions must appear as one flat transaction to the database in order to share locks
    - Problem: relational db do not provide application with transaction ID (xid are purely internal).
      - Instead DBS assigns txn to connections
        - All requests must be filtered through same connection?
      - XA-interface: advanced database interface that provides transaction identifier.
Optimistic CC

- Locking is conservative
  - Locking overhead even if no conflicts
  - Deadlock detection/resolution (especially problematic in distributed environment)

- Optimistic concurrency control
  - Perform operation first
  - Check for conflicts only later (e.g., at commit time)

- Centralized systems never used optimistic CC
- Interesting alternative for distributed environment
Kung-Robinson Model

- **Working Phase:**
  - If first operation on X, then load last committed version from DB and cache, read/write cached version
  - Otherwise read/write cached version
  - Keep WriteSet containing objects written
  - Keep ReadSet containing objects read

- **Validation Phase**
  - Check whether transaction conflicts with other transactions

- **Update Phase**
  - Upon successful validation, cached version of updated objects are written back to DB (= changes are made public)

- **Assumption (for simplicity):**
  - Only one transaction may be in validation phase and update phase at a time (e.g., implemented as critical section)
Example
Backward Validation

- Maintain transaction counter $T_n$, initialize $T_n := 0$
- Upon begin of transaction $T_i$ (could be first read/write request)
  - $\text{Start}T_n(T) := T_n$
  - $\text{WriteSet}(T) := \text{RreadSet}(T) := \{\}
- Upon $\text{ri}(x), \text{wi}(x)$ request of $T_i$
  - If first operation on $X$, load last committed version from DB and cache,
  - read/write cached version of $X$
  - If $\text{ri}(x)$, then $\text{ReadSet}(T_i) := \text{ReadSet}(T_i) \cup \{x\}$
  - If $\text{wi}(x)$, then $\text{WriteSet}(T_i) := \text{WriteSet}(T_i) \cup \{x\}$
- At time of validation of transaction $T_i$
  - For all $\text{Start}T_n(T) < j \leq T_n$
    - Let $WS$ be the write set of transaction $T$ with $TS(T) = j$
    - If $WS \cap \text{ReadSet}(T_i) \neq \{\}$, then abort $T_i$
  - If not aborted
    - $T_n = T_n + 1$
    - $TS(T_i) = T_n$
    - Write $\text{WriteSet}(T_i)$ to DB
Discussion

- Validation and Update Phase in Critical section
  - Reduced concurrency
  - Optimizations possible

- Space overhead:
  - To validate Tj, must have WriteSets for all Ti where Ti < Tj and Ti was active when Tj began.
  - Each Txn must record read/write activity

- No deadlock but potentially more aborts.
Forward Validation

- Keep track of all active transactions
- For simplicity: assume that at time of validation and update, other transactions may NOT read/write
- At validation of $T$
  
  For each active transaction $T'$, if
  
  \[ \text{WriteSet}(T) \cap \text{ReadSet}(T') \] is empty, then ok
  
  else validation does not succeed.

- Actions:
  
  Abort $T$ or abort conflicting $T'$. 
Application: Long lasting transactions

1. AS sends BOT to DBS
2. DBS sends BOT to AS
3. AS reads x, then writes x
4. DBS writes x, commits
5. AS writes x, commits
6. DBS checks version, detects collision
7. DBS sends BOT to AS
8. AS commits
9. DBS commits

COMP-577: Distributed Database Systems
Example

- Mark source with unique identifier
  - ✋ Extra field in relation
  - ✋ Updated at every write
- Compare all values of the object

http://www.agiledata.org/essays/concurrencyControl.html
Give each transaction a timestamp (TS) when it begins
- Time or some numeric (increasing) IDs
- If operation $O_{ij}$ of transaction $T_i$ conflicts with operation $O_{kl}$ of transaction $T_k$, and $TS(T_i) < TS(T_j)$, then $O_{ij}$ must occur before $O_{kl}$.
- I.e., we serialize all transactions in timestamp order, conflicting operations are always ordered in timestamp order.

Keep several versions of an object
- Each write operation creates a new version of an object. The version has the timestamp of the transaction as WRITE timestamp
  - Object $X$, $X(WS=TS(T_i))$ is the version created by transaction $T_j$
- There can exist several committed versions and several versions of active transactions.

Read operation of transaction $T_j$ reads version with highest timestamp smaller than $TS(T_i)$

Each object version has a READ timestamp. It is the highest timestamp of a transaction that read this object.
Operation check

- Transaction Ti submits read operation on object X:
  - Get version X(TS(T_j)) of X where TS(T_j) is the maximum timestamp smaller or equal than TS(T_i);
  - If T_j has already committed
    - read immediately
    - If read timestamp of version smaller than TS(T_i), set the read timestamp to TS(T_i)
  - If T_j is still active, wait until it commits/aborts then reapply the rule

- Transaction Ti submits write operation on object X:
  - May not be successful if there is a transaction T_j that should have read the version of X that Ti is going to write; i.e., T_j read X(TS(T_k)) but TS(T_k) < TS(T_i) < TS(T_j), i.e., T_j should have read Ti's version.
  - As any potentially conflicting read operation will have been directed to the most recent version of the object, Ti only has to check the version V of X with the highest WRITE timestamp TS_{maxearlier} < TS(T_i)
  - If read timestamp of V bigger than TS(T_i), abort Ti, otherwise create new object version with write timestamp TS(T_i)
Global timestamps

- **Centralized timestamp generation:**
  - Only one instance that provides timestamps
  - Local schedulers have to accept global timestamp

- **Distributed transaction managers**
  - Add process ID as second component:
    - $TS(T) = (\text{local-counter, process-id})$
    - Lexicographic comparison
    - Problematic if one site executes many more transactions than other sites
  - Use Lamport clock instead of local-counter to keep transaction-ids in synch.
  - Again: all schedulers have to accept global timestamp
What happens if the local schedulers do not accept global timestamp but generate local timestamps?

- $D_i = \{a\}, D_j = \{b\}$
- $T_1 = w_1(a), w_1(b), T_2 = w_2(b), w_2(a)$,
- GTM submits $w_1(a)$ to TM$i$ and $w_2(b)$ to TM$j$,
- At TM$i$: $TS(T_1) = 100$ and at TM$j$: $TS(T_2) = 200$,
- GTM submits $w_1(b)$ to TM$j$ and $w_2(a)$ to TM$i$,
- At TM$i$: $TS(T_2) = 101$ and at TM$j$: $TS(T_1) = 201$,
- Successful execution at both sites:
  - $Si$: $w_1^{100}(a), w_2^{101}(a)$
  - $Sj$: $w_2^{200}(b), w_1^{201}(b)$
- Locally serializable histories, global history non-serializable
Ticketing

❖ At each site $S_j$, we introduce a new data item that serves as a ticket: “$t-j$”. Each global transaction reads the current value of “$t-j$” and increases it by one before performing any other operation on $S_j$.

❖ With this, global transactions $T_1$ and $T_2$ artificially conflict at all nodes that they both access.

❖ At the GTM we maintain a ticket graph whose edges correspond to global active transactions.

- There is an edge from $T_1$ to $T_2$, $T_1 \rightarrow T_2$ means that at least at one node $T_1$ has read a ticket value smaller than the one read by $T_2$.
- At commit, we check for cycles. If $T_1$ is not involved in any cycle, we can commit it.

❖ $D_i = \{a\}, \quad D_j = \{b\}$

❖ $T_1 = w_1(a), \quad w_1(b), \quad T_2 = w_2(b), \quad w_2(a)$

❖ $TM_i: r_1^{100}(t-i) \quad w_1^{100}(t-i++) \quad w_1^{100}(a) \quad r_2^{101}(t-i) \quad w_2^{101}(t-i++) \quad w_2^{101}(b)$

❖ $TM_j: r_2^{200}(t-j) \quad w_2^{200}(t-j++) \quad w_2^{200}(b) \quad r_1^{201}(t-j) \quad w_1^{201}(t-j++) \quad w_1^{201}(b)$