Chapter 10: Transactions in Federated Databases
Topic of this Chapter

- Serializability in distributed systems (i.e., isolation).
- Several transactions.
- One solution has already been proposed (see next slide).
- In this chapter, we look at one specific case (federated databases) where that solution is insufficient.
Distributed Locking

- Data is partitioned among nodes.
- Each node synchronizes operations that access data on its partition.
- No further communication (except for replicas): distributed execution of transactions and operations already adapted to distribution of data.
- Release of locks with strict 2PL as part of ACP.
- Biggest problem: global deadlocks.
Global Transaction Management

- Works, but may be too optimistic.
- Global TM may synchronize global transactions.
Transaction Management in Federated DBMSs (1)

Introduction

Strictness

COCSR Tickets

Global Transactions

Global TM

Local TMs

Local Transactions
Transaction Management in Federated DBMSs (2)

- Common assumption: all transactions under control of a global distributed transaction manager (TM).
- Now: local transactions unknown to the GTM in addition.
How to achieve global serializability?

Is the following sufficient?

- Same concurrency-control protocol, and
- each server guarantees local serializability?
Example (1)

- \( D_1 = \{a\}; \ D_2 = \{b, c\} \),
- global transactions \( T_1 = r(a) \ w(b); \ T_2 = r(c) \ w(a) \),
- local transactions \( T_3 = r(b) \ w(c) \).
- Why are \( T_1 \) and \( T_2 \) global, and \( T_3 \) is local?
- Global TM is 'conservative'; serial execution \( (T_1; T_2) \).
- Possible local histories:
  - server 1: \( s_1 = r_1(a) \w_2(a) \)
  - server 2: \( s_2 = r_3(b) \w_1(b) \ r_2(c) \w_3(c) \)
- What are the local serialization orders?
Example (2)

- Corresponding global history:
  \[ s = r_1(a) \ r_3(b) \ w_1(b) \ r_2(c) \ w_2(a) \ w_3(c) \]
- \( s_1, \ s_2 \) conflict-serializable: \( s_1 \approx_{c} T_1 T_2; \ s_2 \approx_{c} T_2 T_3 T_4 \)
Direct vs. Indirect Conflict

- Important distinction: *direct conflict* vs. *indirect conflict*.
Global Serializability – Overview

- Global serializability based on local guarantees.
  1. Strictness, and commit is delayed,
  2. commitment ordering.
- Tickets
  (do not require additional local guarantees; ticket $\equiv$ additional local database object that global transactions must manipulate)
Strictness (1)

- **strictness** :=
  write(x, val) is delayed until all transactions that have written x before have committed or aborted, + cascadelessness.

- **Cascadelessness** :=
  Each transaction only reads values of committed transactions.
Strictness (2)

- $T_1 = w_1[x] w_1[y] w_1[z] c_1$
- $T_2 = r_2[u] w_2[x] r_2[y] w_2[y] c_2$

- $H_9 = w_1[x] w_1[y] r_2[u] w_2[x] w_1[z] c_1 r_2[y] w_2[y] c_2$
- $H_{10} = w_1[x] w_1[y] r_2[u] w_1[z] c_1 w_2[x] r_2[y] w_2[y] c_2$

- Which histories are strict?
- Which ones are cascadeless?
Strictness (3)

- Strictness alone does not suffice to ensure global serializability.
Strictness (4)

Strictness alone does not suffice to ensure global serializability – example:

- $D_1 = \{a,b\}; D_2 = \{c,d\}$,
- global transactions: $T_1 = w(a) \ w(d), \ T_2 = w(c) \ w(b)$,
- local transactions: $T_3 = r(a) \ r(b), \ T_4 = r(c) \ r(d)$,
- local histories:
  - $s_1 = w_1(a) \ c_1 \ r_3(a) \ r_3(b) \ c_3 \ w_2(b) \ c_2$
  - $s_2 = w_2(c) \ c_2 \ r_4(c) \ r_4(d) \ c_4 \ w_1(d) \ c_1$
- $s_1$ and $s_2$ are strict, but serialization orders differ.
- Observe that order of commits of global transactions differs between nodes.
Strictness (5)

- Do not issue commit right away.
- *Global transaction is commit-deferred* := global TM sends commit to local sites only after the local sites have acknowledged execution of all operations.
Strictness (6)

- Continuation of example:
  - $D_1 = \{a, b\}; D_2 = \{c, d\}$
  - global transactions: $T_1 = w(a) w(d), T_2 = w(c) w(b)$
  - local transactions: $T_3 = r(a) r(b), T_4 = r(c) r(d)$
  - local histories:
    - $s_1 = w_1(a) c_1 r_3(a) r_3(b) c_3 w_2(b) c_2$
    - $s_2 = w_2(c) c_2 r_4(c) r_4(d) c_4 w_1(d) c_1$
  - Not allowed. Deadlock situation.
Strictness (7)

- Continuation of example:
  - $D_1 = \{a,b\}; D_2 = \{c,d\}$
  - global transactions: $T_1 = w(a) w(d)$, $T_2 = w(c) w(b)$
  - local transactions: $T_3 = r(a) r(b)$, $T_4 = r(c) r(d)$
  - local histories:
    - $s_1 = w_1(a) c_1 r_3(a) r_3(b) c_3 w_2(b) c_2$
    - $s_2 = w_2(c) c_2 r_4(c) r_4(d) c_4 w_1(d) c_1$
  - $c_2$ takes place strictly after $c_1$ in $s_1$:
    - $r_3(a)$ after $c_1$ because of strictness.
    - $w_2(b)$ after $r_3(b)$
      according to serialization order.
    - $c_2$ after $w_2(b)$
      because of commit-deferredness.
  - Analogously, $c_1$ takes place strictly after $c_2$. 
Strictness (8)

- Theorem: let $s$ be a global history for $s_1, ..., s_n$. $s_i$ is strict for $1 \leq i \leq n$, and all global transactions are commit-deferred. $\Rightarrow s$ is globally serializable.
Strictness (9)

- Why is commit-deferredness, together with local serializability, not sufficient? Why do we need strictness as well?
Strictness (10)

- Running example:
  - $D_1 = \{a, b\}$; $D_2 = \{c, d\}$
  - global transactions: $T_1 = w(a) \ w(d)$, $T_2 = w(c) \ w(b)$
  - local transactions: $T_3 = r(a) \ r(b)$, $T_4 = r(c) \ r(d)$
  - Global TM issues $w_1(a)$ and $w_1(d)$.
    Then $w_2(c)$ and $w_2(b)$.
  - No guarantee that operations take place in that order. (At least when global TM does without handshaking.)
  - Thus, local histories so far could be:
    - $s_1 = w_1(a) \ r_3(a) \ r_3(b) \ c_3 \ w_2(b)$
    - $s_2 = w_2(c) \ r_4(c) \ r_4(d) \ c_4 \ w_1(d)$
  - Global TM now issues $c_1$, on both nodes.
Commitment Ordering (1)

- Global serializability based on local guarantees.
- Approach 2: commitment ordering (COCSR).
- \textit{Commit order conflict serializability} :=
  Two transactions conflict.
  \Rightarrow Commit operations in conflict order.
Commitment Ordering (2)

A DBMS that implements commitment ordering works as follows – example:

- applications issue $w_1(x)$, $w_2(x)$, $c_2$, strictly after each other (handshaking).
- Now $c_1$ is submitted to the DBMS – DBMS must reject it, must abort $T_1$. Execution order $w_1(x)$ $w_2(x)$ $c_2$ $c_1$ does not have commit operations in conflict order.
Commitment Ordering (3)

- COCSR, and commits of global transactions are executed one after the other ("handshaking").
  ⇒ History is globally serializable.
Commitment Ordering (4)

- Example – two histories:
  - $s_1 = r_1(a) c_1 w_3(a) w_3(b) c_3 r_2(b) c_2$
  - $s_2 = w_4(c) r_1(c) r_2(d) r_4(e) c_1 c_2$

Observe that $s_2$ is not COCSR.
Commitment Ordering (5)

- Example – two histories:
  - $s_1 = r_1(a) \ c_1 \ w_3(a) \ w_3(b) \ c_3 \ r_2(b) \ c_2$
  - $s_2 = w_4(c) \ r_1(c) \ r_2(d) \ r_4(e) \ c_1 \ c_2$

Observe that $s_2$ is not COCSR.

- Why is COCSR alone not sufficient?
  - Suppose $c_1$ and $c_2$ are issued in parallel; $c_1$ is delayed.
  - This may result in commit order $c_2 \ c_4 \ c_1$.
  - I.e., COCSR on both sites, but no local serializability.
Tickets (1)

- In what follows:
  nodes do not offer additional properties such as strictness or COCSR, only CSR locally.

- Example again:
  - \( D_1 = \{a, b\}; \ D_2 = \{c, d\} \),
  - global transactions \( T_1 = r(a) \ r(c); \ T_2 = r(b) \ r(d) \),
  - local transactions \( T_3 = w(a) \ w(b); \ T_4 = w(c) \ w(d) \)
  - The following local histories are not globally correct:
    - \( s_1 = r_1(a) \ c_1 \ w_3(a) \ w_3(b) \ c_3 \ r_2(b) \ c_2 \)
    - \( s_2 = w_4(c) \ r_1(c) \ c_1 \ r_2(d) \ c_2 \ w_4(d) \ c_4 \)

- Objective must be:
  execute \( T_2 \) only if it cannot be part of a cycle.
Tickets (2)

- Convert indirect conflict into direct one that global TM can recognize.
- Additional data object in each database („ticket“).
- Comparable with logical timestamp.
- Each global transaction
  - reads ticket,
  - writes back incremented value.
  
  „Take-a-ticket“ operation.
Tickets – Example

- \( D_1=\{a\}, \ D_2=\{b, \ c\}, \)
- global transactions \( T_1=r(a) \ w(b); \ T_2=w(a) \ r(c) \)
- local transaction \( T_3=r(b) \ w(c) \)
- possible local histories (without tickets):
  - \( s_1 = r_1(a) \ c_1 \ w_2(a) \ c_2 \)
  - \( s_2 = r_3(b) \ w_1(b) \ c_1 \ r_2(c) \ c_2 \ w_3(c) \ c_3 \)
- use of tickets:
  - \( s_1 = r_1(I_1) \ w_1(I_1+1) \ r_1(a) \ c_1 \ r_2(I_1) \ w_2(I_1+1) \ w_2(a) \ c_2 \)
  - \( s_2 = r_3(b) \ r_1(I_2) \ w_1(I_2+1) \ w_1(b) \ c_1 \)
    \( r_2(I_2) \ w_2(I_2+1) \ r_2(c) \ c_2 \ w_3(c) \ c_3 \)
- \( s_2 \) not conflict-serializable any more.
- The following history would be OK:
  - \( s_2 = r_1(I_2) \ w_1(I_2+1) \ w_1(b) \ c_1 \)
    \( r_2(I_2) \ w_2(I_2+1) \ r_2(c) \ c_2 \ r_3(b) \ w_3(c) \ c_3 \)
Tickets – Comment

- If order of tickets is same on all nodes, different local serialization orders will incur a conflict on one node.
- Topic of next slides: How/when to enforce that ticket ordering is same on all nodes.
Optimistic vs. Conservative Ticket Approach

- Conservative ticket approach:
  global TM fixes order of transactions taking tickets.
- Also possible:
  optimistic approach to ensure the same serialization order on all components.
  - TM simply issues ticket operations.
    TM must know their order of execution.
  - Ticket order must be the same everywhere.
  - Ticket-order graph:
    - global TM administers it.
    - Edge $T_i \rightarrow T_j$: a subtransaction of $T_i$ has read ticket before $T_j$.
    - Cycle $\Rightarrow$ abort.
Discussion

- 'lightweight' approach.
- Except for the fact that nodes must 'tolerate' additional data objects, no further interference with node autonomy.
Potential Exam Questions

- Why are local transactions in distributed systems problematic from a concurrency-control point of view?
- Give an example of an indirect conflict.
- How can global serializability in federated DBMS be ensured?
- What are tickets? Why could they be preferred over alternative approaches?
Literature

- This material is taken from the chapter on federated DBMS in the Weikum/Vossen book.