5. Advanced transaction models

5. Advanced transaction models

5. Advanced transaction models

Use of Semantic Information

- So far: Read-Write model
  - Low-level operations
  - No particular optimization possible
- Now: Use of semantic information
- Higher operations
- Permutability relations

Allowed permutation of operations

1. Transactions T_1, T_2 read two equal or different data objects:

$$r_i(x); r_j(y) \equiv r_j(y); r_i(x) \quad \text{if} \quad i \neq j$$

2. Transaction T_1 reads value, transaction T_2 writes another value:

$$r_i(x); w_j(y) \equiv w_j(y); r_i(x) \quad \text{if} \quad i \neq j, x \neq y$$

3. Transactions T_1, T_2 write two different data objects:

$$w_i(x); w_j(y) \equiv w_j(y); w_i(x) \quad \text{if} \quad i \neq j, x \neq y$$

New definition of Serializability

A schedule s is *conflict serializable* if it can be transformed into a serial schedule using a a finite sequence of allowed permutations $p; q \rightarrow q; p$.

New definition of Serializability: Example I

$$s = r_1(x) r_2(x) w_1(x) w_2(x)$$

- Application of Rule(1):

$$s' = r_2(x) r_1(x) w_1(x) w_2(x)$$

- No further permutation rules (base rules) can be applied
  $\rightsquigarrow$ Schedule s' is not a serial schedule
  $\rightsquigarrow$ Schedule s is not serializable

New definition of Serializability: Example II

$$s = r_1(x) r_2(x) w_1(x) w_2(y)$$

- Application of rules (1) and (3):
$$s' = r_2(x) r_1(x) w_2(y) w_1(x)$$
- Application of rule (2):
$$s'' = r_2(x) w_2(y) r_1(x) w_1(x)$$
- Schedule s'' is a serial schedule
  $\rightsquigarrow$ Schedule s is serializable

Permutation Table

$x \neq y$	$r_1(x)$	$r_1(y)$	$w_1(x)$	$w_1(y)$
$r_2(x)$	$\surd$	$\surd$	–	$\surd$
$r_2(y)$	$\surd$	$\surd$	$\surd$	–
$w_2(x)$	–	$\surd$	–	$\surd$
$w_2(y)$	$\surd$	–	$\surd$	–

Simplified Permutation Table

	$r_1(x)$	$w_1(x)$
$r_2(x)$	$\surd$	–
$w_2(x)$	–	–

Case Study

- Bank with two kinds of accounts:
  - Accounts for regular debits and credits
  - An account that includes the sum of all deposits in some particular accounts
- With every update, the account that includes the sum of deposits has to be updated $\rightsquigarrow$
  - Almost every transaction needs to access the sum object (sum account) $\rightsquigarrow$ *hot spot object*

New Operations for Hot Spot Objects

- $incr(x, n)$ (Adding a value)
- $decr(x, n)$ (Subtracting a value)

Permutation Table

	$r_1(x)$	$w_1(x)$	$incr_1(x, \_)$	$decr_1(x, \_)$
$r_2(x)$	$\surd$	–	–	–
$w_2(x)$	–	–	–	–
$incr_2(x, \_)$	–	–	$\surd$	$\surd$
$decr_2(x, \_)$	–	–	$\surd$	$\surd$

Compatibility of *incr*- and *decr*-operations!

Compatibility of Operations

$$incr_1(x,5);\ decr_2(x,4) \equiv decr_2(x,4);\ incr_1(x,5)$$

$\Rightarrow$ permutable!

$\rightarrow$ Execution order of operations *incr* and *decr* is arbitrary

Compensating Actions

Operation	Compensation
incr(x,n)	decr(x,n)
decr(x,m)	incr(x,m)

Use of Compensating Actions

Abort of a transaction $\rightarrow$ instead of **`abort`**, perform compensating action:

- Instead of: $incr_i(x, n)a_i$
- Now: $incr_i(x, n)$ ... $decr_i(x, n)c_i$

Use of term rewriting semantics

- Term substitution in schedules defines serializability, since missing conflicts can be explained by the permutation of rewrites
- If term rewrites are enhanced by compensation rules, it simplifies rollbacks on hot spot objects

Problems with ACID-Transactions

Distributed transactions:

- *Atomicity:* global transactions include local (visible) sub-transactions
- *Isolation:* global serializability sequences can hardly be forced

Further problems with particular applications

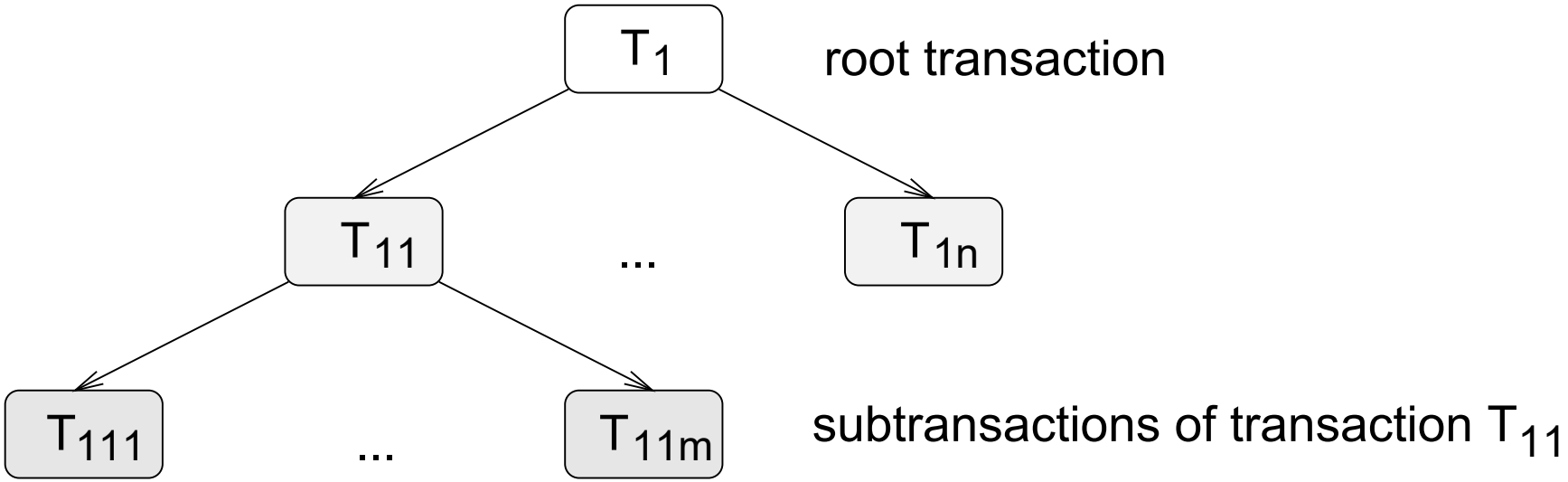
- *Transactions with side-effects* and atomicity
  - sending e-mails, manufacturing transactions (planing a workpiece)
- *Interactive Sessions* with isolation as well as parallelism
- *Design transactions in engineering* with atomicity as well as consistency
- *Cooperative work* with isolation
- *Transactions in real-time applications* with atomicity

Advanced Transaction Models

Principles based on two models:

- *Nested transactions*: Hierarchical accumulation of father/son transactions
- *Sagas*:
  Intermediate results are made available for other transactions by a commit. In case of a later abort, these results are undone.

Transaction Tree



ACID Properties of transaction trees

- *Isolation:*
  Results of a sub-transaction are being transferred to the father transaction, not visible for other concurrent transactions
- *Atomicity:*
  Either all transactions of the transaction tree are successful, or they all

Closed Nested Transactions I

- Transfer of locks of sub-transaction to its father transaction; locks are, therefore, passed on “upwards“ in the hierarchy (towards the root)
- Results of a closed nested transaction are only released when the root transaction has committed

Closed Nested Transactions II

- Atomicity:
  1. Abort of a father transaction forces abort of all sub-transactions
  2. A transaction of the transaction tree can only be successful if all sub-transactions are successful
  3. Abort of a sub-transaction leads to an abort of the father transaction

Open Nested Transactions (ONT)

- Results are already made available when the sub-transaction commits
- Atomicity of open nested transactions

  Given: Two transactions t_i and t_j, where t_j is vital son of t_i

  1. **abort**(t_i) forces an **abort**(t_j)
  2. **commit**(t_i) is only possible after **commit**(t_j)
- Classes of sub-transactions:
  Vital / non-vital transactions, contingency transactions

Reactions of **abort**(t_j) in ONT

1. **Ignore** for "not vital" sub-transactions
2. Restart of aborted sub-transaction: **retry** t_j, possibly depending on the cause of the abort
3. Attempt to perform (**try**) a *contingency transaction* (alternative execution of contingency transactions in case of abort or infeasibility of a transaction)
4. Abort of the father transaction: **abort**(t_i)

Sagas

Components of a Saga:

- A set of transactions T
- For each $T_i \in T$, a compensating transaction C_i that is able to semantically restore the state before a transaction

Saga $=$ particular ONT of depth 1

Valid execution histories

- Valid executions of a Saga:

$$T_1 T_2 \ldots T_n \qquad \text{(correct execution)}$$

or

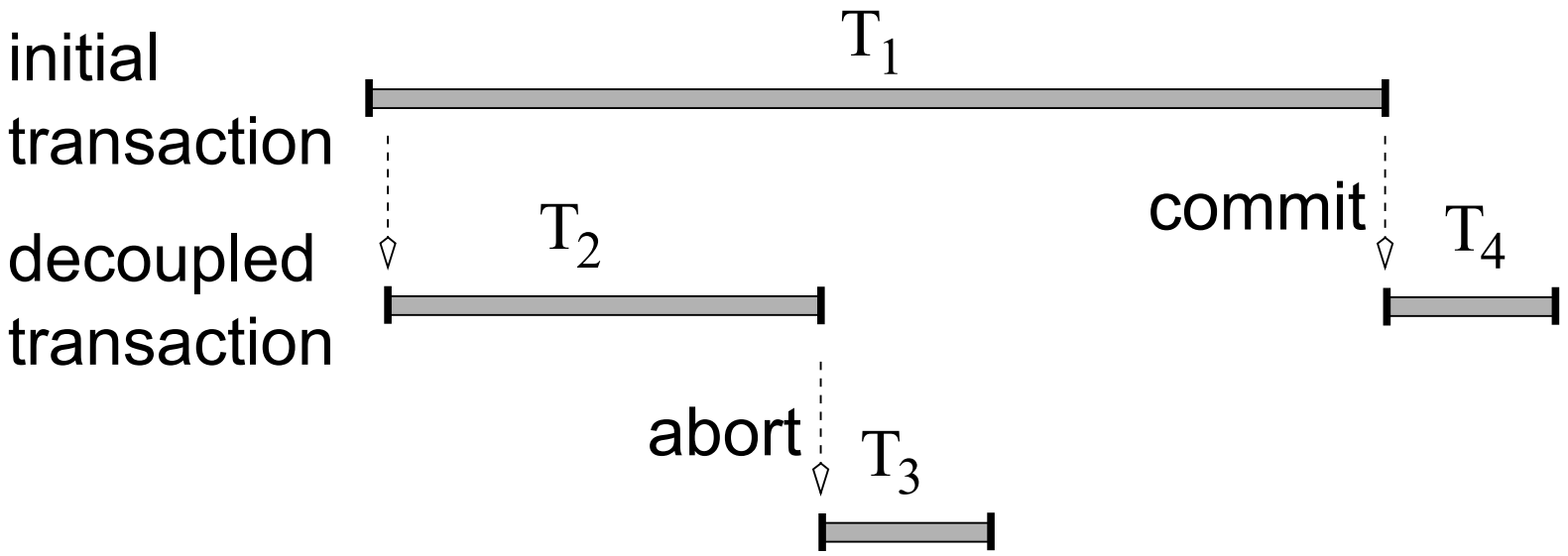
$$T_1 T_2 \ldots T_i C_i C_{i-1} \ldots C_2 C_1 \qquad \text{(controlled abort)}$$

- If sub-transaction $T_{i+1}, 1 \leq i < n$ is aborted, the second case applies.

Example for execution of a saga

1. Transaction T_1: Withdraw 10 Euros
2. Transaction T_2: Buy item
3. **`abort`** of T_2
4. Compensatory transaction C_1: Deposit 10 Euros

Decoupled sub-transactions I



Decoupled sub-transactions II

- Sub-transactions may occur temporally decoupled from the root transactions
- A sub-transaction may commit, even if the father transaction is aborted
- Apart from explicitly calling a sub-transaction, it can also be activated by a commit or abort of another transaction
  - Equivalent to special cases of control mode **`detached but causally dependent`** in active databases
  - Special dependencies are **`parallel`**, **`sequential`** (start of transaction after successful completion of triggering transaction) and **`exclusive`** (start only after abort of triggering transaction)