Semantics-based Concurrency Control and Transaction Processing

Krithi Ramamritham

Outline

- ▷ Motivation
- ▷ Our specific goals
- ▷ Preliminaries
- ▷ Semantics-based concurrency control
- ▷ Specifying, analyzing, and synthesizing, extended transaction models
- ▷ Correctness criteria beyond serializability
- ▷ Concluding Remarks

Motivation

▷ – *Atomic transactions*,

– *read/write data models*, and

– *serializability*

impact *functionality* and *efficiency* when used for

▷ *(database) applications*

– which are *distributed*,

– use *complex objects*

– have *high throughput demands*

– and are deployed in *environments* with

re-active (endless) activities

open-ended (long-lived) activities

collaborating (cooperative) activities

Examples: Heterogeneous and Multi-media Databases

Design Environments, Advanced OS and AI Systems

<u>**Transactions**</u>

Combine several important notions:

- *Consistency*: the correctness of the state of the database that a committed transaction produces.

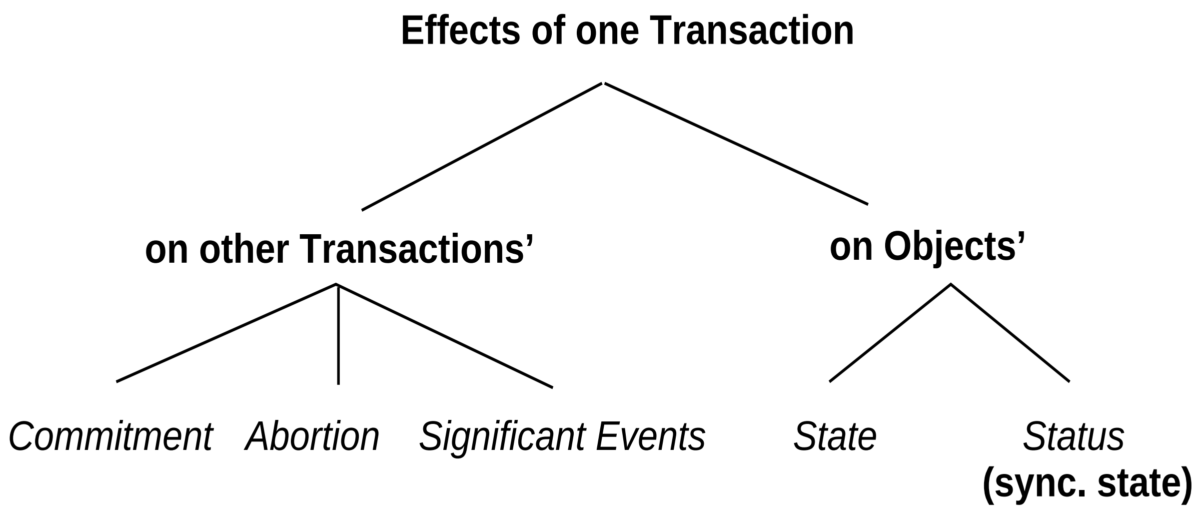
- *Visibility*: the ability of one transaction to see the results of another transaction *while* it is executing.

- *Recovery*: the ability, in the event of failure, to take the database to a state that is considered correct.

- *Permanence*: the ability of one transaction to effect its results in the database.

▷ The flexibility of a transaction model depends on the way these four notions are combined.

Preliminaries



History is a partial order of

- Significant (transaction management) Events
  - Commit, Abort, Begin, Spawn, Split, Joint, Delegate
- Object Events
  - operations on objects

Conflicts between Operations

$H^{(ob)}$: the projection of the history with respect to the operations on *ob*.

Operations p and q *conflict* in a state produced by $H^{(ob)}$, denoted by $\mathit{conflict}(\mathrm{H}^{(ob)}, p, q)$,

if the final state of an object is dependent of their execution order.

if their return values are dependent of their execution order.

$$
\begin{aligned}
(state(H^{(ob)} \circ p,\ q) \ &\neq\ state(H^{(ob)} \circ q,\ p)) \quad \vee \\
(return(H^{(ob)},\ q) \ &\neq\ (return(H^{(ob)} \circ p,\ q)) \vee \\
(return(H^{(ob)},\ p) \ &\neq\ (return(H^{(ob)} \circ q,\ p))
\end{aligned}
$$

Two operations that do not conflict are *compatible*.

Serializability

$\mathcal{C}$ – binary relation on transactions in T.

Let H be the history of events relating to committed transactions in T.

$\forall\ t_i, t_j \in T, t_i \neq t_j,$

$(t_i\ \mathcal{C}\ t_j)$ if

$\exists ob\ \exists p, q\ (\mathit{conflict}(\mathrm{p}_{t_i}[ob], q_{t_j}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob])).$

H is *(conflict) serializable* iff

$\forall t \in T, \neg(t\ \mathcal{C}^*\ t).$

Properties of Atomic Objects

An object ob behaves *correctly* iff

$\forall t_i, t_j, t_i \neq t_j \ \forall p, q$

$$(\textit{return-value-dependent}(p, q) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob])) \Rightarrow$$

$$((\mathsf{Abort}_{t_i}[p_{t_i}[ob]] \in H^{(ob)}) \Rightarrow (\mathsf{Abort}_{t_j}[q_{t_j}[ob]] \in H^{(ob)}))$$

An object ob behaves *serializably* iff

$$\forall t \ \forall p \ (Commit_t[p_t[ob]] \in H^{(ob)}) \Rightarrow \neg(t\mathcal{C}^* t)$$

An object ob is *atomic* if

ob behaves *correctly* and *serializably*.

Dependencies between Transactions

Interactions between a pair of transactions are expressed in terms of relations between *significant* events:

- $\triangleright$ *commit dependency* $(t_j \ \mathcal{CD}\ t_i)$

  if both t_j and t_i commit, t_i commits before t_j:

  $(\mathsf{Commit}_{t_j} \in H) \Rightarrow ((\mathsf{Commit}_{t_i} \in H) \Rightarrow (\mathsf{Commit}_{t_i} \rightarrow \mathsf{Commit}_{t_j}))$

  Example: $(read_{t_i}[x] \rightarrow write_{t_j}[x]) \Rightarrow (t_j \ \mathcal{CD}\ t_i)$

- $\triangleright$ *abort dependency* $(t_j \ \mathcal{AD}\ t_i)$

  if t_i aborts, then t_j aborts:

  $(\mathsf{Abort}_{t_i} \in H) \Rightarrow (\mathsf{Abort}_{t_j} \in H)$

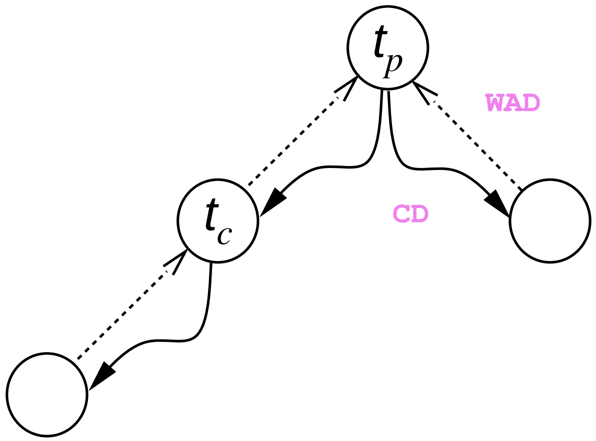
  Example: $(write_{t_i}[x] \rightarrow read_{t_j}[x]) \Rightarrow (t_j \ \mathcal{AD}\ t_i)$

▷ *weak-abort dependency* $(t_j \ \mathcal{WD} \ t_i)$

if t_i aborts and t_j not yet committed, t_j aborts:

$((\mathsf{Abort}_{t_i} \in H) \wedge \neg(\mathsf{Commit}_{t_j} \rightarrow \mathsf{Abort}_{t_i})) \Rightarrow (\mathsf{Abort}_{t_j} \in H).$

Example:



▷ Structure of Nested Transactions

$$
\begin{aligned}
post(Spawn_{t_p}[t_c]) \Rightarrow & \\
& (((t_c \ \mathcal{WD} \ t_p) \quad \in \quad DepSet_{ct}) \wedge \\
& ((t_p \ \mathcal{CD} \ t_c) \quad \in \quad DepSet_{ct}))
\end{aligned}
$$

Other Types of Dependencies

▷ *exclusion dependency* $(t_j \ \mathcal{ED} \ t_i)$

if transactions t_i commits and t_j aborts:

$(Commit_{t_i} \in H) \Rightarrow (Abort_{t_j} \in H)$.

▷ *compensation dependency* $(t_j \ \mathcal{CMD} \ t_i)$

if t_i aborts, t_j commits:

$(Abort_{t_i} \in H) \Rightarrow (Commit_{t_j} \in H)$.

▷ *begin dependency* $(t_j \ \mathcal{BD} \ t_i)$

t_j cannot begin executing until t_i has begun:

$(Begin_{t_j} \in H) \Rightarrow (Begin_{t_i} \to Begin_{t_j})$.

▷ *serial dependency* $(t_j \ \mathcal{SD} \ t_i)$

t_j cannot begin execution until t_i commits or aborts:

$(Begin_{t_j} \in H) \Rightarrow (\epsilon \to Begin_{t_j})$ where $\epsilon \in \{Commit_{t_i}, Abort_{t_i}\}$.

Sources of Dependencies

- $\triangleright$ Dependencies may be a result of

  – the structure (direct)

  e.g., explicit nesting structure of Nested Transactions

  – the behavior (indirect)

  e.g., operations on shared objects

  of transactions.

- $\triangleright$ By being able to express dependencies from the two sources, ACTA is able to capture various forms of interactions.

Concurrent Operations – Effects and Dependencies

Operation q makes p's effects/observations obsolete

- if q is a modifier and p is an observer or modifier.

To ensure serializability,

q can commit only after p commits or aborts, i.e.,
T_q is *commit-dependent* (CD) on T_p.

(Actually, (T_p $\mathcal{C}$ T_q) is sufficient.) Operation q observes the effects of p

- if q is an observer and p is a modifier.

To guarantee failure atomicity,

q has to abort if p aborts, i.e.,
T_q has an *abort-dependency* (AD) on T_p.

	O	M
O		AD
M	CD	CD

(A blank entry indicates a Null-dependency.)

AD is more restrictive (*stronger*) than CD, and
CD is more restrictive than ND.

What if p is *a modifier as well as an observer*?

Dependency (O,MO)

$$= \mathit{stronger}((\mathrm{O},\mathrm{M}),(\mathrm{O},\mathrm{O}))$$
$$= \mathit{stronger}(\mathrm{AD},\mathrm{ND})$$
$$= \mathrm{AD}.$$

	O	M	MO
O		AD	AD
M	CD	CD	CD
MO	CD	AD	AD

This is the basis for semantics-based concurrency control.

Extended Transaction Models

▷Transactions are ascribed properties that make them resemble anything between *threads* and *atomic transactions*

Some relevant questions

– What are the properties of given transaction model *vis a vis* visibility, consistency, recovery, and permanence?

– How does one transaction model differ from another?

– Can two models be used in conjunction?

– What are the effects of changing some of the properties of a given model?

– What are the mechanisms needed for managing extended transactions and achieving the desired correctness properties?

Visibility and Conflicts

Each transaction t is associated with:

- $View_t$: the subhistory visible to t at any given point in time.

  $$View_t = Projection(H_{ct}, \mathcal{P}redicate(t, H_{ct}, DepSet_{ct}))$$
  $$\text{E.g., } View_{t_c} = H_{ct}$$

  Thus, $View_t$ defines the *state* of objects visible to t.

- $ConflictSet_t$: the set of operations w.r.t. which conflicts need be considered.

  $$ConflictSet_t = \{p_{t_i}[ob] \mid \mathcal{P}redicate(t, t_i, H_{ct}, DepSet_{ct}))\}$$
  $$\text{E.g., } ConflictSet_{t_c} = \{p_{t_i}[ob] \mid t_i \neq t_c, t_i \notin Ancestor(t_c), \mathcal{I}nprogress(p_{t_i}[ob])\}$$

$AccessSet_t$: the set containing all the operations performed by t.

$$AccessSet_t = \{p_t[ob] \in H_{ct}\}$$

Delegation

$Delegate_{t_i}[t_j, p]$: t_i delegates to t_j the responsibility for committing or aborting p.

It is as if t_j performed p and *not* t_i.

Delegation affects the interpretation of history:

- redirects the dependencies induced by operations performed on the delegated objects from the delegator to the delegatee,

- does not induce dependencies between delegator and delegatee; hence, delegation broadens the visibility of the delegatee,

Delegation in Nested Transactions

▷ Inheritance is an instance of delegation

Delegation from t_c to t_p occurs when t_c commits:

$$(\mathsf{Commit}_{t_c} \in H) \Leftrightarrow (Delegate_{t_c}[t_p, AccessSet_{t_c}] \in H)$$

Axiomatic definition of Nested Transactions

t_0 denotes the root transaction.

t_p denotes a root or a subtransaction.

t_c denotes a subtransaction of t_p.

1. $SE_{t_0} = \{\textsf{Begin, Spawn, Commit, Abort}\}$
2. $IE_{t_0} = \{\textsf{Begin}\}$
3. $TE_{t_0} = \{\textsf{Commit, Abort}\}$
4. $SE_{t_c} = \{\textsf{Spawn, Commit, Abort}\}$
5. $IE_{t_c} = \{\textsf{Spawn}\}$
6. $TE_{t_c} = \{\textsf{Commit, Abort}\}$

t_0 denotes the root transaction. t_p denotes a root or a subtransaction.

t_c denotes a subtransaction of t_p.

7. $\exists ob\ \exists p\ (Commit_{t_0}[p_{t_0}[ob]] \in H) \Rightarrow (\mathsf{Commit}_{t_0} \in H)$

8. $(\mathsf{Commit}_{t_0} \in H) \Rightarrow \forall ob\ \forall p\ ((p_{t_0}[ob] \in H) \Rightarrow (Commit_{t_0}[p_{t_0}[ob]] \in H))$

9. $\exists ob\ \exists p\ (Abort_{t_p}[p_{t_p}[ob]] \in H) \Rightarrow (\mathsf{Abort}_{t_p} \in H)$

10. $(\mathsf{Abort}_{t_p} \in H) \Rightarrow \forall ob\ \forall p\ ((p_{t_p}[ob] \in H) \Rightarrow (Abort_{t_p}[p_{t_p}[ob]] \in H))$

11. $View_{t_0} = H_{ct}; View_{t_c} = H_{ct}$

12. $ConflictSet_{t_0} = \{p_t[ob] \mid t \neq t_0, \mathcal{I}nprogress(p_t[ob])\}$

13. $ConflictSet_{t_c} = \{p_t[ob] \mid t \neq t_c, t \notin Ancestors(t_c), \mathcal{I}nprogress(p_t[ob])\}$

14. $post(\mathsf{Spawn}_{t_p}[t_c]) \Rightarrow (((t_c\ \mathcal{WD}\ t_p) \in DepSet_{ct}) \wedge ((t_p\ \mathcal{CD}\ t_c) \in DepSet_{ct}))$

15. $(\mathsf{Commit}_{t_c} \in H) \Leftrightarrow (Delegate_{t_c}[t_p, AccessSet_{t_c}] \in H)$

16. $\forall t \in Descendants(t_p)\ \forall ob\ \forall q$

    $pre(q_{t_p}[ob]) \Rightarrow (\nexists p\ ((p_t[ob] \in H_{ct}) \wedge conflict(\mathrm{p,q}))$

Properties of Nested Transactions

Lemma: (*no orphan commits*)

Let H be a history of a nested transaction,

t_p and t_c be transactions,

t_p is the parent of t_c.

An *orphan*, i.e., an uncommitted child with a committed or aborted parent, aborts.

$$
\begin{aligned}
&(((\mathsf{Commit}_{t_p} \in H) \land \neg(\mathsf{Commit}_{t_c} \to \mathsf{Commit}_{t_p}))\\
&\quad \lor((\mathsf{Abort}_{t_p} \in H) \land \neg(\mathsf{Commit}_{t_c} \to \mathsf{Abort}_{t_p})))\\
&\qquad\qquad\qquad\qquad \Rightarrow (\mathsf{Abort}_{t_c} \in H)
\end{aligned}
$$

Let $H^{Committed}$ be the *committed projection* of H; i.e.,

$$H^{committed} = Projection(H, p_t[ob] \mid (Commit_t \in H))$$

<u>*Theorem:*</u>

Nested transactions have the following properties:

(1) if H a history of nested transactions, $H^{Committed}$ is *serializable*

(2) All operations in a committed history have been performed by root transactions:

$$\exists t\ \exists p\ (p_t[ob] \in H^{Committed}) \Rightarrow (t \text{ is a root transaction})$$

(3) If a root transaction t_0 aborts, all operations performed by t_0 and its descedents abort:

$$(\mathsf{Abort}_{t_0} \in H) \Rightarrow$$

$$\forall t, t = t_0 \vee t \in Descendants(t_0)\ \exists t_i, t_i = t_0 \vee t_i \in Descendants(t_0)\ \ \forall ob\ \forall p$$

$$((q_t[ob] \in H) \Rightarrow (Abort_{t_i}[p_{t \ldots t_i}[ob]] \in H))$$

Sagas' Correctness Properties

Theorem: The component transactions of a Saga produce one of the following committed histories:

1. All component transactions commit in the required order.

   $\forall i, 1 < i \leq n \; (\mathsf{Commit}_{T_{i-1}} \rightarrow \mathsf{Commit}_{T_i})$

2. For all committed components,

   their compensating transactions commit in the required order:

   $\exists k, 1 \leq k \leq n \; \forall i, 1 < i < k-1$

   $((\mathsf{Commit}_{T_{i-1}} \rightarrow \mathsf{Commit}_{T_i}) \wedge$

   $(\mathsf{Commit}_{T_{k-1}} \rightarrow \mathsf{Commit}_{CT_{k-1}}) \wedge$

   $(\mathsf{Commit}_{CT_i} \rightarrow \mathsf{Commit}_{CT_{i-1}}))$

Benefits of using the ACTA framework

A general and flexible framework for specifying the properties of transactions

- Consistency in terms of *allowable*, *required,* or *proscribed* relationships between significant events.
- Visibility in terms of *View* and *ConflictSet*.
- Recovery in terms of the effects of significant events and *delegation*.
- Permanence by separating operation commitment and transaction commitment.

- allows for analysis of correctness properties
- unifies existing transaction models
- facilitates comparisons
- supports integration
- supports the development of new transaction models
- ACTA forms the basis for a *theory* of extended transactions.

Cooperative Serializability

Extended transactions collaborate over some objects while maintaining the consistency of the objects.

The set of cooperative transactions becomes the unit of serializability.

Let T_c be a set of cooperative transactions, $T_c \subseteq T$.

Let $\mathcal{C}_c$ be a binary relation on transactions in T.

$$
\begin{array}{l}
\forall\ t_i,\ t_j,\ t_k\ \in\ T, t_i \neq t_j, t_i \neq t_k, t_j \neq t_k\ \forall T_C \subseteq T \\
\quad (t_i\ \mathcal{C}_c\ t_j), \text{ if} \\
\qquad \exists ob\ \exists p, q\ ((t_i \notin T_c, t_j \notin T_c\ (\mathit{conflict}(\mathrm{p}_{t_i}[ob], q_{t_j}[ob])\ \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]))) \vee \\
\qquad (t_i \notin T_c, t_j \in T_c, t_k \in T_c\ (\mathit{conflict}(\mathrm{p}_{t_i}[ob], q_{t_k}[ob])\ \wedge (p_{t_i}[ob] \rightarrow q_{t_k}[ob]))) \vee \\
\qquad (t_i \in T_c, t_j \notin T_c, t_k \in T_c\ (\mathit{conflict}(\mathrm{p}_{t_k}[ob], q_{t_j}[ob])\ \wedge (p_{t_k}[ob] \rightarrow q_{t_j}[ob]))))
\end{array}
$$

H is *cooperative serializable* iff

$$\forall t \in T\ \neg(t\ \mathcal{C}_c^*\ t)$$

Summary

- Semantics-based *concurrency control* protocols that use
  - Semantics of the operations, and
  - Structure of the database objects.

  ▷ Have developed several protocols plus a method to extract concurrency properties of objects.

- (Extended) transaction semantics.

  ▷ Have developed a formal transaction framework, called ACTA, for specifying, analyzing, and synthesizing extended transaction models.

- Correctness criteria that extend serializability.

  ▷ Have specified various correctness criteria using ACTA and have developed a taxonomy of these criteria.

Our work has implications for real-time databases as well.