Analyzing Snapshot Isolation

Andrea Cerone   Alexey Gotsman

IMDEA Software Institute

PODC 2016

presented by Dima Kuznetsov

Agenda

- We focus on *Snapshot Isolation* presented in the last talk
- ... same context of DBMS and transactional memory systems

- DBMS typically offer various guarantees for transaction management
- Each mode exhibits different *anomalies*
- Stronger modes exhibit less anomalies at expense of performance

    - Stronger guarantees incur more overhead on the DBMS side
    - Less allowed behaviors $\rightarrow$ more concurrent transactions expected to abort

From Wikipedia[1]:

*A transaction executing under **snapshot isolation** appears to operate on a personal snapshot of the database, taken at the start of the transaction. When the transaction concludes, it will successfully commit only if the values updated by the transaction have not been changed externally since the snapshot was taken.*

We'll focus on **strong session** *Snapshot Isolation*:
Transactions are grouped into sessions, a transaction's snapshot is expected to include *all preceding transactions* of the same session.

Outline

We'll use a lot of notation similar to Daniel's talk two weeks ago

- $Obj = \{x, y, \dots\}$ - objects in the data set
- $Event = \{e, f, \dots\}$ - transaction events
- $Op = \{read(x, n), write(x, n) \mid x \in Obj, n \in \mathbb{Z}\}$
- $op : Event \to Op$

Definition

A **transaction** $T, S, \ldots$ is a pair (E, po) where $E \subseteq Event$ is a *finite*, non-empty set of events and **program-order** $po \subseteq E \times E$ is a total order.

Where ...

- *total order* is a transitive and irreflexive relation that orders all pairs

Definition

A **history** is a pair $\mathcal{H} = (\mathcal{T}, SO)$ where $\mathcal{T}$ is a finite set of transactions with disjoint set of events and the **session-order** $SO \subseteq \mathcal{T} \times \mathcal{T}$ is a union of total orders defined on disjoint subsets of $\mathcal{T}$, which correspond to transactions in different sessions.

Assumptions:

- All transactions commit (aborted ones do not affect the history)
- All histories are finite (no infinite computations)

Definition

An **abstract execution** is a tuple $\mathcal{X} = (\mathcal{T}, SO, VIS, CO)$, where $(\mathcal{T}, SO)$ is a history and the **visibility** and **commit order** $VIS, CO \subseteq \mathcal{T} \times \mathcal{T}$ are such that $VIS \subseteq CO$ and CO is total.

- We'll use $(T, S) \in \textit{VIS}$ and $T \xrightarrow{\textit{VIS}} S$ interchangeably for $\textit{VIS}$ and other relations.
- For $\mathcal{H} = (\mathcal{T}, \textit{SO})$ we will shorten $(\mathcal{T}, \textit{SO}, \textit{VIS}, \textit{CO})$ to $(\mathcal{H}, \textit{VIS}, \textit{CO})$
- For $\mathcal{H} = (\mathcal{T}, \textit{SO})$ and other tuples, we'll use $\mathcal{T}_{\mathcal{H}}$ to denote that $\mathcal{T}$ is part of the tuple $\mathcal{H}$

For the relations defined in abstract execution:

- $T \xrightarrow{VIS} S$ means that T is included in S's snapshot.
- $T \xrightarrow{CO} S$ means that T is committed before S.
- $VIS \subseteq CO$ makes sure that snapshots include only already committed transactions.

We'll now try to define *snapshot isolation* and *serializability* in terms of **consistency axioms**:

$$ExecSI = \begin{aligned}\{\mathcal{X} \mid \mathcal{X} \vDash &\text{INT} \land \text{EXT} \land \text{SESSION} \land \\ &\text{PREFIX} \land \text{NOCONFLICT}\}\end{aligned}$$

$$ExecSER = \{\mathcal{X} \mid \mathcal{X} \vDash \text{INT} \land \text{EXT} \land \text{SESSION} \land \text{TOTALVIS}\}$$

$$HistSI = \{\mathcal{H} \mid \exists VIS, CO : (\mathcal{H}, VIS, CO) \in ExecSI\}$$

$$HistSER = \{\mathcal{H} \mid \exists VIS, CO : (\mathcal{H}, VIS, CO) \in ExecSER\}$$

Definition (Internal consistency)

INT - ensures that a read event e on object x returns the same value a as the last write or read on x in the same transaction.

$$\forall (E, po) \in \mathcal{T}.\forall e \in E.\forall x, n :$$
$$op(e) = read(x, n) \wedge \{f \mid op(f) = \_(x, \_) \wedge f \xrightarrow{po} e\} \neq \emptyset \Rightarrow$$
$$op\left(max_{po}\{f \mid op(f) = \_(x, \_) \wedge f \xrightarrow{po} e\}\right) = \_(x, n)$$

Where ...

- $max_R(A) = \{a \mid \forall b \in A : a = b \vee (b, a) \in R\}$
- $min_R(A) = \{a \mid \forall b \in A : a = b \vee (a, b) \in R\}$

Definition (External consistency)

Ext - ensures that if $T \vdash read(x, n)$ then the value is taken from the last visible transaction that wrote to x according to commit order.

$$\forall T \in \mathcal{T}. \forall x, n :$$
$$T \vdash read(x, n) \Rightarrow max_{CO}\left(VIS^{-1}(T) \cap WriteTx_x\right) \vdash write(x, n)$$

Where ...
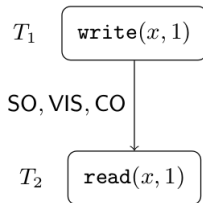
- $R^{-1}(a) = \{b \mid (b, a) \in R\}$
- $WriteTx_x = \{T \mid T \vdash write(x, \_)\}$
- $T \vdash write(x, n)$ if T writes to x and n is the final value of x.
- $T \vdash read(x, n)$ if T reads from x and n is the value of x at the first read.

Definition (Session visibility)

SESSION - requires a snapshot to include all preceding transactions of the same session.

$$SO \subseteq VIS$$

(a) Session guarantees.

$$T_1 \quad \boxed{\texttt{write}(x, 1)}$$

$$SO, VIS, CO$$

$$T_2 \quad \boxed{\texttt{read}(x, 1)}$$

Example

T_1 ordered after T_2 by *SO* (therefore by *VIS* and *CO*), T_2 must read 1 from x.

Definition (Prefix)

PREFIX - ensures that if snapshot taken by T includes S, then it includes all transactions committed before S as well.
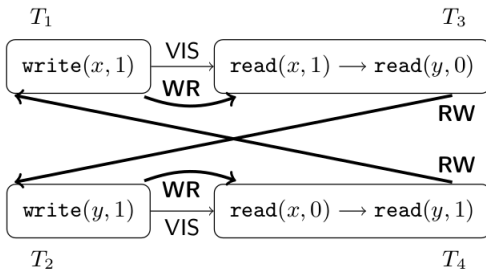
$$CO; VIS \subseteq VIS$$

Where ...

- $R_1; R_2 = \{(a, b) \mid \exists c : (a, c) \in R_1 \land (c, b) \in R_2\}$

The **long-fork** anomaly is prevented by PREFIX axiom:

(c) Long fork.

Example

Consider T_1 commits before T_2, then since T_4's snapshot contains T_2 (due to *VIS*), it must include T_1 as well
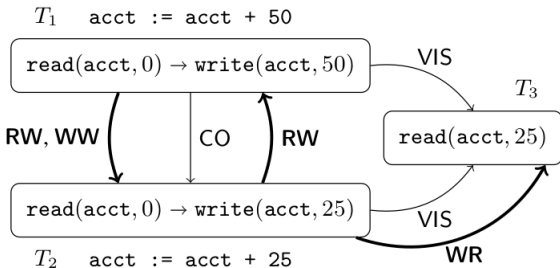
Definition (No conflict check)

NOCONFLICT - ensures that for any two transactions writing to the same object, one has to be aware of the other.

$$\forall T, S \in \mathcal{T}. \forall x, n.$$
$$(T, S \in \textit{WriteTx}_x \land T \neq S) \Rightarrow \left(T \xrightarrow{\textit{VIS}} S \lor S \xrightarrow{\textit{VIS}} T \right)$$

The **lost-update** anomaly is prevented by NOCONFLICT axiom:

(b) Lost update.

Example

T_1 and T_2 concurrently increment *acct* object, but neither $T_1 \xrightarrow{VIS} T_2$ nor $T_2 \xrightarrow{VIS} T_1$.

Definition (Total visibility)

TOTALVIS - requires total order on the visibility relation, giving us *serializability* of transactions.

$$VIS = CO$$

The **write-skew** anomaly allowed by Snapshot Isolation is prevented by TOTALVIS axiom:

(d) Write skew. Initially $\mathtt{acct1} = \mathtt{acct2} = 60$.

T_1

```
if (acct1 + acct2 > 100)
    acct1 := acct1 - 100
```

$$\boxed{\mathtt{read(acct1, 60)} \rightarrow \mathtt{read(acct2, 60)} \rightarrow \mathtt{write(acct1, -40)}}$$

RW ↓   ↑ RW

T_2

```
if (acct1 + acct2 > 100)
    acct2 := acct2 - 100
```

$$\boxed{\mathtt{read(acct1, 60)} \rightarrow \mathtt{read(acct2, 60)} \rightarrow \mathtt{write(acct2, -40)}}$$

Example

With TOTALVIS, either T_1 or T_2 would have to be aware of the other, and we won't be able to read stale values.

Outline

- Our goal now is to characterize SI in terms of dependencies between transactions.
- Then we'll be able to decide whether SI allows a given history by looking for appropriate dependencies.

Definition

Consider execution $\mathcal{X} = (\mathcal{T}, SO, VIS, CO)$, for $x \in Obj$ we define:

- **read-dependency** $WR_{\mathcal{X}}(x)$ as:

$$T \xrightarrow{WR_{\mathcal{X}}(x)} S \Leftrightarrow$$
$$S \vdash read(x, n) \wedge T = max_{CO}\left(VIS^{-1}(S) \cap WriteTx_x\right)$$

- **write-dependency** $WW_{\mathcal{X}}(x)$ as:

$$T \xrightarrow{WW_{\mathcal{X}}(x)} S \Leftrightarrow T \xrightarrow{CO} S \wedge T, S \in WriteTx_x$$

- **anti-dependency** $RW_{\mathcal{X}}(x)$ as:

$$T \xrightarrow{RW_{\mathcal{X}}(x)} S \Leftrightarrow T \neq S \wedge \exists T'. T' \xrightarrow{WR_{\mathcal{X}}(x)} T \wedge T' \xrightarrow{WW_{\mathcal{X}}(x)} S$$

Definition

Consider execution $\mathcal{X} = (\mathcal{T}, SO, VIS, CO)$, for $x \in Obj$ we define:

- **read-dependency** $WR_{\mathcal{X}}(x)$ as:

$$T \xrightarrow{WR_{\mathcal{X}}(x)} S \Leftrightarrow$$
$$S \vdash read(x, n) \wedge T = max_{CO}\left(VIS^{-1}(S) \cap WriteTx_x\right)$$

- **write-dependency** $WW_{\mathcal{X}}(x)$ as:

$$T \xrightarrow{WW_{\mathcal{X}}(x)} S \Leftrightarrow T \xrightarrow{CO} S \wedge T, S \in WriteTx_x$$

- **anti-dependency** $RW_{\mathcal{X}}(x)$ as:

$$T \xrightarrow{RW_{\mathcal{X}}(x)} S \Leftrightarrow T \neq S \wedge \exists T'. T' \xrightarrow{WR_{\mathcal{X}}(x)} T \wedge T' \xrightarrow{WW_{\mathcal{X}}(x)} S$$

Definition

Consider execution $\mathcal{X} = (\mathcal{T}, SO, VIS, CO)$, for $x \in Obj$ we define:

- **read-dependency** $WR_{\mathcal{X}}(x)$ as:

$$T \xrightarrow{WR_{\mathcal{X}}(x)} S \Leftrightarrow$$
$$S \vdash read(x, n) \wedge T = max_{CO}\left(VIS^{-1}(S) \cap WriteTx_x\right)$$

- **write-dependency** $WW_{\mathcal{X}}(x)$ as:

$$T \xrightarrow{WW_{\mathcal{X}}(x)} S \Leftrightarrow T \xrightarrow{CO} S \wedge T, S \in WriteTx_x$$
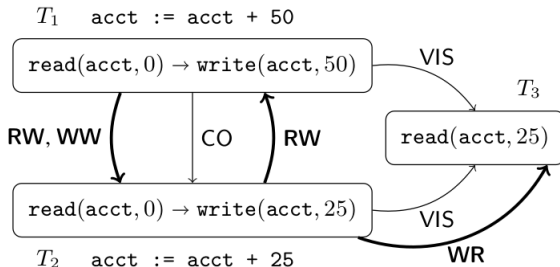
- **anti-dependency** $RW_{\mathcal{X}}(x)$ as:

$$T \xrightarrow{RW_{\mathcal{X}}(x)} S \Leftrightarrow T \neq S \wedge \exists T'. T' \xrightarrow{WR_{\mathcal{X}}(x)} T \wedge T' \xrightarrow{WW_{\mathcal{X}}(x)} S$$

Informally,

- $T \xrightarrow{WR_x(x)} S$ means that S reads T's write to x.
- $T \xrightarrow{WW_x(x)} S$ means that S overwrites T's write to x.
- $T \xrightarrow{RW_x(x)} S$ means that S overwrites the write to x read by T.

(b) Lost update.

Example

- T_3 reads *acct* from T_2's write $\Rightarrow T_2 \xrightarrow{WR(acct)} T_3$
- T_2 overwrites *acct* written in $T_1 \Rightarrow T_1 \xrightarrow{WW(acct)} T_2$
- Both T_1 and T_2 overwrite *acct*'s initial value read by both, $T_1 \xrightarrow{RW(acct)} T_2$ and $T_2 \xrightarrow{RW(acct)} T_1$.

Definition

A **dependency graph** is a tuple $\mathcal{G} = (\mathcal{T}, SO, WR, WW, RW)$, where $(\mathcal{T}, SO)$ is a history and:

- WR: $Obj \to 2^{\mathcal{T} \times \mathcal{T}}$ is such that:
  - $\forall T, S. \forall x. T \xrightarrow{WR(x)} S \Rightarrow \exists n. T \neq S \wedge T \vdash write(x, n) \wedge S \vdash read(x, n)$
  - $\forall S \in \mathcal{T}. \forall x. S \vdash read(x, \_) \Rightarrow \exists T. T \xrightarrow{WR(x)} S$
  - $\forall T, T', S \in \mathcal{T}. \forall x. \left(T \xrightarrow{WR(x)} S \wedge T' \xrightarrow{WR(x)} S \right) \Rightarrow T = T'$

- WW: $Obj \to 2^{\mathcal{T} \times \mathcal{T}}$ is such that for every $x \in Obj$, $WW(x)$ is a total order on the set $WriteTx_x$.

- RW: $Obj \to 2^{\mathcal{T} \times \mathcal{T}}$ is derived from WR and WW as in the definition of $RW_x(x)$.

Definition

A **dependency graph** is a tuple $\mathcal{G} = (\mathcal{T}, SO, WR, WW, RW)$, where $(\mathcal{T}, SO)$ is a history and:

- WR: $Obj \rightarrow 2^{\mathcal{T} \times \mathcal{T}}$ is such that:
    - $\forall T, S. \forall x. T \xrightarrow{WR(x)} S \Rightarrow \exists n. T \neq S \wedge T \vdash write(x, n) \wedge S \vdash read(x, n)$
    - $\forall S \in \mathcal{T}. \forall x. S \vdash read(x, \_) \Rightarrow \exists T. T \xrightarrow{WR(x)} S$
    - $\forall T, T', S \in \mathcal{T}. \forall x. \left(T \xrightarrow{WR(x)} S \wedge T' \xrightarrow{WR(x)} S \right) \Rightarrow T = T'$

- WW: $Obj \rightarrow 2^{\mathcal{T} \times \mathcal{T}}$ is such that for every $x \in Obj$, $WW(x)$ is a total order on the set $WriteTx_x$.

- RW: $Obj \rightarrow 2^{\mathcal{T} \times \mathcal{T}}$ is derived from WR and WW as in the definition of $RW_x(x)$.

Definition

A **dependency graph** is a tuple $\mathcal{G} = (\mathcal{T}, SO, WR, WW, RW)$, where $(\mathcal{T}, SO)$ is a history and:

- WR: $Obj \rightarrow 2^{\mathcal{T} \times \mathcal{T}}$ is such that:
  - $\forall T, S.\forall x. T \xrightarrow{WR(x)} S \Rightarrow \exists n. T \neq S \wedge T \vdash write(x, n) \wedge S \vdash read(x, n)$
  - $\forall S \in \mathcal{T}.\forall x. S \vdash read(x, \_) \Rightarrow \exists T. T \xrightarrow{WR(x)} S$
  - $\forall T, T', S \in \mathcal{T}.\forall x. \left(T \xrightarrow{WR(x)} S \wedge T' \xrightarrow{WR(x)} S \right) \Rightarrow T = T'$

- WW: $Obj \rightarrow 2^{\mathcal{T} \times \mathcal{T}}$ is such that for every $x \in Obj$, $WW(x)$ is a total order on the set $WriteTx_x$.

- RW: $Obj \rightarrow 2^{\mathcal{T} \times \mathcal{T}}$ is derived from WR and WW as in the definition of $RW_{\mathcal{X}}(x)$.

Proposition

For any $\mathcal{X} \in ExecSI$,

$$graph(\mathcal{X}) = (\mathcal{T}_{\mathcal{X}}, SO_{\mathcal{X}}, WR_{\mathcal{X}}, WW_{\mathcal{X}}, RW_{\mathcal{X}})$$

is a dependency graph.

Proof

By showing *graph*($\mathcal{X}$) satisfies all requirements of a dependency graph.

Outline

We'll show that SI is characterized by dependency graphs that contain only cycles with at least two adjacent anti-dependency edges.

Theorem

Let

$$GraphSER = \{\mathcal{G} \mid (\mathcal{T}_\mathcal{G} \vDash \text{INT})$$
$$((SO_\mathcal{G} \cup WR_\mathcal{G} \cup WW_\mathcal{G} \cup RW_\mathcal{G}) \text{ is acyclic})\}$$

Then

$$HistSER = \{\mathcal{H} \mid \exists WR, WW, RW.$$
$$(\mathcal{H}, WR, WW, RW) \in GraphSER\}$$

In other words, execution is serializable if it can be extended into an acyclic dependency graph.

Theorem

Let

$$GraphSI = \{\mathcal{G} \mid (\mathcal{T}_\mathcal{G} \vDash \text{INT}) \wedge$$
$$\left(\left((SO_\mathcal{G} \cup WR_\mathcal{G} \cup WW_\mathcal{G}) \, ; RW_\mathcal{G}^? \right) \text{ is acyclic} \right) \}$$
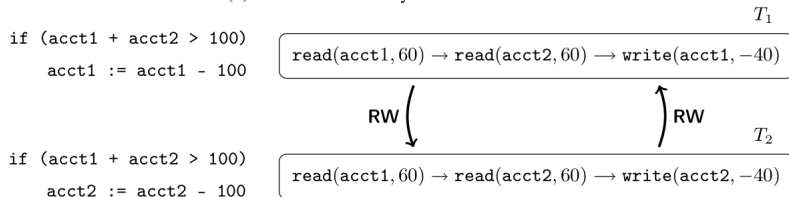
Then

$$HistSI = \{\mathcal{H} \mid \exists WR, WW, RW.(\mathcal{H}, WR, WW, RW) \in GraphSI\}$$

Where ...

- $R^? = R \cup \{(a, a) \mid a \in A\}$

(d) Write skew. Initially $\texttt{acct1} = \texttt{acct2} = 60$.

```
if (acct1 + acct2 > 100)
    acct1 := acct1 - 100
```

$$T_1$$

$$\boxed{\texttt{read}(\texttt{acct1}, 60) \rightarrow \texttt{read}(\texttt{acct2}, 60) \rightarrow \texttt{write}(\texttt{acct1}, -40)}$$

RW $\Big\downarrow$ $\Big\uparrow$ RW

$$T_2$$

```
if (acct1 + acct2 > 100)
    acct2 := acct2 - 100
```

$$\boxed{\texttt{read}(\texttt{acct1}, 60) \rightarrow \texttt{read}(\texttt{acct2}, 60) \rightarrow \texttt{write}(\texttt{acct2}, -40)}$$

- Prohibited under **serializability**, and has a dependency graph cycle $T_1 \xrightarrow{RW} T_2 \xrightarrow{RW} T_1$.
- However, allowed under **SI**

In contrast, following is *not* allowed under SI:

(b) Lost update.

(c) Long fork.
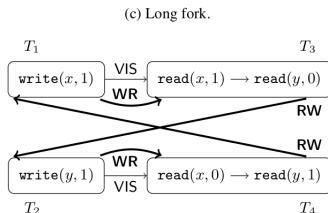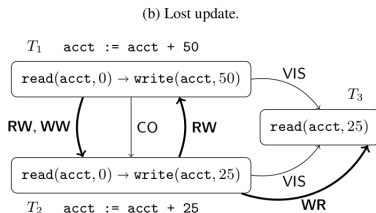
contain cycles without adjacent anti dependencies.

To prove the previous theorem we'll show a stronger result:

Theorem

1. **Soundness:** $\forall \mathcal{G} \in GraphSI. \exists \mathcal{X} \in ExecSI. graph(\mathcal{X}) = \mathcal{G}$
2. **Completeness:** $\forall \mathcal{X} \in ExecSI. graph(\mathcal{X}) \in GraphSI$

The *completeness* closely follows from existing results[2]. We will focus on the *soundness*.

Objective

Soundness: $\forall \mathcal{G} \in GraphSI.\exists \mathcal{X} \in ExecSI.graph(\mathcal{X}) = \mathcal{G}$

Proof sketch

- Construct a basic **pre-execution** from $\mathcal{G}$
- Iteratively extend it until satisfies execution definition

Definition

A tuple $\mathcal{P} = (\mathcal{T}, SO, VIS, CO)$ is a **pre-execution** if it satisfies all the conditions of being an *execution*, except *CO* is a *strict partial order* that may not be total.

Where ...

- *strict partial order* is a transitive and irreflexive relation

Definition

We let *PreExecSI* be the set of pre-executions satisfying the SI axioms:

$$PreExecSI = \{\mathcal{P} \mid \mathcal{P} \vDash \text{INT} \wedge \text{EXT} \wedge \text{SESSION} \wedge$$
$$\text{PREFIX} \wedge \text{NOCONFLICT}\}$$

- Consider $\mathcal{G} = (\mathcal{H}, WR, WW, RW)$, let $\mathcal{P} = (\mathcal{H}, VIS, CO)$ a respective pre-execution.
- VIS, CO must hold the following to conform with $\mathcal{G}$

$$SO \cup WR \cup WW \subseteq VIS$$
$$CO; VIS \subseteq VIS$$
$$VIS \subseteq CO$$
$$CO; CO \subseteq CO$$
$$VIS; RW \subseteq CO$$

Lemma

Let $\mathcal{G} = (\mathcal{T}, SO, WR, WW, RW)$ be a dependency graph, for any relation $R \subseteq \mathcal{T} \times \mathcal{T}$, the relations

$$VIS = (((SO \cup WR \cup WW); RW^?) \cup R)^*;$$
$$(SO \cup WR \cup WW)$$
$$CO = (((SO \cup WR \cup WW); RW^?) \cup R)^+$$

are a solution to the system of inequalities in the previous slide. They also are the smallest solution to the system for which $R \subseteq CO$.

Where...

- R^+ a transitive closure of R
- R^* a transitive and reflexive closure of R

FIXME add intuition

Proof outline.

Let $\mathcal{G} = (\mathcal{T}, SO, WR, WW, RW) \in GraphSI$

- Define $\mathcal{P}_0$ derived from the last lemma by fixing $R_0 = \emptyset$.
- Construct $\{\mathcal{P}_i = (\mathcal{T}, SO, VIS_i, CO_i)\}_{i=0}^n$ series of pre-executions.
- While CO_i is not total:
    - Pick arbitrary pair T, S not ordered by CO_i
    - $R_{i+1} = R_i \cup \{(T, S)\}$
    - Use the lemma with $R = R_{i+1}$ to derive VIS_{i+1}, CO_{i+1} (and thus $\mathcal{P}_{i+1}$)
- Let $\mathcal{X} = \mathcal{P}_n$ as CO_n is now total.

$\square$

Outline

Transaction Chopping under SI:

- We'll derive a static analysis that checks if transactions can be chopped into smaller sessions
- The analysis will suggest an optimized program provided any execution with chopped transactions does not exhibit new behaviors.

Definition

For history $\mathcal{H}$, let

$$\approx_{\mathcal{H}} = SO_{\mathcal{H}} \cup SO_{\mathcal{H}}^{-1} \cup \{(T, T) \mid T \in \mathcal{T}_{\mathcal{H}}\}$$

the equivalence relation grouping transactions from same session.

Definition

Let $\boxed{T}_{\mathcal{H}} = (E, po)$ where $E = (\bigcup \{E_S \mid S \approx_{\mathcal{H}} T\})$ and

$$po = \{(e, f) \mid \left(\exists S. e, f \in E_S \wedge e \xrightarrow{po_S} f \wedge S \approx_{\mathcal{H}} T \right) \vee$$
$$\left(\exists S, S'. e \in E_S \wedge f \in E_{S'} \wedge S \xrightarrow{SO_{\mathcal{H}}} S' \wedge S' \approx_{\mathcal{H}} T \right)\}$$

Informally, $\boxed{T}_{\mathcal{H}}$ is the result of splicing all transactions in session of T into the same transaction.
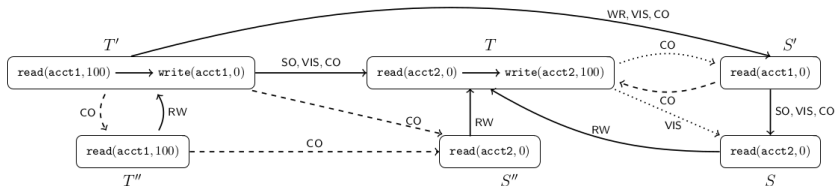
Definition

For history $\mathcal{H}$, let

$$splice(\mathcal{H}) = \left(\{ \boxed{T}_{\mathcal{H}} \mid T \in \mathcal{T}_{\mathcal{H}} \}, \emptyset \right)$$

history resulting from splicing all sessions in a history.

- We'll call $\mathcal{G} \in$ *GraphSI* **spliceable** if exists a dependency graph $\mathcal{G}' \in$ *GraphSI* such that $\mathcal{H}_{\mathcal{G}'} = splice(\mathcal{H}_{\mathcal{G}})$.
- For graph $\mathcal{G}$ we let $\approx_{\mathcal{G}} = \approx_{\mathcal{H}_{\mathcal{G}}}$.

Consider graph $\mathcal{G}$:

T', T: session transfer $\{ \text{tx} \{ \text{acct1} = \text{acct1} - 100 \}; \text{tx} \{ \text{acct2} = \text{acct2} + 100 \} \}$
T'': session lookup1 $\{ \text{tx} \{ \text{return acct1} \} \}$
S'': session lookup2 $\{ \text{tx} \{ \text{return acct2} \} \}$
S', S: session lookupAll $\{ \text{tx} \{ \text{var1} = \text{acct1} \}; \text{tx} \{ \text{var2} = \text{acct2} \}; \text{return var1} + \text{var2} \}$

Example

The above graph is not spliceable:

- $T' \xrightarrow{\text{VIS}} S'$
- $\neg T \xrightarrow{\text{VIS}} S$
- $\boxed{T}_{\mathcal{G}} \xrightarrow{\text{WR}(acct1)} \boxed{S}_{\mathcal{G}}$ but $\neg \boxed{T}_{\mathcal{G}} \xrightarrow{\text{WR}(acct2)} \boxed{S}_{\mathcal{G}}$

Definition

Given $\mathcal{G}$ let **dynamic chopping graph** $DCG(\mathcal{G})$ obtained from $\mathcal{G}$ by

- Removing $WR_\mathcal{G}$, $WW_\mathcal{G}$, $RW_\mathcal{G}$ edges between transactions related by $\approx_\mathcal{G}$
- Adding SO^{-1} edges

We'll classify the edges as following:

- SO - **successor** edges
- SO^{-1} - **predecessor** edges
- $(WR_\mathcal{G} \cup WW_\mathcal{G} \cup RW_\mathcal{G}) \setminus \approx_\mathcal{G}$ - **conflict** edges

Definition

Given $\mathcal{G}$ let **dynamic chopping graph** $DCG(\mathcal{G})$ obtained from $\mathcal{G}$ by

- Removing $WR_{\mathcal{G}}$, $WW_{\mathcal{G}}$, $RW_{\mathcal{G}}$ edges between transactions related by $\approx_{\mathcal{G}}$
- Adding SO^{-1} edges

We'll classify the edges as following:

- SO - **successor** edges
- SO^{-1} - **predecessor** edges
- $(WR_{\mathcal{G}} \cup WW_{\mathcal{G}} \cup RW_{\mathcal{G}}) \setminus \approx_{\mathcal{G}}$ - **conflict** edges

Definition

A cycle in $DCG(\mathcal{G})$ is **critical** if:

- Does not contain 2 occurrences of the same vertex
- Contains 3 consecutive edges in form of *conflict-predecessor-conflict*
- Any 2 anti dependency edges ($RW_\mathcal{G} \setminus \approx_\mathcal{G}$) are separated by at least one read ($WR_\mathcal{G} \setminus \approx_\mathcal{G}$) or write ($WW_\mathcal{G} \setminus \approx_\mathcal{G}$) dependency edge

Consider $\mathcal{G}$
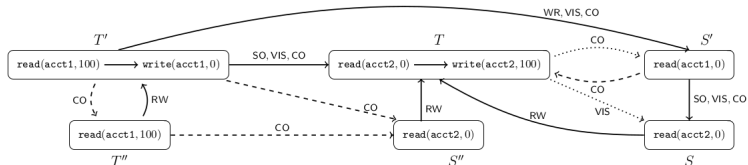
T', T: session transfer { tx { acct1 = acct1 - 100 }; tx { acct2 = acct2 + 100 } }
T'': session lookup1 { tx { return acct1 } }
S'': session lookup2 { tx { return acct2 } }
S', S: session lookupAll { tx { var1 = acct1 }; tx { var2 = acct2 }; return var1 + var2 }
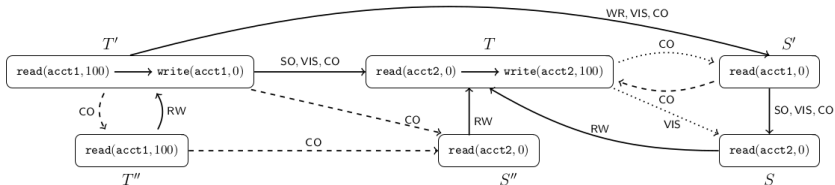
Example

$DCG(\mathcal{G})$ contains a critical cycle:

$$S' \xrightarrow{SO_{\mathcal{G}}} S \xrightarrow{RW_{\mathcal{G}}} T \xrightarrow{SO_{\mathcal{G}}^{-1}} T' \xrightarrow{WR_{\mathcal{G}}} S'$$

Theorem

For $\mathcal{G} \in$ GraphSI, if DCG($\mathcal{G}$) contains no critical cycles, then $\mathcal{G}$ is spliceable.

We use the last theorem to derive the static analysis.

- Assume set of **programs** $\mathcal{P} = \{P_1, P_2, \dots\}$, each defining code of a session resulting from chopping a single transaction.
- Each P_i is composed of k_i **program pieces**
- W_j^i and R_j^i sets of objects written or read by j-th piece of P_i

T', T: session transfer { tx { acct1 = acct1 - 100 }; tx { acct2 = acct2 + 100 } }
T'': session lookup1 { tx { return acct1 } }
S'': session lookup2 { tx { return acct2 } }
S', S: session lookupAll { tx { var1 = acct1 }; tx { var2 = acct2 }; return var1 + var2 }

Example

We have 4 *programs*, one for each session. Each transaction is a *program piece*.

For transfer session we have 2 program pieces with

- $T' : W_1^1 = R_1^1 = \{acct1\}$
- $T : W_2^1 = R_2^1 = \{acct2\}$

- History $\mathcal{H}$ **can be produced** by programs $\mathcal{P}$ if there's 1:1 correspondence between every session in $\mathcal{H}$ and program $P_i \in \mathcal{P}$, and each transaction in the session corresponds to respective program piece, along with its read/write sets.
- Chopping is defined **correct** if every dependency graph $\mathcal{G} \in$ *GraphSI*, where $\mathcal{H}_{\mathcal{G}}$ can be produced by $\mathcal{P}$ is splice-able.

Consider program set $\mathcal{P}$

Definition

Static chopping graph $SCG(\mathcal{P})$ is a graph where nodes are program pieces in form of (i, j) and the edge $(i_1, j_1), (i_2, j_2)$ is present if:

- $i_1 = i_2$ and:
    - $j_1 < j_2$ (a **successor** edge)
    - $j_1 > j_2$ (a **predecessor** edge)
- $i_1 \neq i_2$ and:
    - $W_{j_1}^{i_1} \cap R_{j_2}^{i_2} \neq \emptyset$ (a **read dependency** edge)
    - $W_{j_1}^{i_1} \cap W_{j_2}^{i_2} \neq \emptyset$ (a **write dependency** edge)
    - $R_{j_1}^{i_1} \cap W_{j_2}^{i_2} \neq \emptyset$ (an **anti dependency** edge)

- The edge set of static graphs $SCG(\mathcal{P})$ over-approximate the edge sets of the dynamic graphs $DCG(\mathcal{G})$ corresponding to graphs $\mathcal{G}$ produced by programs $\mathcal{P}$.
- The chopping defined by $\mathcal{P}$ is correct if $SCG(\mathcal{P})$ contains no critical cycles (as defined for dynamic graphs).
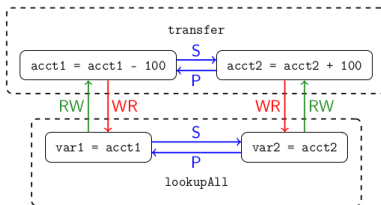
Figure 5: The static chopping graph of the programs {transfer, lookupAll} from Figure 4. Dashed boxes group program pieces into sessions.

Example

Fig. 5 contains a critical cycle:

$$(var1 = acct1) \xrightarrow{RW} (acct1 = acct1 - 100) \xrightarrow{S}$$

$$(acct2 = acct2 + 100) \xrightarrow{WR} (var2 = actt2) \xrightarrow{P} (var1 = acct1)$$
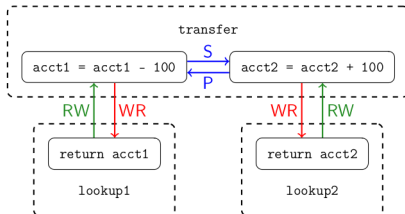
$\Rightarrow$ not a valid chopping

Figure 6: The static chopping graph of the programs {transfer, lookup1, lookup2} **from Figure 4.**

Example

Fig. 6 contains a single cycle, where two vertices appear twice $\Rightarrow$ not a critical cycle. The above chopping is spliceable.

Robustness:
We'll derive an analysis that check where an application behaves the same way under a weak consistency model as it does under a strong one.
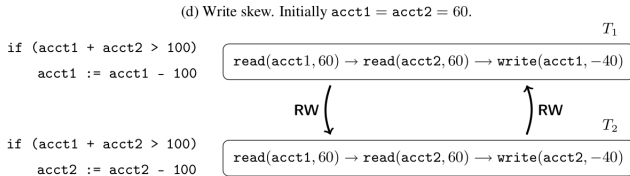
Robustness against SI towards SER

- Check if a given application running under **SI**, behaves the same as if it runs under **serializability** model.
- Specifically, no histories in *HistSI* \ *HistSER*

Theorem

For any $\mathcal{G}$, we have $\mathcal{G} \in GraphSI \setminus GraphSER$ iff. $\mathcal{T}_{\mathcal{G}} \vDash$ INT, $\mathcal{G}$ contains a cycle, and all its cycles have at least two adjacent anti-dependency edges.

Consider $\mathcal{G}$:

(d) Write skew. Initially $acct1 = acct2 = 60$.

```
if (acct1 + acct2 > 100)
    acct1 := acct1 - 100
```

T_1

$\boxed{read(acct1, 60) \rightarrow read(acct2, 60) \rightarrow write(acct1, -40)}$

RW

```
if (acct1 + acct2 > 100)
    acct2 := acct2 - 100
```

RW

T_2

$\boxed{read(acct1, 60) \rightarrow read(acct2, 60) \rightarrow write(acct2, -40)}$
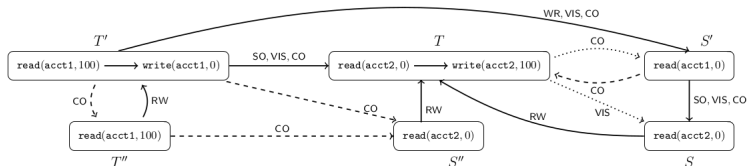
Example

The above graph contains a cycle with two adjacent anti-dependencies.
$\Rightarrow \mathcal{G} \in \textit{GraphSI} \setminus \textit{GraphSER}$

Consider $\mathcal{G}$:

T', T: session transfer { tx { acct1 = acct1 - 100 }; tx { acct2 = acct2 + 100 } }
T'': session lookup1 { tx { return acct1 } }
S'': session lookup2 { tx { return acct2 } }
S', S: session lookupAll { tx { var1 = acct1 }; tx { var2 = acct2 }; return var1 + var2 }

Example

The above graph contains no cycles.
$\Rightarrow \mathcal{G} \notin GraphSI \setminus GraphSER \land \mathcal{G}_2 \in GraphSER$

Static analysis:

- Assume code of transactions defined by set of programs $\mathcal{P}$ with given read and write sets.
- Based on them, derive **static dependency graph**, over-approximating possible dependencies that can exist.
- Check that static dependency graph contains no cycles with two adjacent anti-dependency edges.

Robustness against PSI towards SI

- Check if a given application running under **PSI**, behaves the same as if it runs under **SI** model.
- Again, make sure there are no histories in *HistPSI \ HistSI*

Definition

Sets of executions and histories **allowed by parallel SI** are:

$$ExecPSI = \begin{aligned} \{\mathcal{X} \mid \mathcal{X} \vDash &\text{INT} \wedge \text{EXT} \wedge \text{SESSION} \wedge \\ &\text{TRANSVIS} \wedge \text{NOCONFLICT}\} \end{aligned}$$

$$HistPSI = \{\mathcal{H} \mid \exists VIS, CO.(\mathcal{H}, VIS, CO) \in ExecPSI\}$$

Definition

TRANSVIS axiom ensures that transactions ordered by *VIS* are observed by others in this order. However, allows transactions unrelated by *VIS* to be observed in different orders; in particular, allows *long fork* anomaly.

Theorem

Let

$$GraphPSI = \{\mathcal{G} \mid (\mathcal{T}_\mathcal{G} \vDash \text{INT}) \wedge$$
$$\left(\left((SO_\mathcal{G} \cup WR_\mathcal{G} \cup WW_\mathcal{G})^+ ; RW_\mathcal{G}^? \right) \text{ is irreflexive} \right) \}$$

Then

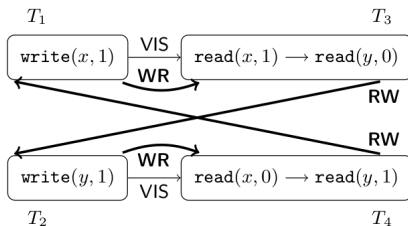$$HistPSI = \{\mathcal{H} \mid \exists WR, WW, RW.(\mathcal{H}, WR, WW, RW) \in GraphPSI\}$$

Theorem

For any $\mathcal{G}$, we have $\mathcal{G} \in GraphPSI \setminus GraphSI$ iff. $\mathcal{T}_\mathcal{G} \vDash \text{INT}$, $\mathcal{G}$ contains at least one cycle with no adjacent anti-dependency edges, and all its cycles have at least two anti dependency edges.

Consider $\mathcal{G}$:

(c) Long fork.

Example

Contains a cycle, with two *non-adjacent* anti-dependencies.
$\Rightarrow \mathcal{G} \in GraphPSI \setminus GraphSI$

Static analysis:
In similar fashion to the previous robustness analysis, desired criteria:

- *Static dependency graph* contains no cycles where there are at least two anti-dependency edges and no two anti-dependency edges are adjacent.

Thank you!