Analyzing Snapshot Isolation

Andrea Cerone *Alexey Gotsman*

IMDEA Software Institute

PODC 2016

presented by Dima Kuznetsov

Agenda

1. Introduction
2. Snapshot Isolation
   - Definitions
   - Dependency Graphs
   - Characterization
3. Static Analysis
   - Transaction Chopping
   - Robustness

- We focus on *Snapshot Isolation* presented in the last talk
- ... same context of DBMS and transactional memory systems
- We *won't* focus on the replicated aspect of the data-bases, but rather on semantics of concurrent user sessions.

- 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.*

[1] https://en.wikipedia.org/wiki/Snapshot_isolation#Definition

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

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

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.

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.

Assumptions:

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

Some notation:

- We'll use $(T, S) \in \mathit{VIS}$ and $T \xrightarrow{\mathit{VIS}} S$ interchangeably for *VIS* and other relations.
- For $\mathcal{H} = (\mathcal{T}, \mathit{SO})$ we will shorten $(\mathcal{T}, \mathit{SO}, \mathit{VIS}, \mathit{CO})$ to $(\mathcal{H}, \mathit{VIS}, \mathit{CO})$
- For $\mathcal{H} = (\mathcal{T}, \mathit{SO})$ 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 define *snapshot isolation* and *serializability* in terms of **consistency axioms**:

Definition

$$\mathit{ExecSI} = \{\mathcal{X} \mid \mathcal{X} \models \textsc{Int} \wedge \textsc{Ext} \wedge \textsc{Session} \wedge \\ \textsc{Prefix} \wedge \textsc{NoConflict}\}$$

$$\mathit{ExecSER} = \{\mathcal{X} \mid \mathcal{X} \models \textsc{Int} \wedge \textsc{Ext} \wedge \textsc{Session} \wedge \textsc{TotalVis}\}$$

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

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

Where ...

- $\mathcal{X} \models \textsc{Prop}$ means that execution does not violate property $\textsc{Prop}$

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 ...

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

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 ...

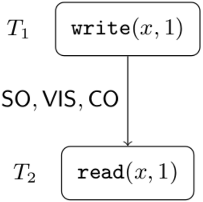
- $T \vdash read(x, n)$ if T reads from x and n is the value of x at the first read.
- $T \vdash write(x, n)$ if T writes to x and n is the final value of x.
- $R^{-1}(a) = \{b \mid (b, a) \in R\}$
- $WriteTx_x = \{T \mid T \vdash write(x, \_)\}$

Definition (Session visibility)

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

$$SO \subseteq VIS$$

(a) Session guarantees.



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 ...

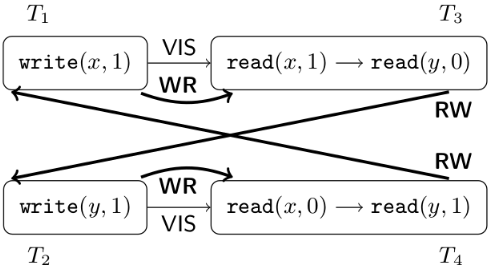
- $R_1 ; R_2 = \{(a, b) \mid \exists c : (a, c) \in R_1 \wedge (c, b) \in R_2\}$

Stronger requirement than VIS transitivity we required from *causal consistency* and *parallel snapshot isolation*.

$$VIS \subseteq CO \wedge CO; VIS \subseteq VIS \Rightarrow VIS; VIS \subseteq VIS$$

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

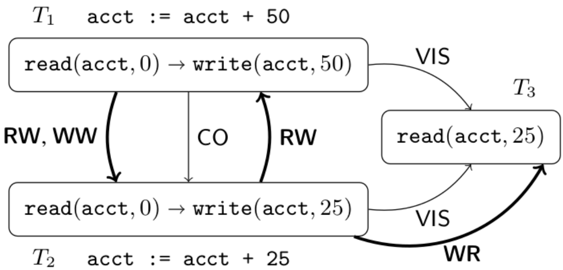
Definition (No conflict check)

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

$$\forall T, S \in \mathcal{T}. \forall x, n.$$
$$(T, S \in \mathit{WriteTx}_x \wedge T \neq S) \Rightarrow \left(T \xrightarrow{\mathit{VIS}} S \vee S \xrightarrow{\mathit{VIS}} T \right)$$

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.

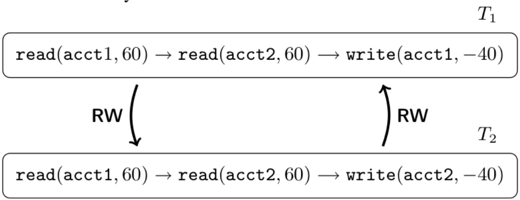
$$\mathit{VIS} = \mathit{CO}$$

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

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

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

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



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

A **dependency graph** is a tuple $\mathcal{G} = (\mathcal{T}, SO, WR, WW, RW)$, where $(\mathcal{T}, SO)$ is a history and for each $x \in Obj$ we define relations $WR(x), WW(x), RW(x)$ that satisfy the following:

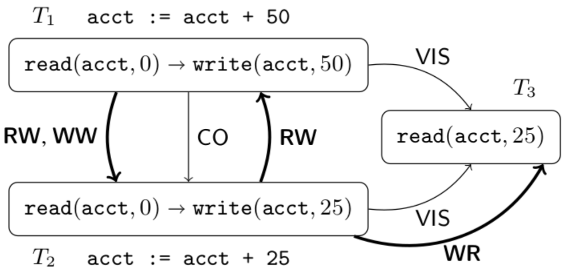
- $WR(x)$:
  - $\forall T, S. T \xrightarrow{WR(x)} S \Rightarrow \exists n. T \neq S \wedge T \vdash write(x, n) \wedge S \vdash read(x, n)$
  - $\forall S \in \mathcal{T}. S \vdash read(x, \_) \Rightarrow \exists T. T \xrightarrow{WR(x)} S$
  - $\forall T, T', S \in \mathcal{T}. \left(T \xrightarrow{WR(x)} S \wedge T' \xrightarrow{WR(x)} S\right) \Rightarrow T = T'$
- $WW(x)$ is a total order over $WriteTx_x$.
- $RW(x)$ is derived from $WR(x)$ and $WW(x)$ such that $T \xrightarrow{RW(x)} S \Leftrightarrow T \neq S \wedge \exists T'. T' \xrightarrow{WR(x)} T \wedge T' \xrightarrow{WW(x)} S$

We'll use WW to denote $\bigcup_{x \in Obj} WW(x)$ for WW and the other two relations.

Informally,

- $T \xrightarrow{WR(x)} S$ means that S reads T's write to x.
  - We'll call an edge in WR a **read dependency**
- $T \xrightarrow{WW(x)} S$ means that S overwrites T's write to x.
  - We'll call an edge in WW a **write dependency**
- $T \xrightarrow{RW(x)} S$ means that S overwrites the write to x read by T.
  - We'll call an edge in RW an **anti dependency**

(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

Consider execution $\mathcal{X} = (\mathcal{T}, SO, VIS, CO)$, for $x \in Obj$ we define relations $WR_{\mathcal{X}}, WW_{\mathcal{X}}, RW_{\mathcal{X}}$ that satisfy the following:

- $T \xrightarrow{WR_{\mathcal{X}}(x)} S \Leftrightarrow$
  $$S \vdash read(x, n) \wedge T = max_{CO}\left(VIS^{-1}(S) \cap WriteTx_x\right)$$
- $T \xrightarrow{WW_{\mathcal{X}}(x)} S \Leftrightarrow T \xrightarrow{CO} S \wedge T, S \in WriteTx_x$
- $T \xrightarrow{RW_{\mathcal{X}}(x)} S \Leftrightarrow T \neq S \wedge \exists T'. T' \xrightarrow{WR_{\mathcal{X}}(x)} T \wedge T' \xrightarrow{WW_{\mathcal{X}}(x)} S$

Proposition

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

$$\mathit{graph}(\mathcal{X}) = (\mathcal{T}_{\mathcal{X}}, \mathit{SO}_{\mathcal{X}}, \mathit{WR}_{\mathcal{X}}, \mathit{WW}_{\mathcal{X}}, \mathit{RW}_{\mathcal{X}})$$

is a dependency graph.

Proof

By showing $\mathit{graph}(\mathcal{X})$ 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

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

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

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

Then

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

Where ...

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

The relation

$$(SO_{\mathcal{G}} \cup WR_{\mathcal{G}} \cup WW_{\mathcal{G}}); RW_{\mathcal{G}}^{?}$$

includes edges of the following form:

- SO
- $SO; RW$
- WR
- $WR; RW$
- WW
- $WW; RW$

If there is a cycle in $\mathcal{G}$, it:

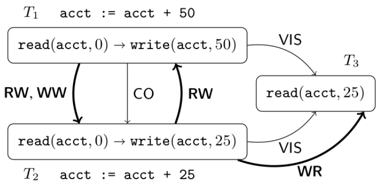
- cannot be composed only of $SO \cup WR \cup WW$ edges, otherwise it is a cycle in CO
- cannot contain only non-adjacent anti-dependency, otherwise above relation is cyclic.

$\Rightarrow$ any cycle has to have at least two adjacent anti dependencies.

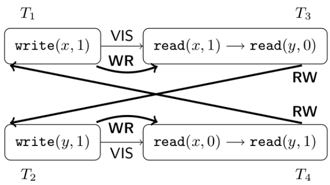
The latter type of cycles are disallowed under SI, they allow *long fork* and *lost update* anomalies.

Prohibited under SI:

(b) Lost update.



(c) Long fork.

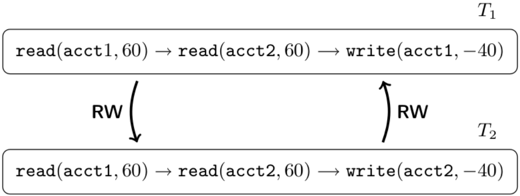


contain cycles without adjacent anti dependencies.

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

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

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



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

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

Theorem

1. ***Soundness:***

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

2. ***Completeness:***

$$\forall \mathcal{X} \in \mathit{ExecSI}. \mathit{graph}(\mathcal{X}) \in \mathit{GraphSI}$$

Theorem

Completeness:

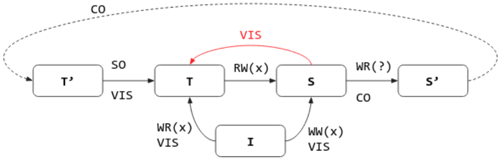
$$\forall \mathcal{X} \in \mathit{ExecSI}.\mathit{graph}(\mathcal{X}) \in \mathit{GraphSI}$$

Closely follows from existing results[2].

[2] Making Snapshot Isolation Serializable, 2005, A. Fekete et al

Consider SI execution, if we have a cycle in the dependency graph:

- then it contains at least one *RW* edge (other types of edges included on the *CO*)
- if the *RW* edge does not have an adjacent *RW* edge, it has to be included in the *CO*:



- red edge entailed by the *CO* edge from S to T', and *VIS* edge from T' to T (PREFIX: $CO; VIS \subseteq VIS$)
- it violates EXT axiom, T reads x from I though S is visible and ordered after I in the commit order.

Theorem

Soundness:

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

Proof sketch

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

Definition

A tuple $\mathcal{P} = (\mathcal{T}, \mathit{SO}, \mathit{VIS}, \mathit{CO})$ 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:

$$\mathit{PreExecSI} = \{\mathcal{P} \mid \mathcal{P} \models \textsc{Int} \wedge \textsc{Ext} \wedge \textsc{Session} \wedge \textsc{Prefix} \wedge \textsc{NoConflict}\}$$

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

$\mathit{SO} \cup \mathit{WR} \cup \mathit{WW} \subseteq \mathit{VIS}$	*VIS* must conform with read/write dependecies of $\mathcal{G}$ and with *SO* to hold SESSION axiom.
$\mathit{CO}; \mathit{VIS} \subseteq \mathit{VIS}$	to ensure PREFIX axiom.
$\mathit{VIS} \subseteq \mathit{CO}$	to conform with SI definition - only committed transactions in snapshots.
$\mathit{CO}; \mathit{CO} \subseteq \mathit{CO}$	transitivity of *CO*
$\mathit{VIS}; \mathit{RW} \subseteq \mathit{CO}$	to ensure EXT axiom

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

$$
\begin{aligned}
VIS &= (((SO \cup WR \cup WW); RW^{?}) \cup R)^{*}; \\
&\quad (SO \cup WR \cup WW) \\
CO &= (((SO \cup WR \cup WW); RW^{?}) \cup R)^{+}
\end{aligned}
$$

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

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.

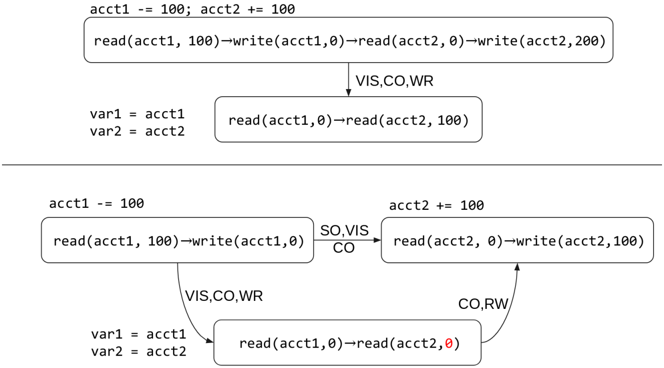
$\square$

Outline

1. Introduction
2. Snapshot Isolation
   - Definitions
   - Dependency Graphs
   - Characterization
3. Static Analysis
   - Transaction Chopping
   - Robustness

Transaction Chopping under SI:

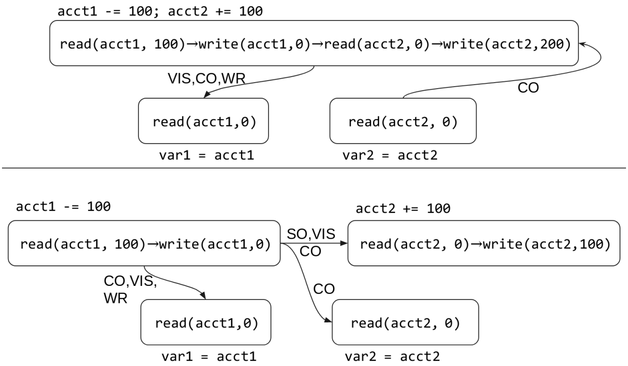
- We'll derive a static analysis that checks if transactions can be chopped into smaller sessions
- The analysis will suggest an optimized program provided any execution with chopped transactions does not exhibit new behaviors.



Example

Given the top graph, can we chop the top transaction into 2 parts?

No... bottom transaction will be able to observe intermittent state.



Example

What if we decouple the reads into separate sessions?

Each read is allowed to observe the transfer in any order, no new observable 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

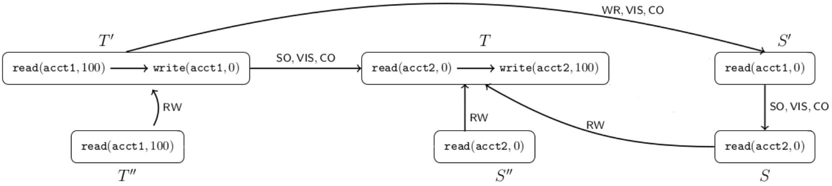
$$\mathit{splice}(\mathcal{H}) = \left(\{\boxed{T}_{\mathcal{H}} \mid T \in \mathcal{T}_{\mathcal{H}}\}, \emptyset\right)$$

history resulting from splicing all sessions in a history.

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

Intuitively, if $\mathcal{G}$ is spliceable, then $\mathit{splice}(\mathcal{H}_{\mathcal{G}})$ can be chopped into $\mathcal{H}_{\mathcal{G}}$.

Consider graph $\mathcal{G}$:



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 is not spliceable:

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

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

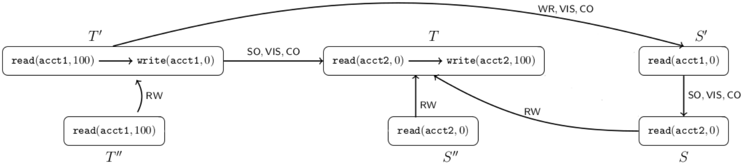
Definition

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

Intuitively, we want criteria for $DCG(\mathcal{G})$ that translates into an invalid cycle on the spliced graph.

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.

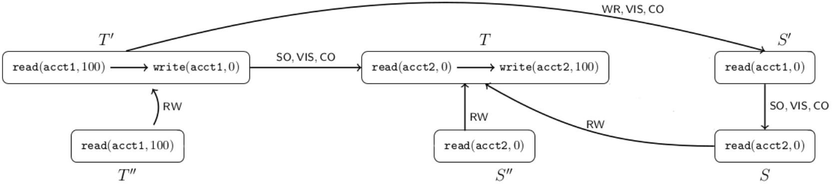
Specifically:

- If $\mathcal{G}$ is spliceable, then the spliced graph $\mathcal{G}' \in$ *GraphSI*
- $\Rightarrow \mathcal{G}' \in$ *GraphSI* so it contains no cycles without adjacent anti-dependencies
- $\Rightarrow$ we might be able to chop $\mathcal{G}'$ into $\mathcal{G}$

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^i_j and R^i_j sets of objects written or read by j-th piece of P_i

- 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 $\mathcal{G} \in \mathit{GraphSI}$, where $\mathcal{H}_{\mathcal{G}}$ can be produced by $\mathcal{P}$ is spliceable.



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

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

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:
  - $W^{i_1}_{j_1} \cap R^{i_2}_{j_2} \neq \emptyset$ (a **read dependency** edge)
  - $W^{i_1}_{j_1} \cap W^{i_2}_{j_2} \neq \emptyset$ (a **write dependency** edge)
  - $R^{i_1}_{j_1} \cap W^{i_2}_{j_2} \neq \emptyset$ (an **anti dependency** edge)

- The edge set of static graphs $\mathit{SCG}(\mathcal{P})$ over-approximate the edge sets of the dynamic graphs $\mathit{DCG}(\mathcal{G})$ corresponding to graphs $\mathcal{G}$ produced by programs $\mathcal{P}$.
- The chopping defined by $\mathcal{P}$ is correct if $\mathit{SCG}(\mathcal{P})$ contains no critical cycles (as defined for dynamic graphs).

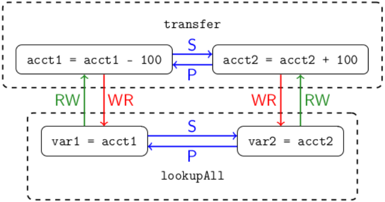


Figure 5: The static chopping graph of the programs {`transfer`, `lookupAll`} from Figure 4. Dashed boxes group program pieces into sessions.

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

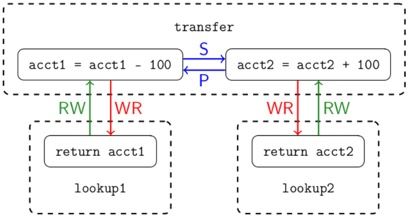


Figure 6: The static chopping graph of the programs {`transfer`, `lookup1`, `lookup2`} **from Figure 4.**

Example

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

Outline

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

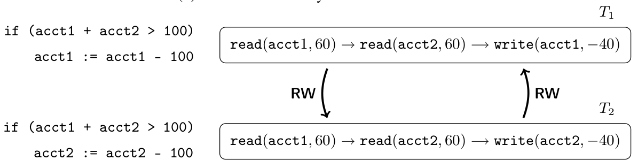
- Check if a given application running under **SI**, behaves the same as if it runs under **serializability** model.
- Specifically, no histories in $\mathit{HistSI} \setminus \mathit{HistSER}$

Theorem

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

Consider $\mathcal{G}$:

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



Example

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

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 $\mathit{HistPSI} \setminus \mathit{HistSI}$

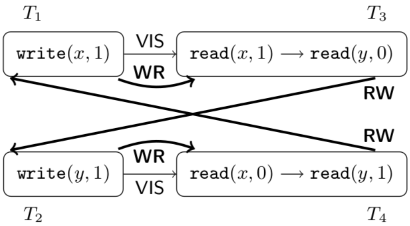
Definition

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

$$\mathit{ExecPSI} = \begin{array}{l}\{\mathcal{X} \mid \mathcal{X} \models \textsc{Int} \wedge \textsc{Ext} \wedge \textsc{Session} \wedge \\ \qquad\qquad \textsc{TransVis} \wedge \textsc{NoConflict}\}\end{array}$$

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

(c) Long fork.



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

$$\mathit{GraphPSI} = \{\mathcal{G} \mid (\mathcal{T}_{\mathcal{G}} \models \textsc{Int}) \wedge \left(\left((\mathit{SO}_{\mathcal{G}} \cup \mathit{WR}_{\mathcal{G}} \cup \mathit{WW}_{\mathcal{G}})^{+} ; \mathit{RW}_{\mathcal{G}}^{?}\right) \text{ is irreflexive}\right)\}$$

Then

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

Theorem

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

Static analysis:

- Similar to SI/SER case, but checks the static dependency graph for the above criteria.

Thank you!