

Analyzing Snapshot Isolation

Andrea Cerone Alexey Gotsman

IMDEA Software Institute

PODC 2016

Outline

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

Intro

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

Intro

- 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 → more concurrent transactions expected to abort

Snapshot Isolation

Snapshot Isolation - originally specified as operational model:

- When transaction T begins, $snapshot_T$ is taken
- All reads in T read values from $snapshot_T$
- All writes in T write to transient write-set
- T commits only if passes write conflict check:
 - No object in T 's write-set was updated by other transactions since $snapshot_T$ was taken

Outline

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

Definitions

We'll start by formulating a declarative definition of Snapshot Isolation.

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
- $Op = \{read(x, n), write(x, n) \mid x \in Obj, n \in \mathbb{Z}\}$
- $op : Event \rightarrow Op$

- **Strict partial order** - transitive + irreflexive relation
- **Total order** - strict partial order that orders all pairs

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

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.

We elide treatment of:

- aborted transactions - all transactions in any history are committed.
- infinite computations - histories are always finite.

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 $(T, S) \in VIS$ and $T \xrightarrow{VIS} S$ interchangeably for VIS and other relations.
- For $\mathcal{H} = (\mathcal{T}, SO)$ we'll shorten $(\mathcal{T}, SO, VIS, CO)$ to (\mathcal{H}, VIS, CO)

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.

For a set A and a total order $R \subseteq A \times A$

- $\max_R(A) = \{a \mid \forall b \in A : a = b \vee (b, a) \in R\}$
- $\min_R(A) = \{a \mid \forall b \in A : a = b \vee (a, b) \in R\}$
- $R^{-1}(a) = \{b \mid (b, a) \in R\}$
- $R_1; R_2 = \{(a, b) \mid \exists c : (a, c) \in R_1 \wedge (c, b) \in R_2\}$
- $R? = R \cup \{(a, a) \mid a \in A\}$
- R^+ a transitive closure of R
- R^* a transitive and reflexive closure of R

For $T = (E, po)$ we'll use:

- $T \vdash \text{write}(x, n)$ if T writes to x and n is s.t.

$$op(\max_{po}\{e \mid op(e) = \text{write}(x, _)\}) = \text{write}(x, n)$$

- $T \vdash \text{read}(x, n)$ if T reads x before writing to x and n is s.t.

$$op(\min_{po}\{e \mid op(e) = _(x, _)\}) = \text{read}(x, n)$$

We'll also use $\text{Write}Tx_x = \{T \mid T \vdash \text{write}(x, _)\}$

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

$$ExecSI = \{\mathcal{X} \mid \mathcal{X} \models INT \wedge EXT \wedge SESSION \wedge \\ PREFIX \wedge NOCONFLICT\}$$

$$ExecSER = \{\mathcal{X} \mid \mathcal{X} \models INT \wedge EXT \wedge SESSION \wedge TOTALVIS\}$$

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

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

INT - internal consistency axiom: 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.

$$\forall (E, po) \in \mathcal{T}. \forall e \in E. \forall x, n :$$

$$\begin{aligned} op(e) = read(x, n) \wedge \{f \mid op(f) = _ (x, _) \wedge f \xrightarrow{po} e\} \neq \emptyset \Rightarrow \\ op\left(\max_{po}\{f \mid op(f) = _ (x, _) \wedge f \xrightarrow{po} e\}\right) = _ (x, n) \end{aligned}$$

EXT - external consistency axiom: ensures that if $T \vdash \text{read}(x, n)$ then the value is taken from the last visible transaction that wrote to x according to commit order.

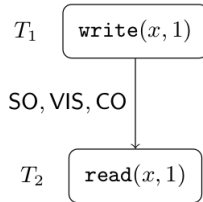
$$\forall T \in \mathcal{T}. \forall x, n :$$

$$T \vdash \text{read}(x, n) \Rightarrow \max_{CO} \left(\text{VIS}^{-1}(T) \cap \text{WriteTx}_x \right) \vdash \text{write}(x, n)$$

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

$$SO \subseteq VIS$$

(a) Session guarantees.

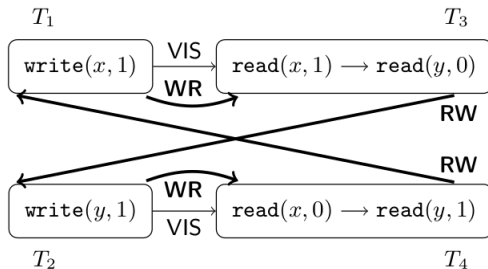


PREFIX - ensures that if snapshot taken by T includes S , then it includes all transactions committed before S as well.

$$CO; VIS \subseteq VIS$$

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

(c) Long fork.



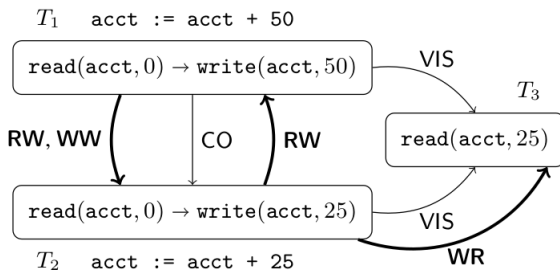
NOCONFLICT - ensures that for any two transactions writing to the same object, one has to be aware of the other.

$$\forall T, S \in \mathcal{T}. \forall x, n.$$

$$(T, S \in \text{WriteTx}_x \wedge T \neq S) \Rightarrow \left(T \xrightarrow{\text{VIS}} S \vee S \xrightarrow{\text{VIS}} T \right)$$

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

(b) Lost update.



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 $\text{acct1} = \text{acct2} = 60$.

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

T_1
read(acct1, 60) \rightarrow read(acct2, 60) \rightarrow write(acct1, -40)

RW \downarrow

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

T_2
read(acct1, 60) \rightarrow read(acct2, 60) \rightarrow write(acct2, -40)

\uparrow RW

Outline

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

- 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.

Let $\mathcal{X} = (\mathcal{T}, SO, VIS, CO)$ be an execution, for $x \in Obj$ we define the following relations on $\mathcal{T}_{\mathcal{X}}$:

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

$$T \xrightarrow{WR_{\mathcal{X}}(x)} S \Leftrightarrow$$

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

Informally: $T \xrightarrow{WR_{\mathcal{X}}(x)} S$ means that S reads T 's write to x .

Let $\mathcal{X} = (\mathcal{T}, SO, VIS, CO)$ be an execution, for $x \in Obj$ we define the following relations on $\mathcal{T}_{\mathcal{X}}$:

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

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

Informally: $T \xrightarrow{WW_{\mathcal{X}}(x)} S$ means that S overwrites T 's write to x .

Let $\mathcal{X} = (\mathcal{T}, SO, VIS, CO)$ be an execution, for $x \in Obj$ we define the following relations on $\mathcal{T}_{\mathcal{X}}$:

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

$$T \xrightarrow{RW_{\mathcal{X}}(x)} S \Leftrightarrow$$

$$T \not\preceq S \wedge \exists T'. T' \xrightarrow{WR_{\mathcal{X}}(x)} T \wedge T' \xrightarrow{WW_{\mathcal{X}}(x)} S$$

Informally: $T \xrightarrow{RW_{\mathcal{X}}(x)} S$ means that S overwrites the write to x read by T .

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:
 - $\forall T, S. \forall x. T \xrightarrow{WR(x)} S \Rightarrow \exists n. T \neq S \wedge T \vdash \text{write}(x, n) \wedge S \vdash \text{read}(x, n)$
 - $\forall S \in \mathcal{T}. \forall x. S \vdash \text{read}(x, _) \Rightarrow \exists T. T \xrightarrow{WR(x)} S$
 - $\forall T, T', S \in \mathcal{T}. \forall x. \left(T \xrightarrow{WR(x)} S \wedge T' \xrightarrow{WR(x)} S \right) \Rightarrow T = T'$

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

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

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

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

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

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

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

Theorem:

Let

$$\text{GraphSER} = \{ \mathcal{G} \mid (\mathcal{T}_{\mathcal{G}} \models \text{INT}) \\ ((\text{SO}_{\mathcal{G}} \cup \text{WR}_{\mathcal{G}} \cup \text{WW}_{\mathcal{G}} \cup \text{RW}_{\mathcal{G}}) \text{ is acyclic}) \}$$

Then

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

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

Theorem:

Let

$$GraphSI = \{\mathcal{G} \mid (\mathcal{T}_{\mathcal{G}} \models INT) \wedge$$
$$(((SO_{\mathcal{G}} \cup WR_{\mathcal{G}} \cup WW_{\mathcal{G}}); RW_{\mathcal{G}}?) \text{ is acyclic})\}$$

Then

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

Similarly, $\mathcal{H} \in HistSI$ if \mathcal{H} can be extended to a dependency graph satisfying *GraphSI*

Example:

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

T_1

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

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

RW \downarrow

\uparrow RW

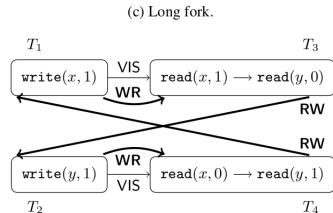
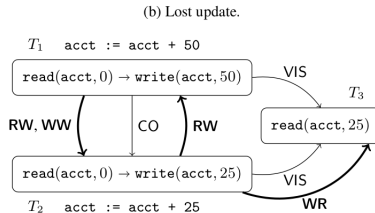
T_2

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

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

- 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:



contain cycles without adjacent anti dependencies.

To prove it we'll show a stronger result:

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

The *completeness* closely follows from existing results¹. We will focus on the *soundness*.

¹Making Snapshot Isolation Serializable, 2005, A. Fekete et al.

Proof sketch:

- Construct a basic **pre-execution** from \mathcal{G}
- Iteratively extend it until satisfies execution 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 strong partial order that may not be total. We let $PreExecSI$ be the set of pre-executions satisfying the SI axioms:

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

- For a given dep. graph $\mathcal{G} = (\mathcal{H}, WR, WW, RW)$, let $\mathcal{P} = (\mathcal{H}, VIS, CO)$ a respective pre-execution.
- To conform with \mathcal{G} we require that VIS, CO hold:

$$SO \cup WR \cup WW \subseteq VIS \quad (1)$$

$$CO; VIS \subseteq VIS \quad (2)$$

$$VIS \subseteq CO \quad (3)$$

$$CO; CO \subseteq CO \quad (4)$$

$$VIS; RW \subseteq CO \quad (5)$$

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 $R \subseteq CO$.

Back to the proof:

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

- 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.

For history \mathcal{H} , let

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

be the equivalence relations grouping transactions from the same session.

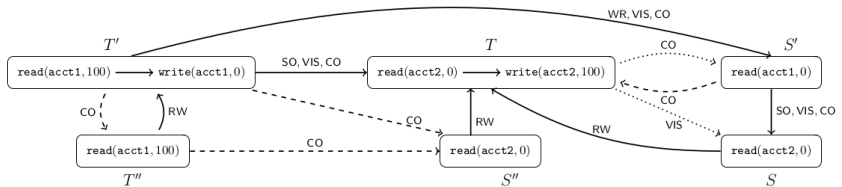
Let $\boxed{T}_{\mathcal{H}} = (E, po)$ where $E = (\bigcup \{E_S \mid S \approx_{\mathcal{H}} T\})$ and

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

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

- 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.
- $\mathcal{G} \in GraphSI$ is **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 $\approx_{\mathcal{G}} = \approx_{\mathcal{H}_{\mathcal{G}}}$.

Example, let 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 }

```

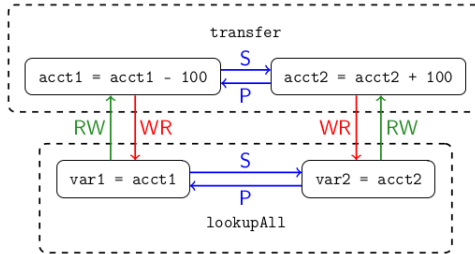
The above graph is not splice-able: $\boxed{S}_{\mathcal{G}}$ observes write by $\boxed{T}_{\mathcal{G}}$ to *acct1* but not to *acct2*.

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 **predecessor** edges $SO_{\mathcal{G}}^{-1}$
- We'll call SO **successor** edges
- And call $(WR_{\mathcal{G}} \cup WW_{\mathcal{G}} \cup RW_{\mathcal{G}}) \setminus \approx_{\mathcal{G}}$ **conflict** edges

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 edges.



This example contains a critical cycle with

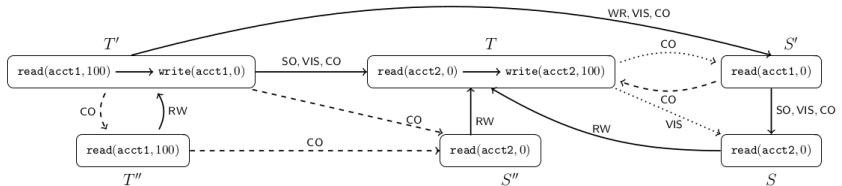
$$\cdot \xrightarrow{S} \cdot \xrightarrow{WR} \cdot \xrightarrow{P} \cdot \xrightarrow{RW} \cdot$$

Theorem:

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

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 }

For *transfer* session we have 2 program pieces with

- $W_1^1 = R_1^1 = \{acct1\}, W_2^1 = R_2^1 = \{acct2\}$

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

We check correctness of chopping \mathcal{P} using its **static chopping graph** $SCG(\mathcal{P})$.

Graph's 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 edges set of the dynamic graphs $DCG(\mathcal{G})$ for 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).

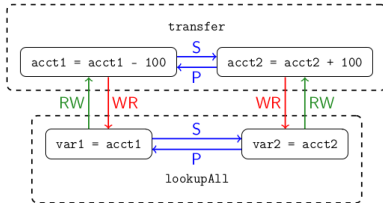


Figure 5: The static chopping graph of the programs $\{\text{transfer}, \text{lookupAll}\}$ from Figure 4. Dashed boxes group program pieces into sessions.

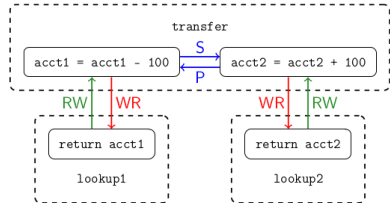


Figure 6: The static chopping graph of the programs $\{\text{transfer}, \text{lookup1}, \text{lookup2}\}$ from Figure 4.

Outline

1 Introduction

2 Snapshot Isolation

- Definitions
- Dependency Graphs
- Characterization

3 Static Analysis

- Transaction Chopping
- **Robustness**

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 $HistSI \setminus HistSER$

Theorem:

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

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.
- Specifically, no histories in $HistPSI \setminus HistSI$

Definition:

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

$$\begin{aligned} \text{ExecPSI} &= \{ \mathcal{X} \mid \mathcal{X} \models \text{INT} \wedge \text{EXT} \wedge \text{SESSION} \wedge \\ &\quad \text{TRANSVIS} \wedge \text{NOCONFLICT} \} \\ \text{HistPSI} &= \{ \mathcal{H} \mid \exists \text{VIS}, \text{CO}. (\mathcal{H}, \text{VIS}, \text{CO}) \in \text{ExecPSI} \} \end{aligned}$$

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

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

Then

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

Theorem:

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

Thank you!