Robustness against read committed: a free transactional lunch

Frank Neven

UHasselt, Data Science Institute, ACSL

PODS 2022

Medieval town of Gruyères

Picture from Tripadvisor

Concurrent transactions & Swiss cheese fondue @Gruyères

Bas Ketsman
Vrije Universiteit Brussel

Christoph Koch
EPFL

Brecht Vandevoort
Universiteit Hasselt

Outline

Outline

Database transactions: concurrent access to data

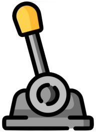
A balancing act

Database transactions: concurrent access to data

A balancing act

No Isolation | *Read Committed* | *Repeatable Read* | *Serializable*

Higher throughput
High number of possible data anomaly types



Isolation Level

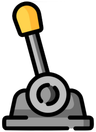
Lower throughput
Low number of possible data anomaly types

Database transactions: concurrent access to data

A balancing act

No Isolation | *Read Committed* | *Repeatable Read* | *Serializable*

Higher throughput
High number of possible data anomaly types



Isolation Level

Lower throughput
Low number of possible data anomaly types

Free lunch: given more knowledge on workload, can you choose a lower isolation level but still have maximal data consistency?

Data inconsistency

Transaction 1 *Withdraw €50 from account A*		Accounts
		$A = €400$ $B = €500$
Get balance $A \rightarrow €400$		

Data inconsistency

Transaction 1 *Withdraw €50 from account A*		Accounts
		$A = €400$ $B = €500$
Get balance $A \rightarrow €400$		
Compute new value		

Data inconsistency

Transaction 1 *Withdraw €50 from account A*		Accounts
		$A = €400$ $B = €500$
Get balance $A \to €400$		
Compute new value		
Set $A = €350$		$A = €350$
Commit		$B = €500$

Data inconsistency

Transaction 1 *Withdraw €50 from account A*	**Transaction 2** *Transfer €400 from account A to B*	Accounts
		$A = €400$ $B = €500$
Get balance $A \rightarrow €400$		
	Get balance $A \rightarrow €400$	
	Get balance $B \rightarrow €500$	
	Compute new values	
Compute new value		

Data inconsistency

Transaction 1 *Withdraw €50 from account A*	**Transaction 2** *Transfer €400 from account A to B*	Accounts
		$A = €400$ $B = €500$
Get balance $A \rightarrow €400$		
	Get balance $A \rightarrow €400$	
	Get balance $B \rightarrow €500$	
Compute new value	*Compute new values*	
	Set $A = €0$	$A = €0$
	Set $B = €900$	$B = €900$
	Commit	

Data inconsistency

Transaction 1 *Withdraw €50 from account A*	**Transaction 2** *Transfer €400 from account A to B*	Accounts
		$A = €400$ $B = €500$
Get balance $A \to €400$		
	Get balance $A \to €400$	
	Get balance $B \to €500$	
Compute new value	*Compute new values*	
	Set $A = €0$	$A = €0$
	Set $B = €900$	$B = €900$
	Commit	
Set $A = €350$		$A = €350$
Commit		$B = €900$

$\to$ Concurrent execution of transactions might lead to data inconsistencies!

Outline

1. Database Concurrency Control (101)
   - Serializability
   - Isolation Levels
   - Robustness
2. Robustness for Transactions
3. Robustness for Transaction Templates
4. Conclusions

Serializability: holy grail for data consistency

Executions that leave the data in a consistent state

Definition

A schedule is serializable if its outcome is equivalent to that of a serial schedule (with the same transactions).

Rationale: if each transaction is correct by itself, then a schedule that comprises any serial execution of these transactions is correct.

Serializability: holy grail for data consistency

Executions that leave the data in a consistent state

Definition

A schedule is serializable if its outcome is equivalent to that of a serial schedule (with the same transactions).

Rationale: if each transaction is correct by itself, then a schedule that comprises any serial execution of these transactions is correct.

Transaction 1 *Withdraw €50 from account A*	**Transaction 2** *Transfer €400 from account A to B*	Accounts
		$A = €400$ $B = €500$
Get balance $A \rightarrow €400$		
	Get balance $A \rightarrow €400$	
	Get balance $B \rightarrow €500$	
Compute new value	*Compute new values*	
	Set $A = €0$	$A = €0$
	Set $B = €900$	$B = €900$
	Commit	
Set $A = €350$		$A = €350$
Commit		$B = €900$

Outcome is not equivalent to

- $T_1; T_2$: $A = -50$, $B = 900$, or,
- $T_2; T_1$: $A = -50$, $B = 900$.

Concurrency control methods that guarantee serializability

Pessimistic concurrency control

Concurrent transactions can be delayed through locking.

Two-phase locking (2PL)

Concurrency control methods that guarantee serializability

Pessimistic concurrency control

Concurrent transactions can be delayed through locking.

Two-phase locking (2PL)

- Regulate access through shared (read) and exclusive (write) locks.

Concurrency control methods that guarantee serializability

Pessimistic concurrency control

Concurrent transactions can be delayed through locking.

Two-phase locking (2PL)

- Regulate access through shared (read) and exclusive (write) locks.
    - R-locks on the same object do not conflict, other combinations do

Concurrency control methods that guarantee serializability

Pessimistic concurrency control

Concurrent transactions can be delayed through locking.

Two-phase locking (2PL)

- Regulate access through shared (read) and exclusive (write) locks.
  - R-locks on the same object do not conflict, other combinations do
  - Before an operation a corresponding lock needs to be acquired. If there is a conflict the acquiring party needs to wait.

Concurrency control methods that guarantee serializability

Pessimistic concurrency control

Concurrent transactions can be delayed through locking.

Two-phase locking (2PL)

- Regulate access through shared (read) and exclusive (write) locks.
  - R-locks on the same object do not conflict, other combinations do
  - Before an operation a corresponding lock needs to be acquired. If there is a conflict the acquiring party needs to wait.
- Two phases:
  - Growing: lock acquiring phase, no locks are released
  - Shrinking: lock releasing phase, no locks are acquired

Two phase locking

Transaction 1 *Withdraw €50 from account A*	**Transaction 2** *Transfer €400 from account A to B*	Accounts
		$A = €400$ $B = €500$
R-lock(A). Read(A)		

Two phase locking

Transaction 1 *Withdraw €50 from account A*	**Transaction 2** *Transfer €400 from account A to B*	Accounts
		$A = €400$ $B = €500$
R-lock(A). Read(A)		
	R-lock(A). Read(A)	
	R-lock(B). Read(B)	

Two phase locking

Transaction 1 *Withdraw €50 from account A*	**Transaction 2** *Transfer €400 from account A to B*	Accounts
		$A = €400$ $B = €500$
R-lock(A). Read(A)		
	R-lock(A). Read(A) R-lock(B). Read(B) *Compute new values*	

Two phase locking

Transaction 1 *Withdraw €50 from account A*	**Transaction 2** *Transfer €400 from account A to B*	Accounts
		$A = €400$ $B = €500$
R-lock(A). Read(A)		
	R-lock(A). Read(A)	
	R-lock(B). Read(B)	
	Compute new values	
	W-lock(A). *Denied*	

Two phase locking

Transaction 1 *Withdraw €50 from account A*	**Transaction 2** *Transfer €400 from account A to B*	Accounts
		$A = €400$ $B = €500$
R-lock(A). Read(A)		
	R-lock(A). Read(A)	
	R-lock(B). Read(B)	
	Compute new values	
Compute new value	W-lock(A). *Denied*	
W-lock(A). *Denied*		

DEADLOCK

Two phase locking

Transaction 1 *Withdraw €50 from account A*	**Transaction 2** *Transfer €400 from account A to B*	Accounts
		$A = €400$ $B = €500$
R-lock(A). Read(A)		
	R-lock(A). Read(A)	
	R-lock(B). Read(B)	
	Compute new values	
Compute new value	W-lock(A). *Denied*	
W-lock(A). *Denied*		

DEADLOCK

Guarantees serializability, but has a negative effect on throughput

- Waiting on release of locks
- Aborts to resolve deadlocks

Multiversion concurrency control (MVCC)

Multiversion

- DBMS maintains multiple versions of an object
  - e.g., achieved through timestamps
- When reading an object
  - no longer blocked by concurrent writer
  - an earlier version can be supplied

Concurrency control methods that guarantee serializability

Optimistic concurrency control

Serializable snapshot isolation

Concurrency control methods that guarantee serializability

Optimistic concurrency control

Serializable snapshot isolation

- Crux:
  - Transaction takes a snapshot of the data at start time and makes tentative changes on the snapshot

Concurrency control methods that guarantee serializability

Optimistic concurrency control

Serializable snapshot isolation

- Crux:
  - Transaction takes a snapshot of the data at start time and makes tentative changes on the snapshot
  - **Snapshot Isolation**: at commit time, check whether concurrent transactions have modified objects that the current transaction wants to install in the database, abort if so (*first committer wins*).

Concurrency control methods that guarantee serializability

Optimistic concurrency control

Serializable snapshot isolation

- Crux:
  - Transaction takes a snapshot of the data at start time and makes tentative changes on the snapshot
  - **Snapshot Isolation**: at commit time, check whether concurrent transactions have modified objects that the current transaction wants to install in the database, abort if so (*first committer wins*).
  - **Serializable SI:** additional *dangerous structure* check

Concurrency control methods that guarantee serializability

Optimistic concurrency control

Serializable snapshot isolation

- Crux:
  - Transaction takes a snapshot of the data at start time and makes tentative changes on the snapshot
  - **Snapshot Isolation**: at commit time, check whether concurrent transactions have modified objects that the current transaction wants to install in the database, abort if so (*first committer wins*).
  - **Serializable SI:** additional *dangerous structure* check
- Mantra: readers do not block writers (and vice-versa), but writers still block writers.

Concurrency control methods that guarantee serializability

Optimistic concurrency control

Serializable snapshot isolation

- Crux:
  - Transaction takes a snapshot of the data at start time and makes tentative changes on the snapshot
  - **Snapshot Isolation**: at commit time, check whether concurrent transactions have modified objects that the current transaction wants to install in the database, abort if so (*first committer wins*).
  - **Serializable SI:** additional *dangerous structure* check
- Mantra: readers do not block writers (and vice-versa), but writers still block writers.
- **Guarantees serializability**, but has a negative effect on throughput:
  - performing checks,
  - possible aborts due to conflicts.

(Serializable) Snapshot Isolation

Transaction 1 *Withdraw €50 from account A*	**Transaction 2** *Transfer €400 from account A to B*	Accounts
		$A = €400$ $B = €500$
Take snapshot Get balance $A \to €400$		

(Serializable) Snapshot Isolation

Transaction 1 *Withdraw €50 from account A*	**Transaction 2** *Transfer €400 from account A to B*	Accounts
		$A = €400$ $B = €500$
Take snapshot Get balance $A \rightarrow €400$		
	Take snapshot Get balance $A \rightarrow €400$ Get balance $B \rightarrow €500$ Set $A = €0$, Set $B = €900$ Commit	
		$A = €0$, $B = €900$

(Serializable) Snapshot Isolation

Transaction 1 *Withdraw €50 from account A*	**Transaction 2** *Transfer €400 from account A to B*	Accounts
		$A = €400$ $B = €500$
Take snapshot Get balance $A \rightarrow €400$		
	Take snapshot Get balance $A \rightarrow €400$ Get balance $B \rightarrow €500$ Set $A = €0$, Set $B = €900$ Commit	
		$A = €0$, $B = €900$
Set $A = €350$ Commit $\rightarrow$ ABORT		

Outline

Isolation level defines a superset of serializable schedules

Trading consistency for increased throughput

Isolation level defines a superset of serializable schedules

Trading consistency for increased throughput

Postgress

1. READ COMMITTED:
   - read last committed version (no locking)
   - a write statement acquires W-lock (released at commit)
   - deadlock → aborts
2. REPEATABLE READ (aka SNAPSHOT ISOLATION)
3. SERIALIZABLE (aka SERIALIZABLE SNAPSHOT ISOLATION)

`https://www.postgresql.org/docs/current/transaction-iso.html`

Schedule for bank example is allowed under RC

but not under SI

Transaction 1 *Withdraw €50 from account A*	**Transaction 2** *Transfer €400 from account A to B*	Accounts
		$A = €400$ $B = €500$
Get balance $A \to €400$		
	Get balance $A \to €400$	
	Get balance $B \to €500$	
Compute new value	*Compute new values*	
	W-lock(A) Set $A = €0$	$A = €0$
	W-lock(B) Set $B = €900$	$B = €900$
	Commit. **Release locks**	
W-lock(A)		
Set $A = €350$		$A = €350$
Commit		$B = €900$

Non-serializable bank example allowed under SI

Allowed under SI

- Account $A = €600$; Account $B = €700$.
- T_A : Withdraw €500 from account A if sum $A + B > €1000$
- T_B : Withdraw €500 from account B if sum $A + B > €1000$

Non-serializable bank example allowed under SI

Allowed under SI

- Account $A = €600$; Account $B = €700$.
- T_A : Withdraw €500 from account A if sum $A + B > €1000$
- T_B : Withdraw €500 from account B if sum $A + B > €1000$
- Serial execution:
  - $T_A; T_B$: $A = €100$; $B = €700$
  - $T_B; T_A$: $A = €600$; $B = €200$

Non-serializable bank example allowed under SI

Allowed under SI

- Account $A =$ €600; Account $B =$ €700.
- T_A : Withdraw €500 from account A if sum $A + B >$ €1000
- T_B : Withdraw €500 from account B if sum $A + B >$ €1000
- Serial execution:
  - $T_A; T_B$: $A =$ €100; $B =$ €700
  - $T_B; T_A$: $A =$ €600; $B =$ €200
- Concurrent execution under SI: $A =$ €100; $B =$ €200

What about a free lunch?

Under which conditions, do isolation levels weaker than serializability, provide the same guarantees as serializability?

Outline

Robustness

Assume an isolation level $\mathcal{I}$ is chosen for a given workload $\mathcal{T}$:

Workload $\mathcal{T}$

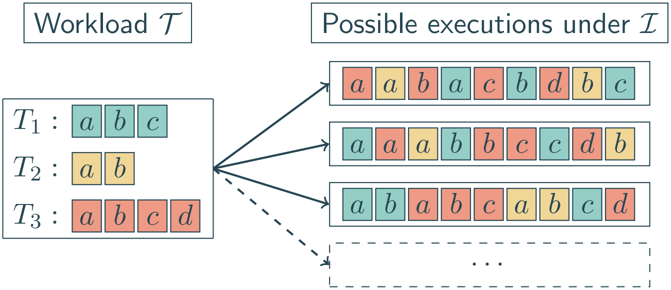
T_1 : a b c

T_2 : a b

T_3 : a b c d

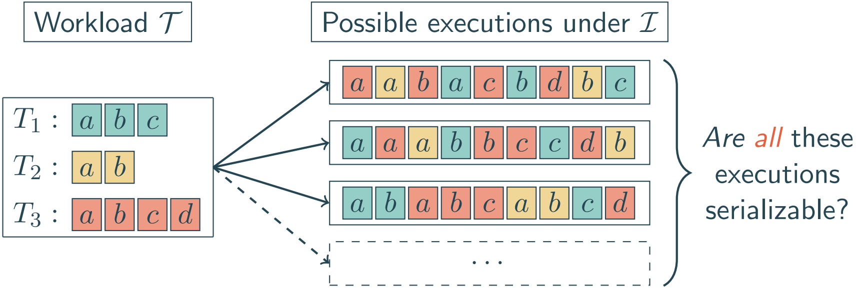
Robustness

Assume an isolation level $\mathcal{I}$ is chosen for a given workload $\mathcal{T}$:



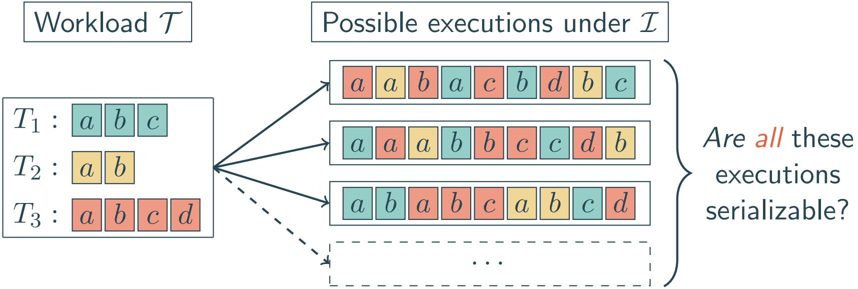
Robustness

Assume an isolation level $\mathcal{I}$ is chosen for a given workload $\mathcal{T}$:



Robustness

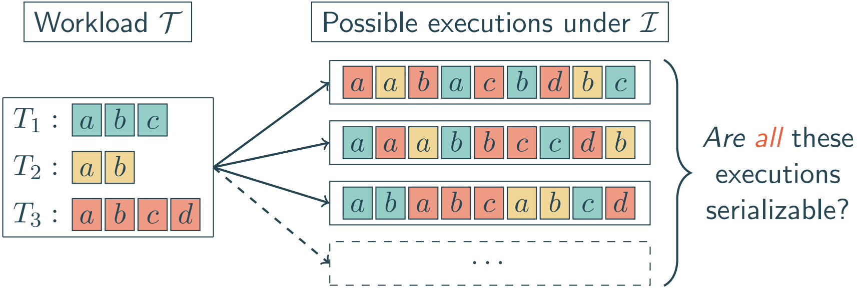
Assume an isolation level $\mathcal{I}$ is chosen for a given workload $\mathcal{T}$:



$\Rightarrow$ Workload $\mathcal{T}$ is robust against isolation level $\mathcal{I}$.

Robustness

Assume an isolation level $\mathcal{I}$ is chosen for a given workload $\mathcal{T}$:



$\Rightarrow$ Workload $\mathcal{T}$ is robust against isolation level $\mathcal{I}$.

Robustness

- guarantees serializability under a lower isolation level
- expected higher throughput

TPC-C is robust against SNAPSHOT ISOLATION

[Fekete et al., 2005]

TPC-C

- is a complex benchmark dealing with most aspects of ordering, paying for, and delivering of goods from warehouses.
- consists of nine tables and five transaction programs.

Transaction Programs:

- NewOrder
- StockLevel
- Payment
- OrderStatus
- Delivery

Robustness

Every workload resulting from instantiations of the transaction programs is serializable when executed under SNAPSHOT ISOLATION.

Work on robustness

[Fekete et al., 2005] [Fekete, 2005] [Alomari et al., 2008] [Alomari and Fekete, 2015] [Bernardi and Gotsman, 2016] [Cerone et al., 2017] [Cerone and Gotsman, 2018] [Beillahi et al., 2019a] [Beillahi et al., 2019b]

Research on robustness...

- ...mostly focused on higher isolation levels (e.g. variations of Snapshot Isolation);
- ...mostly focused on sufficient conditions to guarantee robustness.

Work on robustness

[Fekete et al., 2005] [Fekete, 2005] [Alomari et al., 2008] [Alomari and Fekete, 2015] [Bernardi and Gotsman, 2016] [Cerone et al., 2017] [Cerone and Gotsman, 2018] [Beillahi et al., 2019a] [Beillahi et al., 2019b]

Research on robustness. . .

- . . . mostly focused on higher isolation levels (e.g. variations of Snapshot Isolation);
- . . . mostly focused on sufficient conditions to guarantee robustness.

However, lower isolation levels are used in practice as well:

- RC is the default isolation level in certain databases (e.g. Postgres) [Bailis et al., 2013].
- Focus on RC (and SI) in the rest of this talk [Ketsman et al., 2020, Vandevoort et al., 2021, Vandevoort et al., 2022]

Outline

Transactions

Set $\mathcal{T}$ of transactions

$$T_1: \quad \mathtt{R}_1[\mathtt{x}]\,\mathtt{W}_1[\mathtt{y}]\,\mathtt{C}_1$$
$$T_2: \quad \mathtt{R}_2[\mathtt{z}]\,\mathtt{W}_2[\mathtt{x}]\,\mathtt{W}_2[\mathtt{z}]\,\mathtt{C}_2$$
$$T_3: \quad \mathtt{R}_3[\mathtt{y}]\,\mathtt{W}_3[\mathtt{z}]\,\mathtt{C}_3$$

- assumption:
  - subscripting operations with the index number of the transaction
  - transaction reads and writes at most once the same object
- simplistic model

Schedules

Schedule (history) s over $\mathcal{T}$

(T_1)	$R_1[x_0]$		$W_1[y]C_1$			
(T_2)		$R_2[z_0]$		$W_2[x]$		$W_2[z]C_2$
(T_3)					$R_3[y_1]W_3[z]C_3$	

- total order $<_s$ on operations in $\mathcal{T}$
- $<_s$ is consistent with ordering of the operations in transactions in $\mathcal{T}$

Schedules

Schedule (history) s over $\mathcal{T}$

(T_1)	$\mathtt{R}_1[\mathtt{x}_0]$		$\mathtt{W}_1[\mathtt{y}]\mathtt{C}_1$			
(T_2)		$\mathtt{R}_2[\mathtt{z}_0]$		$\mathtt{W}_2[\mathtt{x}]$		$\mathtt{W}_2[\mathtt{z}]\mathtt{C}_2$
(T_3)					$\mathtt{R}_3[\mathtt{y}_1]\mathtt{W}_3[\mathtt{z}]\mathtt{C}_3$	

- total order $<_s$ on operations in $\mathcal{T}$
- $<_s$ is consistent with ordering of the operations in transactions in $\mathcal{T}$
- maps every read operation to a write operation

Schedules

Schedule (history) s over $\mathcal{T}$

(T_1)	$\mathtt{R}_1[\mathtt{x}_0]$		$\mathtt{W}_1[\mathtt{y}]\mathtt{C}_1$				
(T_2)		$\mathtt{R}_2[\mathtt{z}_0]$		$\mathtt{W}_2[\mathtt{x}]$			$\mathtt{W}_2[\mathtt{z}]\mathtt{C}_2$
(T_3)					$\mathtt{R}_3[\mathtt{y}_0]\mathtt{W}_3[\mathtt{z}]\mathtt{C}_3$		

- total order $<_s$ on operations in $\mathcal{T}$
- $<_s$ is consistent with ordering of the operations in transactions in $\mathcal{T}$
- maps every read operation to a write operation
- initial value x_0, y_0, z_0 for each object x, y, z

Towards serializability

Definition

A schedule is serializable iff it is conflict-equivalent to a single-version serial schedule.

- Serial: schedule that executes transactions in a serial fashion.
- Single-version: only one installed version at the time.
- Several flavors of schedule equivalence: focus on conflict-equivalence.

Towards serializability

Definition

A schedule is serializable iff it is conflict-equivalent to a single-version serial schedule.

- Serial: schedule that executes transactions in a serial fashion.
- Single-version: only one installed version at the time.
- Several flavors of schedule equivalence: focus on conflict-equivalence.

Definition

Two operations are **conflicting** if they are on the same object, and at least one of them is a write.

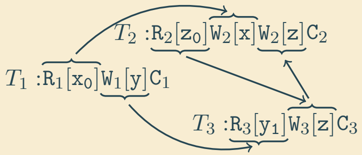
From conflicts to dependencies relative to a schedule s

- $T \to T'$ iff T accesses x, later T' accesses x, and the accesses conflict
- induces a **relative ordering** of transactions in a serial schedule that preserves the order of conflicts

Schedule s

(T_1)	$R_1[x_0]$		$W_1[y]C_1$			
(T_2)		$R_2[z_0]$		$W_2[x]$		$W_2[z]C_2$
(T_3)					$R_3[y_1]W_3[z]C_3$	

Conflict Graph $CG(s)$



$(T : b) \to (T' : a)$:

- write-write dependency
- write-read dependency
- read-write (anti-)dependency

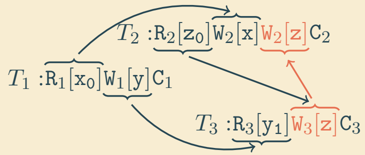
From conflicts to dependencies relative to a schedule s

- $T \to T'$ iff T accesses x, later T' accesses x, and the accesses conflict
- induces a **relative ordering** of transactions in a serial schedule that preserves the order of conflicts

Schedule s

(T_1)	$R_1[x_0]$		$W_1[y]C_1$			
(T_2)		$R_2[z_0]$		$W_2[x]$		$W_2[z]C_2$
(T_3)					$R_3[y_1]W_3[z]C_3$	

Conflict Graph $CG(s)$



$(T : b) \to (T' : a)$:

- write-write dependency: b is ww-conflicting with a and $b <_s a$
- write-read dependency
- read-write (anti-)dependency

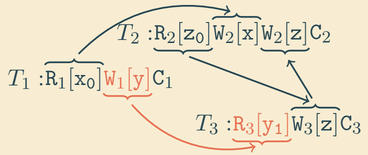
From conflicts to dependencies relative to a schedule s

- $T \to T'$ iff T accesses x, later T' accesses x, and the accesses conflict
- induces a **relative ordering** of transactions in a serial schedule that preserves the order of conflicts

Schedule s

(T_1)	$R_1[x_0]$		$W_1[y]C_1$			
(T_2)		$R_2[z_0]$		$W_2[x]$		$W_2[z]C_2$
(T_3)					$R_3[y_1]W_3[z]C_3$	

Conflict Graph $CG(s)$



$(T : b) \to (T' : a)$:

- write-write dependency
- write-read dependency: b is wr-conflicting with a, and a reads the version written by b (or later)
- read-write (anti-)dependency

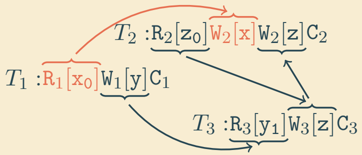
From conflicts to dependencies relative to a schedule s

- $T \to T'$ iff T accesses x, later T' accesses x, and the accesses conflict
- induces a **relative ordering** of transactions in a serial schedule that preserves the order of conflicts

Schedule s

(T_1)	$R_1[x_0]$		$W_1[y]C_1$			
(T_2)		$R_2[z_0]$		$W_2[x]$		$W_2[z]C_2$
(T_3)					$R_3[y_1]W_3[z]C_3$	

Conflict Graph $CG(s)$



$(T : b) \to (T' : a)$:

- write-write dependency
- write-read dependency
- read-write (anti-)dependency: b is rw-conflicting with a, and b reads a version installed before the version written by a

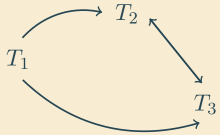
From conflicts to dependencies relative to a schedule s

- $T \to T'$ iff T accesses x, later T' accesses x, and the accesses conflict
- induces a **relative ordering** of transactions in a serial schedule that preserves the order of conflicts

Schedule s

(T_1)	$R_1[x_0]$		$W_1[y]C_1$				
(T_2)		$R_2[z_0]$		$W_2[x]$			$W_2[z]C_2$
(T_3)						$R_3[y_1]W_3[z]C_3$	

Conflict Graph $CG(s)$



$(T : b) \to (T' : a)$:

- write-write dependency
- write-read dependency
- read-write (anti-)dependency

Conflict serializability

Definition

Two schedules s and s' are **conflict-equivalent** iff $CG(s) = CG(s')$.

Conflict serializability

Definition

Two schedules s and s' are **conflict-equivalent** iff $CG(s) = CG(s')$.

Definition

A schedule s over $\mathcal{T}$ is **(conflict) serializable** iff it is conflict-equivalent to a single-version serial schedule.

Conflict serializability

Definition

Two schedules s and s' are **conflict-equivalent** iff $CG(s) = CG(s')$.

Definition

A schedule s over $\mathcal{T}$ is **(conflict) serializable** iff it is conflict-equivalent to a single-version serial schedule.

Theorem (e.g.,[Papadimitriou, 1986])

A schedule s over $\mathcal{T}$ is conflict serializable iff $CG(s)$ is acyclic.

Robustness against an isolation level $\mathcal{I}$

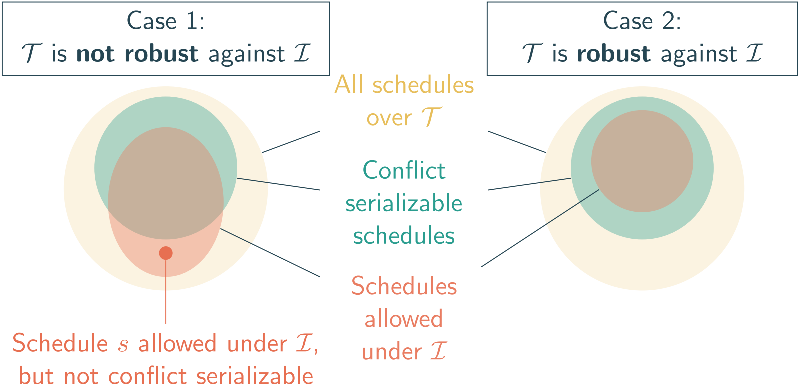
Definition

A set of transactions $\mathcal{T}$ is robust against $\mathcal{I}$ iff
every schedule for $\mathcal{T}$ that is allowed under $\mathcal{I}$ is serializable.

Robustness against an isolation level $\mathcal{I}$

Definition

A set of transactions $\mathcal{T}$ is robust against $\mathcal{I}$ iff
every schedule for $\mathcal{T}$ that is allowed under $\mathcal{I}$ is serializable.



Outline

1. Database Concurrency Control (101)
2. Robustness for Transactions
   - Snapshot Isolation
   - Multiversion Read Committed
3. Robustness for Transaction Templates
4. Conclusions

Snapshot isolation

- rset(T): set of objects *read* in transaction T
- wset(T): set of *modified* objects in transaction T

Snapshot Isolation (SI)

A schedule is allowed under SI iff

- every read operation refers to the last committed version *before the start of the current transaction.*
- First Committer Wins: a transaction T can not commit if wset$(T) \cap$ wset$(T') \neq \emptyset$ for any transaction T' concurrent with T.

Snapshot Isolation: properties of cycles in graphs

Snapshot Isolation: properties of cycles in graphs

For s a schedule allowed under SI:

- $T \rightarrow^{ww} T'$: T finishes before T' starts

Snapshot Isolation: properties of cycles in graphs

For s a schedule allowed under SI:

- $T \rightarrow^{ww} T'$: T finishes before T' starts
- $T \rightarrow^{wr} T'$: T finishes before T' starts

Snapshot Isolation: properties of cycles in graphs

For s a schedule allowed under SI:

- $T \rightarrow^{ww} T'$: T finishes before T' starts
- $T \rightarrow^{wr} T'$: T finishes before T' starts

Observation

There can be not be a cycle in the CG of a schedule in SI containing

only ww- and wr-dependencies.

Snapshot Isolation: properties of cycles in graphs

For s a schedule allowed under SI:

- $T \rightarrow^{ww} T'$: T finishes before T' starts
- $T \rightarrow^{wr} T'$: T finishes before T' starts

Observation

There can be not be a cycle in the CG of a schedule in SI containing

only ww- and wr-dependencies.

Indeed, a cycle

$$T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$$

implies that

T_1 finishes before T_1 starts.

Snapshot Isolation: properties of cycles in graphs

A cycle in $CG(s)$ must contain at least one rw-dependency.

Theorem ([Fekete, 2005])

If s in SI is not serializable, then $CG(s)$ contains a chord-free cycle

$$T \rightarrow \cdots \rightarrow T_a \rightarrow^{rw} T_b \rightarrow^{rw} T_c \rightarrow \cdots \rightarrow T$$

where wset$(T_a) \cap$ wset$(T_b) = \emptyset$ and wset$(T_b) \cap$ wset$(T_c) = \emptyset$.

Robustness against SI

Interference Graph $IG(\mathcal{T})$ (static dependency graph)

- Superposition of dependencies for all possible schedules
- Nodes in $IG(\mathcal{T})$ are transactions in $\mathcal{T}$.
- Edges indicate interference between transactions:
  1. $T_1 \rightarrow^e T_2$ if
     - rset$(T_1) \cap$ wset$(T_2) \neq \emptyset$ and wset$(T_1) \cap$ wset$(T_2) = \emptyset$
     - exposed (vulnerable) edge
  2. else, $T_1 \rightarrow^p T_2$ if
     - at least one transaction writes to a commonly accessed attribute
     - protected (non-vulnerable) edge

Robustness against SI

Interference Graph $IG(\mathcal{T})$ (static dependency graph)

- Superposition of dependencies for all possible schedules
- Nodes in $IG(\mathcal{T})$ are transactions in $\mathcal{T}$.
- Edges indicate interference between transactions:
  1. $T_1 \rightarrow^e T_2$ if
     - rset$(T_1) \cap$ wset$(T_2) \neq \emptyset$ and wset$(T_1) \cap$ wset$(T_2) = \emptyset$
     - exposed (vulnerable) edge
  2. else, $T_1 \rightarrow^p T_2$ if
     - at least one transaction writes to a commonly accessed attribute
     - protected (non-vulnerable) edge

Property

Let s be a schedule for $\mathcal{T}$ allowed under SI,

a cycle in a $CG(s)$ implies a cycle in $IG(\mathcal{T})$.

Simple structure of counter example schedule

Theorem ([Fekete, 2005])

A set of transactions $\mathcal{T}$ is not robust against SI iff

$$IG(\mathcal{T}) \text{ contains a chord-free cycle } T \cdots \rightarrow T_a \rightarrow^e T_b \rightarrow^e T_c \rightarrow \cdots T$$

Simple structure of counter example schedule

Theorem ([Fekete, 2005])

A set of transactions $\mathcal{T}$ is not robust against SI iff
$IG(\mathcal{T})$ *contains a chord-free cycle* $T \cdots \to T_a \to^e T_b \to^e T_c \to \cdots T$

Counter example split schedule s

start(T_b) $\quad$ T_b

T_c

$\cdots T \cdots$

T_a

Simple structure of counter example schedule

Theorem ([Fekete, 2005])

A set of transactions $\mathcal{T}$ is not robust against SI iff
$IG(\mathcal{T})$ contains a chord-free cycle $T \cdots \to T_a \to^e T_b \to^e T_c \to \cdots T$

Counter example split schedule s

start(T_b) T_b
T_c
$\cdots T \cdots$
T_a

Requirements

- T_b does not have a ww- or wr-dependency with any of the other transactions
- $T_b \to^{rw} T_c$
- $T_a \to^{rw} T_b$
- $T_c \to \cdots \to T \to \cdots \to T_a$

Outline

1. Database Concurrency Control (101)
2. Robustness for Transactions
   - Snapshot Isolation
   - Multiversion Read Committed
3. Robustness for Transaction Templates
4. Conclusions

Multiversion read committed

Dirty writes

A schedule exhibits a dirty write if the following occurs:

$$\begin{array}{llll} (T_i) & \dots \mathtt{W}_i[\mathtt{x}] \dots & & \dots \mathtt{C}_i \\ (T_j) & & \dots \mathtt{W}_j[\mathtt{x}] \dots & \end{array}$$

Multiversion read committed

Dirty writes

A schedule exhibits a dirty write if the following occurs:

$$\begin{array}{llll} (T_i) & \ldots \mathtt{W}_i[\mathtt{x}] \ldots & & \ldots \mathtt{C}_i \\ (T_j) & & \ldots \mathtt{W}_j[\mathtt{x}] \ldots & \end{array}$$

Multiversion Read Committed (MVRC)

A schedule is allowed under MVRC iff

- it does not exhibit a dirty write, and
- every read operation refers to the most recent committed version

Robustness: SI vs MVRC

We can view an isolation level $\mathcal{I}$ as a set of allowed schedules.

Observation

Let $\mathcal{I} \subseteq \mathcal{J}$ and $\mathcal{T}$ as set of transactions:

non-robustness of $\mathcal{T}$ against $\mathcal{I}$ implies
non-robustness of $\mathcal{T}$ against $\mathcal{J}$.

Robustness: SI vs MVRC

We can view an isolation level $\mathcal{I}$ as a set of allowed schedules.

Observation

Let $\mathcal{I} \subseteq \mathcal{J}$ and $\mathcal{T}$ as set of transactions:

non-robustness of $\mathcal{T}$ against $\mathcal{I}$ implies
non-robustness of $\mathcal{T}$ against $\mathcal{J}$.

Because of timing of snapshots:

- SI $\not\subseteq$ MVRC, and
- MVRC $\not\subseteq$ SI

Example

$$\begin{array}{llll} T_1: & & W_1[y]\, C_1 & \\ T_2: & R_2[x_0] & & R_2[y]\, C_2 \end{array}$$

Multiversion read committed

For s a schedule allowed under MVRC:

- $T \rightarrow^{ww} T'$: can be concurrent but T commits before T'

Multiversion read committed

For s a schedule allowed under MVRC:

- $T \rightarrow^{ww} T'$: can be concurrent but T commits before T'
- $T \rightarrow^{wr} T'$: can be concurrent but T commits before T'

Multiversion read committed

For s a schedule allowed under MVRC:

- $T \rightarrow^{ww} T'$: can be concurrent but T commits before T'
- $T \rightarrow^{wr} T'$: can be concurrent but T commits before T'

Observation

There can not be a cycle in the CG of a schedule under MVRC containing

only ww- and wr-dependencies.

Multiversion read committed

For s a schedule allowed under MVRC:

- $T \rightarrow^{ww} T'$: can be concurrent but T commits before T'
- $T \rightarrow^{wr} T'$: can be concurrent but T commits before T'

Observation

There can not be a cycle in the CG of a schedule under MVRC containing

only ww- and wr-dependencies.

Indeed, a cycle

$$T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$$

implies that

T_1 commits before T_1 commits.

Robustness against MVRC

Theorem ([Vandevoort et al., 2021])

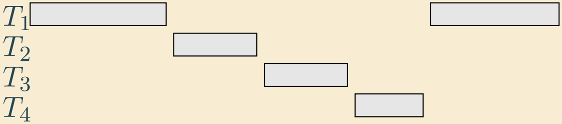
A set of transactions $\mathcal{T}$ is not robust against MVRC iff there exists a counter example **multiversion split schedule**.

Robustness against MVRC

Theorem ([Vandevoort et al., 2021])

A set of transactions $\mathcal{T}$ is not robust against MVRC iff there exists a counter example **multiversion split schedule***.*

Multiversion split schedule

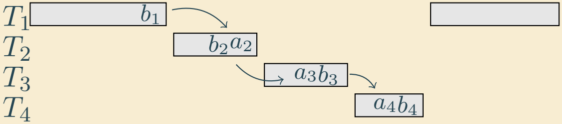


Robustness against MVRC

Theorem ([Vandevoort et al., 2021])

A set of transactions $\mathcal{T}$ is not robust against MVRC iff there exists a counter example **multiversion split schedule**.

Multiversion split schedule



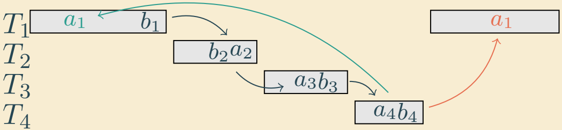
- b_1 is **rw-conflicting** with a_2, b_i is conflicting with a_i, b_4 is conflicting with a_1

Robustness against MVRC

Theorem ([Vandevoort et al., 2021])

A set of transactions $\mathcal{T}$ is not robust against MVRC iff there exists a counter example **multiversion split schedule**.

Multiversion split schedule



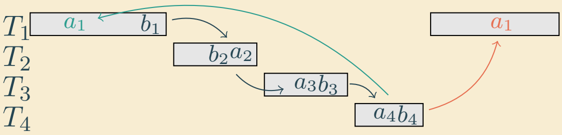
- b_1 is **rw-conflicting** with a_2, b_i is conflicting with a_i, b_4 is conflicting with a_1
- $b_1 <_{T_1} a_1$ or b_4 is rw-conflicting with a_1; and,

Robustness against MVRC

Theorem ([Vandevoort et al., 2021])

A set of transactions $\mathcal{T}$ is not robust against MVRC iff there exists a counter example **multiversion split schedule**.

Multiversion split schedule



- b_1 is **rw-conflicting** with a_2, b_i is conflicting with a_i, b_4 is conflicting with a_1
- $b_1 <_{T_1} a_1$ or b_4 is rw-conflicting with a_1; and,
- there is no write operation in $\text{prefix}_{b_1}(T_1)$ ww-conflicting with a write operation in any of the transactions T_2, T_3, T_4;

Robustness: SI versus MVRC (revisited)

Observation: non-robustness against SI implies non-robustness against MVRC (but not vice versa)

Counter example for SI is also one for MVRC

start(T_b) T_b

T_c

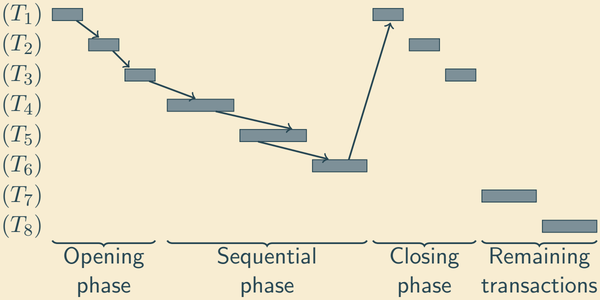
$\cdots T \cdots$

T_a

- $T_b \rightarrow^{rw} T_c$
- T_b does not have a ww-dependency with any of the other transactions
- $T_a \rightarrow^{rw} T_b$

Single-version read committed with locks

Multi-Split Schedule



Theorem ([Ketsman et al., 2020])

A set $\mathcal{T}$ of transactions is not robust against RC iff there is a multi-split schedule over $\mathcal{T}$ allowed under Read Committed.

Robustness problem is coNP-complete.

Summary

Sound and complete algorithms

- Snapshot Isolation [Fekete, 2005]
- Single-version read committed and read uncommitted [Ketsman et al., 2020]
- Multiversion read committed [Vandevoort et al., 2021]

Characterizations in terms of

- cycles of a specific form
- counter example schedules of a specific form

But what about real world transactions?

Real world transactions

- Set $\mathcal{T}$ of transactions is rarely known in advance
- Flow-of-control, inserts, deletes, predicate reads

But what about real world transactions?

Real world transactions

- Set $\mathcal{T}$ of transactions is rarely known in advance
- Flow-of-control, inserts, deletes, predicate reads

Approximate approach [Fekete et al., 2005]

- Construct a super approximation of the interference graph

But what about real world transactions?

Real world transactions

- Set $\mathcal{T}$ of transactions is rarely known in advance
- Flow-of-control, inserts, deletes, predicate reads

Approximate approach [Fekete et al., 2005]

- Construct a super approximation of the interference graph
- If the interference graph does **not** contain a forbidden cycle
  - **then** conclude that the considered setting is robust

But what about real world transactions?

Real world transactions

- Set $\mathcal{T}$ of transactions is rarely known in advance
- Flow-of-control, inserts, deletes, predicate reads

Approximate approach [Fekete et al., 2005]

- Construct a super approximation of the interference graph
- If the interference graph does **not** contain a forbidden cycle
  - **then** conclude that the considered setting is robust
  - **otherwise**, non-robustness can not be concluded

Our approach

- Focus on sets of transactions that are generated through a fixed set of transaction programs
- Provide an adequate formalization that ensures soundness and completeness

Outline

1. Database Concurrency Control (101)
2. Robustness for Transactions
3. Robustness for Transaction Templates
   - Transaction Templates
   - Functional Constraints
   - Limitations
4. Conclusions

Outline

SmallBank benchmark [Alomari et al., 2008]

Database Schema

Account (Name, CustomerID)
Savings (CustomerID, Balance)
Checking (CustomerID, Balance)

Programs

- **Balance**: return total balance for a given customer.
- **DepositChecking**: deposit a given amount on the checking account of a given customer.
- **TransactSavings**: deposit or withdraw a given amount on the savings account of a given customer.
- **Amalgamate**: transfer all funds of one given customer to the checking account of a second given customer.
- **WriteCheck**: write a check of a given amount against a given customer, penalizing if overdrawing.

Transaction templates

Transaction Templates

A transaction template is a sequence of **read** (R), **write** (W) and **atomic update** (U) operations over **typed variables**, where each operation specifies the **list of attributes** that is being read/overwritten.

Example: SmallBank benchmark

WriteCheck:

R[X : Account{Name, CustID}]
R[Y : Savings{CustID, Bal}]
R[Z : Checking{CustID, Bal}]
U[Z : Checking{CustID, Bal}{Bal}]

DepositChecking:

R[X : Account{Name, CustID}]
U[Z : Checking{CustID, Bal}{Bal}]

Atomic update (U) operations combine a read (R) and write (W) operation in one *atomic* operation, that cannot be interleaved by other operations.

Transaction templates and schedules

By assigning tuples to variables, we can instantiate transactions.

Example: SmallBank benchmark

WriteCheck:

$\mathtt{R}[\mathtt{X} : \text{Account}\{\text{Name, CustID}\}]$
$\mathtt{R}[\mathtt{Y} : \text{Savings}\{\text{CustID, Bal}\}]$
$\mathtt{R}[\mathtt{Z} : \text{Checking}\{\text{CustID, Bal}\}]$
$\mathtt{U}[\mathtt{Z} : \text{Checking}\{\text{CustID, Bal}\}\{\text{Bal}\}]$

DepositChecking:

$\mathtt{R}[\mathtt{X} : \text{Account}\{\text{Name, CustID}\}]$
$\mathtt{U}[\mathtt{Z} : \text{Checking}\{\text{CustID, Bal}\}\{\text{Bal}\}]$

Schedule over {WriteCheck, DepositChecking}

WC_1 :	$\mathtt{R}[\mathtt{a}_0]\,\mathtt{R}[\mathtt{s}_0]$			$\mathtt{R}[\mathtt{c}_0]$		$\mathtt{U}[\mathtt{c}_2]\,\mathtt{C}$	$(\mathtt{X} \mapsto \mathtt{a}, \mathtt{Y} \mapsto \mathtt{s}, \mathtt{Z} \mapsto \mathtt{c})$
DC_2 :		$\mathtt{R}[\mathtt{a}_0]$			$\mathtt{U}[\mathtt{c}_0]\,\mathtt{C}$		$(\mathtt{X} \mapsto \mathtt{a}, \mathtt{Z} \mapsto \mathtt{c})$
DC_3 :			$\mathtt{R}[\mathtt{a}'_0]\,\mathtt{U}[\mathtt{c}'_0]\,\mathtt{C}$				$(\mathtt{X} \mapsto \mathtt{a}', \mathtt{Z} \mapsto \mathtt{c}')$

Deciding robustness against RC

Key insight:

If a workload is not robust against MVRC, then a counterexample multiversion split schedule exists with at most **3 different tuples of each type**.

Theorem [Vandevoort et al., 2021]

Deciding robustness against MVRC for a set of transaction templates is in PTIME.

Detecting robustness against RC

Maximal robust subsets by analysis setting for **SmallBank**:

	Robust subsets	[Alomari and Fekete, 2015]
Only R & W	{Bal}	{Bal}
Atomic Updates	{Am, DC, TS}, {Bal, DC}, {Bal, TS}	{Am, DC, TS}, {Bal}
Attr conflicts	{Am, DC, TS}, {Bal, DC}, {Bal, TS}	{Am, DC, TS}, {Bal}

Maximal robust subsets by analysis setting for **TPC-Ckv**:

	Robust subsets	[Alomari and Fekete, 2015]
Only R & W	{OS, SL}	{OS, SL}
Atomic Updates	{Del, Pay, SL}, {NO, SL}, {Pay, OS, SL}	{Del, Pay, SL}, {NO}, {OS, SL}
Attr conflicts	{Del, Pay, NO, SL}, {Pay, OS, SL}	{Del, Pay, SL}, {Del, Pay, NO} {OS, SL}

Increased transaction throughput

- PostgreSQL: isolation levels RC, SI and SSI.
- Robust subset of SmallBank benchmark: {Am, DC, TS}.
- 18000 bank accounts → small subset is a *hotspot*.
- 200 concurrent clients.

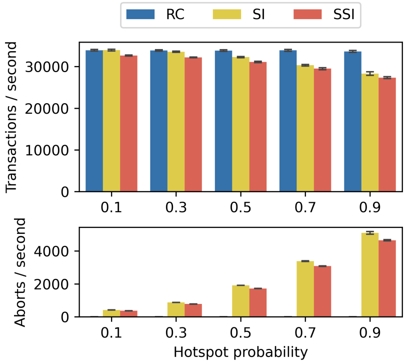


Figure: Hotspot of 1000 accounts.

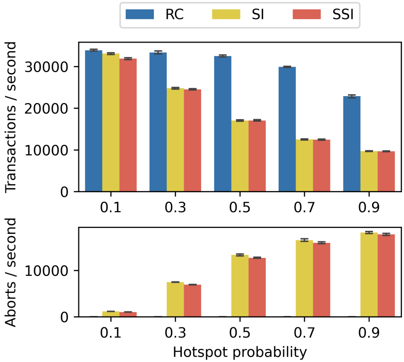


Figure: Hotspot of 100 accounts.

Obtaining robustness

Idea: Modify transaction templates to obtain robustness against RC, *without changing the semantics or database internals*.

Obtaining robustness

Idea: Modify transaction templates to obtain robustness against RC, *without changing the semantics or database internals*.

Promotion

Promote read operations to atomic updates that write back the read value.

Obtaining robustness

Idea: Modify transaction templates to obtain robustness against RC, *without changing the semantics or database internals*.

Promotion

Promote read operations to atomic updates that write back the read value.

Example: SmallBank benchmark

→ Promote all reads accessing a Savings or Checking account.

WriteCheck (original):

R[X : Account{Name, CustID}]
R[Y : Savings{CustID, Bal}]
R[Z : Checking{CustID, Bal}]
U[Z : Checking{CustID, Bal}{Bal}]

WriteCheck (promoted):

R[X : Account{Name, CustID}]
U[Y : Savings{CustID, Bal}{Bal}]
U[Z : Checking{CustID, Bal}{Bal}]
U[Z : Checking{CustID, Bal}{Bal}]

Experiments

Since we modified the templates, outperforming the higher isolation levels is no longer guaranteed!

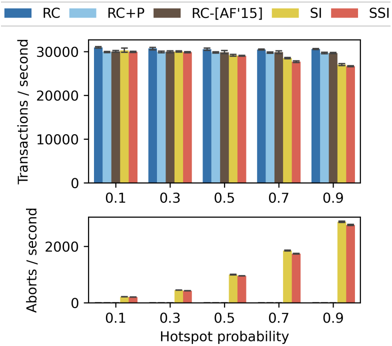


Figure: Hotspot of 1000 accounts.

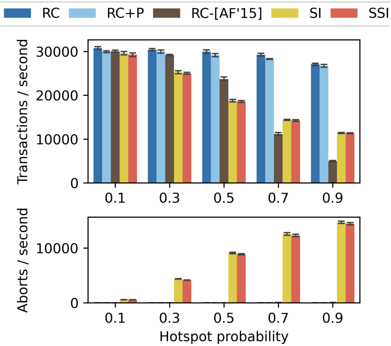


Figure: Hotspot of 100 accounts.

Conclusion

When contention increases, RC+promotion still outperforms higher isolation levels and related work.

Outline

Motivation

Tuples in a database are often related (e.g. foreign key constraints).
$\rightarrow$ modelled as functions.

Functions for SmallBank benchmark

"Each bank account is related to exactly one checking and one savings account." $\Rightarrow$

function f	*dom*(f)	*range*(f)
$f_{A\rightarrow C}$	Account	Checking
$f_{A\rightarrow S}$	Account	Savings

Transaction templates with functional constraints

Example: SmallBank benchmark

Amalgamate:

$$\begin{aligned}
&\mathtt{R}[\mathtt{X}_1 : \text{Account}\{\text{N, C}\}] \\
&\mathtt{R}[\mathtt{X}_2 : \text{Account}\{\text{N, C}\}] \\
&\mathtt{U}[\mathtt{Y}_1 : \text{Savings}\{\text{C, B}\}\{\text{B}\}] \\
&\mathtt{U}[\mathtt{Z}_1 : \text{Checking}\{\text{C, B}\}\{\text{B}\}] \\
&\mathtt{U}[\mathtt{Z}_2 : \text{Checking}\{\text{C, B}\}\{\text{B}\}] \\
&\mathtt{X}_1 \neq \mathtt{X}_2, \\
&\mathtt{Y}_1 = f_{A \to S}(\mathtt{X}_1) \\
&\mathtt{Z}_1 = f_{A \to C}(\mathtt{X}_1) \\
&\mathtt{Z}_2 = f_{A \to C}(\mathtt{X}_2)
\end{aligned}$$

GoPremium:

$$\begin{aligned}
&\mathtt{U}[\mathtt{X} : \text{Account}\{\text{N, C}\}\{\text{I}\}] \\
&\mathtt{R}[\mathtt{Y} : \text{Savings}\{\text{C, I}\}] \\
&\mathtt{U}[\mathtt{Y} : \text{Savings}\{\text{C}\}\{\text{I}\}] \\
&\mathtt{Y} = f_{A \to S}(\mathtt{X})
\end{aligned}$$

Variable assignment should respect all functional constraints.
$\rightarrow$ Rules out schedules that cannot occur in practice.

Functional constraints and robustness

By including functional constraints, we can...

- ...detect more sets of templates as robust against RC;

	Robust subsets SmallBank benchmark
Only R & W	{Bal}
Atomic Updates	{Am, DC, TS}, {Bal, DC}, {Bal, TS}
Attr Conflicts	{Am, DC, TS}, {Bal, DC}, {Bal, TS}
Func Constraints	{Am, DC, TS, GP}, {Bal, DC, GP}, {Bal, TS, GP}

- ...reduce the number of promoted reads required to obtain robustness against RC (e.g. TPC-Ckv).

Deciding robustness against RC

Theorem [Vandevoort et al., 2022]

Robustness against RC for transaction templates with functional constraints is undecidable, even *without disequality constraints*.

Schema graph:

- in NLOGSPACE when functions are bijections and schema graph is acyclic
- in EXPSPACE when schema graph is acyclic Further improvements by restricting...
  - ...templates → EXPTIME.
  - ...number of paths in schema graph → PSPACE.

Outline

Limitations

Assumptions in our formalism:

- **No predicate reads**: tuples are accessed based on a key value that cannot be modified.
  - When including predicate reads, iterating over multiversion split schedules is no longer sufficient.
  - Currently working on a sufficient condition for robustness against RC for a setting with predicate reads.
- All transactions are executed under the **same isolation level**.

Outline

Summary

- Complete characterizations for robustness against RC, MVRC, and SI for workloads specified as transactions. Provide insight into the structure of problematic behaviour.
- Algorithms detecting robustness for workloads specified as transaction templates (with functional constraints).
- Code modification (promotion) to obtain robustness against RC.
- Experimental validation of improved robustness detection (compared to related work) and increased throughput.

Research directions

- Robustness under different **notions for serializability**: final-state serializability, view serializability, semantic serializability.
- **Undecidability boundary** for transaction templates with functional constraints
- **Allocation problem**:
  - given a set of transactions $\mathcal{T}$ and a set of isolation levels $S_{\mathcal{I}}$: assign isolation levels to transactions such that serializability is guaranteed and performance is optimal.
    - addressed by [Fekete, 2005] for SI and S2PL.
- **Quantifying non-robustness**:
  - Probabilistically: How likely is it that an allowed schedule is not serializable? (e.g., [Fekete et al., 2009])
  - Characterize non-serializable schedules (e.g., to help debug anomalies caused by using weaker isolation levels [Gan et al., 2020])
- Robustness for **distributed transactions**

Database concurrency control

A personal reflection

Database concurrency control

A personal reflection

The pros

Database concurrency control

A personal reflection

The pros

- From practice to theory (there and back again?)

Database concurrency control

A personal reflection

The pros

- From practice to theory (there and back again?)
- Relevant and challenging open questions

Database concurrency control

A personal reflection

The pros

- From practice to theory (there and back again?)
- Relevant and challenging open questions
- Classical DB theory (deserves more attention from PODS)

Database concurrency control

A personal reflection

The pros

- From practice to theory (there and back again?)
- Relevant and challenging open questions
- Classical DB theory (deserves more attention from PODS)

Database concurrency control

A personal reflection

The pros

- From practice to theory (there and back again?)
- Relevant and challenging open questions
- Classical DB theory (deserves more attention from PODS)

The cons

Database concurrency control

A personal reflection

The pros

- From practice to theory (there and back again?)
- Relevant and challenging open questions
- Classical DB theory (deserves more attention from PODS)

The cons

- It is not that easy to get into, but I hope you will :-)

Thank you for your attention

Work in collaboration with

Bas Ketsman
Vrije Universiteit Brussel

Christoph Koch
EPFL

Brecht Vandevoort
Universiteit Hasselt

References I

- Alomari, M., Cahill, M., Fekete, A., and Rohm, U. (2008).
  The cost of serializability on platforms that use snapshot isolation.
  In *ICDE*, pages 576–585.
- Alomari, M. and Fekete, A. (2015).
  Serializable use of read committed isolation level.
  In *AICCSA*, pages 1–8.
- Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J. M., and Stoica, I. (2013).
  HAT, not CAP: Towards highly available transactions.
  In *USENIX HotOS*, pages 24–24.
- Beillahi, S. M., Bouajjani, A., and Enea, C. (2019a).
  Checking robustness against snapshot isolation.
  In *CAV*, pages 286–304.

References II

- Beillahi, S. M., Bouajjani, A., and Enea, C. (2019b).
  Robustness against transactional causal consistency.
  In *CONCUR*, pages 1–18.
- Bernardi, G. and Gotsman, A. (2016).
  Robustness against consistency models with atomic visibility.
  In *CONCUR*, pages 7:1–7:15.
- Cerone, A. and Gotsman, A. (2018).
  Analysing snapshot isolation.
  J.ACM, 65(2):1–41.
- Cerone, A., Gotsman, A., and Yang, H. (2017).
  Algebraic Laws for Weak Consistency.
  In *CONCUR*, pages 26:1–26:18.

References III

- Fekete, A. (2005).
  Allocating isolation levels to transactions.
  In *PODS*, pages 206–215.
- Fekete, A., Liarokapis, D., O'Neil, E. J., O'Neil, P. E., and Shasha, D. E. (2005).
  Making snapshot isolation serializable.
  ACM Trans. Database Syst., 30(2):492–528.
- Fekete, A. D., Goldrei, S., and Asenjo, J. P. (2009).
  Quantifying isolation anomalies.
  PVLDB, 2(1):467–478.
- Gan, Y., Ren, X., Ripberger, D., Blanas, S., and Wang, Y. (2020).
  Isodiff: Debugging anomalies caused by weak isolation.
  Proc. VLDB Endow., 13(11):2773–2786.

References IV

- Ketsman, B., Koch, C., Neven, F., and Vandevoort, B. (2020).
  Deciding robustness for lower SQL isolation levels.
  In *PODS*, pages 315–330.
- Papadimitriou, C. H. (1986).
  The Theory of Database Concurrency Control.
  Computer Science Press.
- Vandevoort, B., Ketsman, B., Koch, C., and Neven, F. (2021).
  Robustness against read committed for transaction templates.
  PVLDB, 14(11):2141–2153.
- Vandevoort, B., Ketsman, B., Koch, C., and Neven, F. (2022).
  Robustness against read committed for transaction templates with functional constraints.
  ICDT 2022.