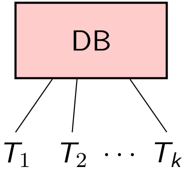
Constraint-Preserving Snapshot Isolation

Stephen J. Hegner
Umeå University
Department of Computing Science
SE-901 87 Umeå, Sweden
hegner@cs.umu.se
http://www.cs.umu.se/~hegner

Database Transactions



- A central feature of modern database-management systems (DBMSs) is the support of concurrent transactions.
- In general, these transactions may both read and write the database.
- When transactions write the database, the updates which they perform must respect the *integrity constraints* of the schema.
- It is generally assumed that these transactions respect the *consistency* property; that is, that they perform operations which preserve the integrity constraints when run alone.
- It is also necessary that these transactions operate in *isolation*, that is that they do not interfere with each other.
- If care is not taken, the results may not be as intended.

Serial Execution of Transactions

Serial execution: A set of transactions runs *serially* if there is no temporal overlap in their operations.

- Serial execution is considered to define optimal isolation, even though the result may depend upon the order of execution.

T_1	T_2	x
Read$\langle x \rangle$		10000
Cpd$\langle x, 10\% \rangle$		10000
Write$\langle x \rangle$		11000
	Read$\langle x \rangle$	11000
	Wd$\langle x, 2000 \rangle$	11000
	Write$\langle x \rangle$	9000

T_1	T_2	x
	Read$\langle x \rangle$	10000
	Wd$\langle x, 2000 \rangle$	10000
	Write$\langle x \rangle$	8000
Read$\langle x \rangle$		8000
Cpd$\langle x, 10\% \rangle$		8000
Write$\langle x \rangle$		8800

- The operations *Cpd* = *compound* and *Wd* = *withdraw* operate internally and do not write the database.

Lost Updates

- If the steps of the transactions are interleaved in certain ways, isolation may be lost.
- One symptom of poor isolation is *lost updates*.

T_1	T_2	x
Read$\langle x \rangle$		10000
Cpd$\langle x, 10\% \rangle$		10000
	Read$\langle x \rangle$	10000
	Wd$\langle x, 2000 \rangle$	10000
	Write$\langle x \rangle$	8000
Write$\langle x \rangle$		11000

T_1	T_2	x
	Read$\langle x \rangle$	10000
	Wd$\langle x, 2000 \rangle$	10000
Read$\langle x \rangle$		10000
Cpd$\langle x, 10\% \rangle$		10000
Write$\langle x \rangle$		11000
	Write$\langle x \rangle$	8000

- In the schedule on the left, the result of T_2 is lost.
- In the schedule on the right, the result of T_1 is lost.

The Model of Operations, Transactions, and Schedules

- Model the database schema as a set of updateable objects.

Object-level model of operations: There are two basic operations:

Read: $\mathrm{r}_T\langle x\rangle$ denotes that transaction T reads data object x.

Write: $\mathrm{w}_T\langle x\rangle$ denotes that transaction T writes data object x.

- In particular, the specific change which T makes to the value of x during a write is **not** modelled.
- A *transaction* is then modelled as a sequence of such operations:

Examples: T_1: $\mathrm{r}_{T_1}\langle x_1\rangle \mathrm{w}_{T_1}\langle x_1\rangle \mathrm{r}_{T_1}\langle x_2\rangle \mathrm{w}_{T_1}\langle x_2\rangle$ T_2: $\mathrm{r}_{T_2}\langle x_1\rangle \mathrm{r}_{T_2}\langle x_3\rangle \mathrm{w}_{T_2}\langle x_3\rangle \mathrm{w}_{T_2}\langle x_2\rangle$

- A *schedule* for a set of transactions is an intertwining of their operation sequences which preserves the local order for each transaction.

Examples:

S_1 : $\mathrm{r}_{T_1}\langle x_1\rangle \mathrm{w}_{T_1}\langle x_1\rangle \mathrm{r}_{T_1}\langle x_2\rangle \mathrm{w}_{T_1}\langle x_2\rangle \mathrm{r}_{T_2}\langle x_1\rangle \mathrm{r}_{T_2}\langle x_3\rangle \mathrm{w}_{T_2}\langle x_3\rangle \mathrm{w}_{T_2}\langle x_2\rangle$

S_2 : $\mathrm{r}_{T_1}\langle x_1\rangle \mathrm{w}_{T_1}\langle x_1\rangle \mathrm{r}_{T_2}\langle x_1\rangle \mathrm{r}_{T_2}\langle x_3\rangle \mathrm{w}_{T_2}\langle x_3\rangle \mathrm{r}_{T_1}\langle x_2\rangle \mathrm{w}_{T_1}\langle x_2\rangle \mathrm{w}_{T_2}\langle x_2\rangle$

- S_1 is a *serial* schedule for $\{T_1, T_2\}$, while S_2 is a non-serial schedule.

The Gold Standard for Isolation: View Serializability

Idea: A schedule is *view serializable* if it can be obtained by rearranging the operations of some serial schedule in such a way that:

- The read operations read from the same writer in each case (which might be the initial database state).
- The final writer of each data object is the same transaction in each case.

- Such a rearrangement does not change the final result of running the transactions.

Examples:

S_1 : $r_{T_1}\langle x_1\rangle w_{T_1}\langle x_1\rangle r_{T_1}\langle x_2\rangle w_{T_1}\langle x_2\rangle r_{T_2}\langle x_1\rangle r_{T_2}\langle x_3\rangle w_{T_2}\langle x_3\rangle w_{T_2}\langle x_2\rangle$

S_2 : $r_{T_1}\langle x_1\rangle w_{T_1}\langle x_1\rangle r_{T_2}\langle x_1\rangle r_{T_2}\langle x_3\rangle w_{T_2}\langle x_3\rangle r_{T_1}\langle x_2\rangle w_{T_1}\langle x_2\rangle w_{T_2}\langle x_2\rangle$

S_3 : $r_{T_1}\langle x_1\rangle r_{T_2}\langle x_1\rangle w_{T_1}\langle x_1\rangle r_{T_2}\langle x_3\rangle w_{T_2}\langle x_3\rangle r_{T_1}\langle x_2\rangle w_{T_1}\langle x_2\rangle w_{T_2}\langle x_2\rangle$

S_4 : $r_{T_1}\langle x_1\rangle w_{T_1}\langle x_1\rangle r_{T_2}\langle x_1\rangle r_{T_2}\langle x_3\rangle w_{T_2}\langle x_3\rangle r_{T_1}\langle x_2\rangle w_{T_2}\langle x_2\rangle w_{T_1}\langle x_2\rangle$

- S_1 and S_2 are view serializable.
- S_3 is not view serializable (changed read source of $r_{T2}\langle x_1\rangle$).
- S_4 is not view serializable (changed final write of x_2).

Guaranteeing View Serializability — SS2PL

System requirement: Need a scheduling algorithm which guarantees view-serializable schedules, not just a test for view serializability.

Strong strict two-phase locking (SS2PL): The classical lock-based solution.

- *Shared* (read) locks and *exclusive* (write) locks are required for all data access.
- Locks may be acquired at any time.
- All locks held until the transaction commits (ends).

Severe drawback: The locking requirements greatly limit concurrency.

- Querying on a non-indexed attribute would require locking the entire table until the end of the transaction!

Incorrect claim: Many DBMS textbooks incorrectly assert that SS2PL is widely used in practice to realize serializable isolation.

- Unknown to many users, the SQL `SERIALIZABLE` mode of isolation does **not** provide view serializability in many systems (*e.g.*, Oracle).
- Even with those systems which do implement SS2PL, it is not widely used due to poor performance.

Levels of Isolation of Transactions

Question: Isn't view serializability necessary to guarantee correct results?

Answer: That depends upon what is meant by "correct".

- Isolation is a matter of degree.

Real-world fact: Lower levels of isolation are used routinely.

- The default level of isolation in many real systems is *read committed*, which guarantees that only committed data are read, but little more.
- Many transactions can tolerate such lower isolation levels without suffering serious consequences.
- The highest level, *view serializable*, is used only where absolutely essential, such as in financial transactions.

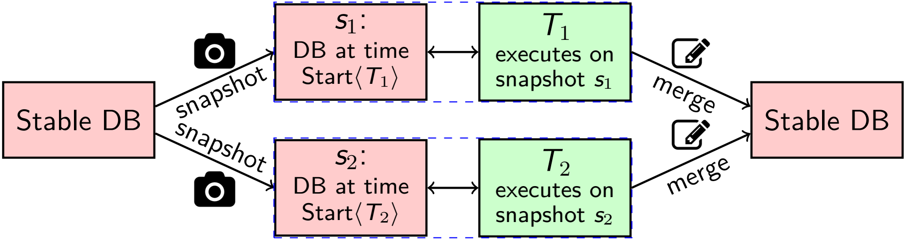
Multiversion Concurrency Control

MVCC: Most modern DBMSs employ *multiversion concurrency control*.

- There may be several *versions* of a given data object x.
- Rather than requiring locks, concurrency is achieved by allowing distinct transactions to operate on distinct versions of x.
- Differences must eventually be resolved, but typically not at the expense of long waits.
- In general, MVCC supports far more concurrency than single-version, lock-based approaches.

- One of the most common approaches within MVCC for achieving a high level of isolation with substantial concurrency is called *snapshot isolation*.
- Because the approach of this research is based upon it, it is worth a closer look.

Snapshot Isolation



- In *snapshot isolation (SI)*, each transaction operates on a *snapshot*:
  - a (private) copy of the database with values taken at the point in time at which the transactions begins.

First Committer Wins (FCW): T_i is allowed to commit its local writes to the stable DB only if no data object x which it writes has been committed, since its snapshot was created, to the stable DB by another transaction.

- Otherwise, it must abort and start over.

Advantages of Snapshot Isolation

- SI has some very attractive properties.

High Level of Isolation: Since each transaction operates on a private copy, isolation is achieved at what appears to be at a relatively high level.

Enhanced concurrency: No locks $\Rightarrow$ writers do not block readers.

  - Readers (almost) never have to wait for writers to finish.
  - The attainable level of concurrency is far greater than that of SS2PL.

- For these reasons, SI is widely used in practice.

Real systems use *first updater wins (FUW)*, and there may be some blocking when foreign-key constraints are checked, but these are details which do not distort the main conclusions.

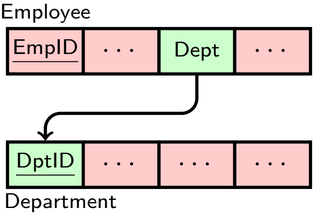
Question: Does SI provide serializable-level isolation?

Answer: That depends upon the definition of *serializable*.

Interdependent Data Objects

Fact: SI does not guarantee view-serializable isolation. □

- An example is defined by a foreign key constraint.



T_1: Delete the *Research* department (which has no employees assigned to it) [modifies Department only].

T_2: Assign Alice to the *Research* department [modifies Employee only].

- Each of T_1 and T_2 may be run by itself with no violation of integrity constraints.
- T_1 and T_2 operate on distinct data objects, yet if run concurrently, a constraint violation occurs if both commit.

Write Skew — Constraint Violation under SI

Fact: Built-in constraints are managed internally by all modern DBMSs, so the previous example, while instructive, is not relevant in a practical sense.

- On the other hand, consraint enforcement for the following situation would likely be implemented with triggers and so not handled internally.

Example (write skew): x and y represent the balances of two accounts.

Integrity constraint: $x + y \geq 500€$ Initial state: $x = 300€$, $y = 300€$

T_1: Withdraw $100€$ from x T_2: Withdraw $100€$ from y.

- Assume that these transactions run concurrently under SI.
- Each transaction run in isolation satisfies the integrity constraint.
- The final state is $(x, y) = (200€, 200€)$, which violates the constraint.
- With serial execution, the second transaction will fail.
- Thus, SI does not guarantee view serializability.

The SQL Standard and Serializability

- SI satisfies the conditions set forth in the SQL standard for the `SERIALIZABLE` isolation level.
- The standard **defines** serializability as the absence of three types of transaction anomalies.

Apparent reason: The architects of the standard could not think of any nonserializable behavior which could arise in the absence of violations of those anomalies.

Consequence: Real systems are free to implement the `SERIALIZABLE` level of isolation as SI, and several do so.

- Unfortunately, many users mistakenly believe that `SERIALIZABLE` isolation in SQL must mean view serializable.

Opinion/Rant: The definition of `SERIALIZABLE` in the SQL standard is a poster child for why good theory is a necessary part of even the most practical endeavors.

The DSG and Conflict Serializability

DSG: The *direct serialization graph (DSG)* has transactions as vertices and three types of edges:

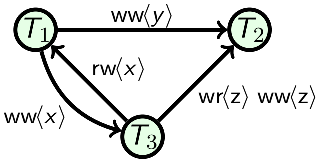
$T_i \xrightarrow{\text{rw}\langle x\rangle} T_j$: T_i reads x and T_j is the next writer of x.

$T_i \xrightarrow{\text{ww}\langle x\rangle} T_j$: T_i and T_j are consecutive writers of x.

$T_i \xrightarrow{\text{wr}\langle x\rangle} T_j$: T_j reads x and T_i is the previous writer of x.

Example: The DSG for

$\text{r}_1\langle x\rangle \text{r}_1\langle y\rangle \; \text{r}_3\langle z\rangle \text{w}_3\langle z\rangle \text{r}_3\langle x\rangle \; \text{r}_2\langle z\rangle \; \text{w}_1\langle x\rangle \text{w}_1\langle y\rangle \; \text{w}_2\langle z\rangle \text{w}_2\langle y\rangle \; \text{w}_3\langle x\rangle$



Theorem: Cycle-free DSG $\Leftrightarrow$ *conflict serializability* $\Rightarrow$ *view serializability*. □

- Stronger than view serializability but the differences are anomalous.
- Useful for testing because the computational complexity is low.

Serializable Snapshot Isolation

Serializable SI (SSI): Augment SI to achieve true view serializability.

Observation: With all transactions running under SI, if T_i and T_j are concurrent and there is an edge $T_i \longrightarrow T_j$ in the DSG, then it must be an rw-edge. □

Dangerous structure in DSG: $T_i \xrightarrow{\text{rw}} T_j \xrightarrow{\text{rw}} T_k$ ($T_i = T_k$ possible) *occurring in a cycle* with $\{T_i, T_j\}$ and $\{T_j, T_k\}$ concurrent.

Theorem [Fekete *et al* 2005]: If a schedule for SI is not view serializable, the DSG must contain a dangerous structure. □

Optimistic strategy: Serializable SI (SSI):

- It is too expensive to maintain the entire DSG.
- Look for *potential* dangerous structures (need not be part of a cycle) and require one transaction to terminate to preserve serializability.
- This requires testing only three transactions at a time.
- But there will be false positives.

Serializable Snapshot Isolation — Practice and Limitations

Use in PostgreSQL: As of version 9.1, SSI is used to implement `SERIALIZABLE` isolation in PostgreSQL.

- Thus, `SERIALIZABLE` isolation is finally truly view serializability.
- Ordinary SI is still available as `REPEATABLE READ` isolation.
- Before version 9.1, both isolation levels were implemented as SI.

Question: Why is there a need for anything more?

Answers:

- SSI results in more false positives (with consequent aborts and reruns) than does ordinary SI.
- For some transaction mixes (particularly interactive and long-running), this may be a severe drawback.

Question: Is there something in between SI and SSI?

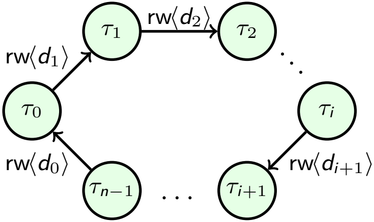
Answer: Yes, *constraint-preserving SI (CPSI)*, the topic of this research.

- Ensures that constraints will be satisfied (no write skew).
- Much simpler algorithm with limited false positives.

Permutation – Nonserializability without Constraint Violation

Example (SI permutation): $n \in \mathbb{N}$;

- d_0,, d_1, ...d_{n-1} data objects.
- $\tau_0, \tau_1, \ldots, \tau_{n-1}$ transactions with τ_i: $d_i \leftarrow d_{(i+1) \bmod n}$.
- The n transactions, run concurrently under SI, effect a permutation of the values of the d_i's (shift clockwise).



- $\tau_i \xrightarrow{\text{rw}\langle d_i \rangle} \tau_{(i+1) \bmod n}$ denotes that τ_1 reads d_i and $\tau_{(i+1) \bmod n}$ writes it.
- This behavior cannot be view serializable since if τ_i is run first, the old value of d_i is lost.
- However, if any transaction (say τ_i) is removed, the result of running all transactions concurrently under SI is serializable.
  - Run them in this order: $\tau_{i+1} \ldots \tau_{n-1} \tau_0 \ldots \tau_{i-1}$.

Observation: For any $n \in \mathbb{N}$, there is a set of n transaction which, when run concurrently under SI, results in nonserializable behavior, yet any proper subset produces serializable behavior under SI. □

Two Types of Reads under SI

Example: Let the database schema have three data objects w, x, and y with the constraint $x + y \geq 500$.

- Transaction T defined by $x \leftarrow x - w$.
- y is the *guard* of the transaction; it must be read in order to verify that the update will satisfy the integrity constraint.
- w must be read only to determine the update; it is not used in the checking the integrity constraint.

The value of y when T commits is critical: If the value of the guard y of T is changed by another concurrent transaction, there is a risk that the constraint will be violated.

Only the snapshot value of w is important for constraint satisfaction: A change to the value of w by another concurrent transaction will not affect whether or not the constraint is satisfied.

The Guard of a Data Object

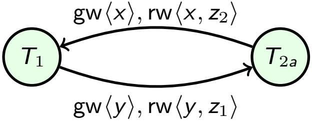
Guard of a transaction: The *guard* of a transaction T is the set of all data objects which must be read by T in order to verify the integrity constraints, but which are not written by T.

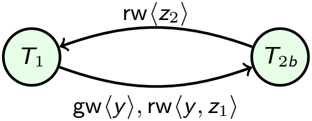
Example: Integer data objects: $\{x, y, z_1, z_2\}$; Constraint: $x + y \geq 500$.

Transaction	Write Set	Read Set	Guard Set
$T_1 : \ x \leftarrow x - z_1; \ z_2 \leftarrow z_2 - 10$	$\{x, z_2\}$	$\{y, z_1\}$	$\{y\}$
$T_{2a} : \ y \leftarrow y + z_2; \ z_1 \leftarrow z_1/2$	$\{y, z_1\}$	$\{x, z_2\}$	$\{x\}$
$T_{2b} : \ y \leftarrow y + \lvert z_2 \rvert; \ z_1 \leftarrow z_1/2$	$\{y, z_1\}$	$\{z_2\}$	$\emptyset$

gw-edge $T_i \xrightarrow{\text{gw}} T_j$ in the (augmented) DSG: T_j writes the guard of T_i.

- $T_i \xrightarrow{\text{gw}} T_j \ \Rightarrow \ T_i \xrightarrow{\text{rw}} T_j$ but not conversely.

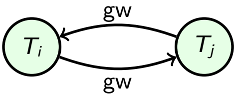




Note: T_{2a} and T_{2b} are alternatives; they cannot run concurrently.

Guard Independence and CPSI

Guard independence of two transactions T_1 and T_2 is the formalization of the condition that a cycle of the form

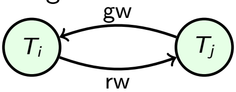


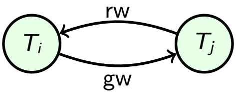
does not exist.

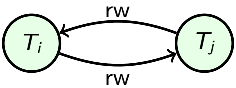
Theorem: Let $\mathbf{T} = \{T_1, T_2, \ldots, T_m\}$ be a set of transactions running under SI according to some schedule $\mathbf{S}$. If every pair of *concurrent* transactions is guard independent, then the result is guaranteed to satisfy all integrity constraints. □

CPSI: Require all pairs of concurrent transactions to be guard independent.

Remark: Cycles of the following three forms are allowed, as long as the rw-edges do not involve guard objects:





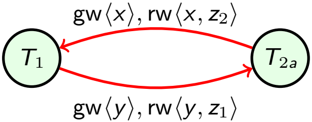


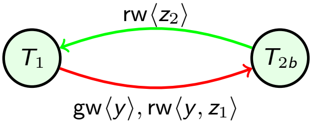
- Assuming all guard objects are read, these would identify a dangerous structure in SSI and result in the termination of one of the transactions.

Example of Guard Independence

Example: Data objects: $\{x, y, z_1, z_2\}$; Constraint: $x + y \geq 500$.

Transaction	Write Set	Read Set	Guard Set
$T_1 : x \leftarrow x - z_1;\ z_2 \leftarrow z_2 - 10$	$\{x, z_2\}$	$\{y, z_1\}$	$\{y\}$
$T_{2a} : y \leftarrow y + z_2;\ z_1 \leftarrow z_1/2$	$\{y, z_1\}$	$\{x, z_2\}$	$\{x\}$
$T_{2b} : y \leftarrow y + \lvert z_2 \rvert;\ z_1 \leftarrow z_1/2$	$\{y, z_1\}$	$\{z_2\}$	$\emptyset$





Note: rw$\langle \alpha \rangle$ not shown if gw$\langle \alpha \rangle$ also holds for data object α on an edge.

- T_1 and T_{2b} are guard independent, while T_1 and T_{2a} are not.

Note: T_{2a} and T_{2b} are alternatives; they cannot run concurrently.

CPSI and False Positives

- False positives may occur under CPSI to the extent that a transaction may avoid reading the entire guard.
  - This is possible if "clever" coding is used.
  - For the most part, such coding is possible only if transactions enforce constraints locally, not if they are implemented using triggers.
  - However, it is possible under certain special circumstances if the trigger is implemented in a very clever way.

Bottom line: The occurrence of false positives depends very much upon how a false positive is defined.

  - All approaches involve false positives to some degree, in that reads or writes may be benign.
  - CPSI avoids many of the false positives which occur under SSI.

CPSI+SSI: CPSI and SSI may be combined so that the only false positives are those which occur in both.

CPSI+CSSI: Even fewer false positives; only dangerous structures caused by guard reads are considered in the SSI component.

Applications of CPSI

Interactive transactions: Those with a human in the loop making decisions.

Example: Business processes; employee requesting travel funds.

- Running time may be extremely long (days).
- Abort and restart is not a viable option.

Negotiation: For interactive transactions, *negotiation* is often a far superior alternative to abort and restart when conflicts occur.

- The transactions in conflict "negotiate" a solution in which the conflict does not occur.

CPSI and negotiation: In CPSI, all conflicts are binary and the conflicts are explicitly identified by the guards.

- This makes it particularly feasible to identify conflicting parties for negotiation.

Conclusions and Further Directions

Conclusions:

New Isolation Level: A new isolation level, *constraint-preserving snapshot isolation (CPSI)*, has been investigated.

SI < CPSI < Ser: It is at a strictly higher level than snapshot isolation, and a strictly lower level than view serializability.

- The test for adherence is much simpler than that for serializable snapshot isolation, with far less risk of false positives.

Further Directions:

Implementation and performance studies: It would be very useful to see how this approach fares in various situations.

Extension to a value-level model: Work is underway to extend the approach to a *value-level model*, in which the transaction manager has simple information about the nature of the updates which the transactions perform.

- This type of extension is critical for *interactive transactions*, in which abort and rerun is not an acceptable strategy for resolving conflicts.

More Information

Comprehensive slides: Slides (124 of them) entitled *Transaction models and concurrency control* from the course *Database System Principles* at Umeå University:

`http://www8.cs.umu.se/kurser/5DV120/V15/Slides/09_trans_5dv120_h.pdf`

Research paper: Hegner, Stephen J., Constraint-preserving snapshot isolation, *Annals of Mathematics and Artificial Intelligence*, to appear:

`http://www8.cs.umu.se/~hegner/Publications/PDF/amai15.pdf`