Constraint-Preserving Isolation of Database Transactions

Stephen J. Hegner

Retired from:
Umeå University, Sweden
hegner@cs.umu.se
http://www.cs.umu.se/~hegner

Currently:
Hegner Consulting, LLC
New London, NH, USA

Database Transactions: Out of Fashion?

- Modern DBMSs support both reads and writes by many users.
- A *transaction* is a series of reads and/or writes, by a single user, to achieve a specific task.
- Virtually all classical and relational DBMSs support transactions.
- Some systems *à la nouvelle mode* (*e.g.*, NoSQL) do not.

Question: Are transactions *démodées*?

Answer: It depends upon the application.

- Some applications do not always require transactions.
  - Operations on the database of a Web-search engine.
  - Operations on the database of a social-networking service.
- Many applications still do require transactions to achieve *isolation*.
  - Financial operations.
  - Legal and court records.
- An example will help illustrate.

Serial Execution of Transactions

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

- In other words, no concurrency.

- 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.
- This can happen with PostgreSQL using the default isolation level `READ COMMITTED`.

Conclusion: Transactions are still important!!!

Isolation of Transactions

Isolation: Concurrent transactions should not interfere with each other.

Reality: Isolation is a matter of degree.

Tradeoff:
- Higher isolation ⇒ reduced concurrency.
- Lower isolation ⇒ undesirable interaction.

Accommodation: DBMSs offer a variety of isolation levels.

Reality: Lower levels of isolation are routinely used in order to achieve satisfactory performance.

- Often with unexpected and/or disastrous side effects.
- Errors can be subtle and extremely difficult to detect ..
  - .. until it is too late.
- Major relational DBMSs (PostgreSQL, Oracle, SQL Server) default to `READ COMMITTED`, a very low (often the lowest) level ..
  - .. despite the fact that the SQL standard mandates the default to be the highest level `SERIALIZABLE`.

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.

View Serializability and SS2PL

Serializability: A schedule of transactions is *view serializable* if its effect is the same as one in which the transactions are run serially.

- Theoretical gold standard for isolation.

Requirement: The scheduler needs to create serializable schedules, not just test existing ones for compliance.

SS2PL: *Strong-Strict Two-Phase Locking* is a lock-based means of ensuring view-serializable schedules.

- A transaction must hold all acquired locks until it commits (finishes).
- Also guarantees other desirable properties (*e.g.*, *recoverability*).

False claim: Many textbooks claim that SS2PL is widely used in practice.

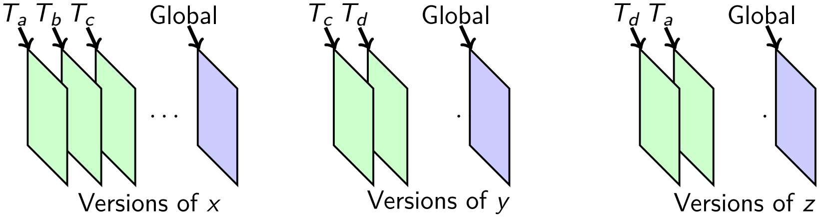
Reality: SS2PL is too limiting of concurrency to be of much use.

- A search on a non-indexed attribute would require (read) locking the whole table, blocking write access by any concurrent transaction.

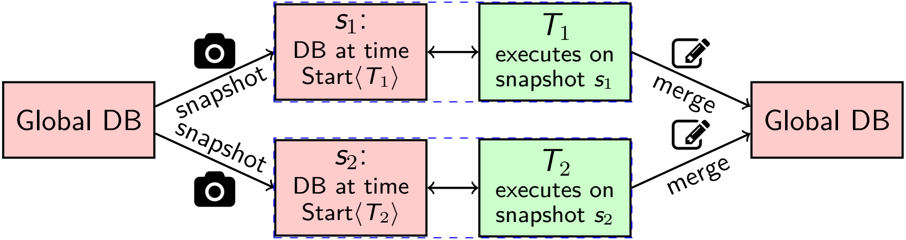
- Only a few relational DBMSs even offer it.

Multiversion Concurrency Control

- Historically, systems which work with just a single version of the database have have been widely used in DBMSs.
  - This model is called *single-version concurrency control (SVCC)*.
- Nowadays, however, a much more common approach is *multiversion concurrency control (MVCC)*.
- In MVCC, there may be several versions of each primitive data object.
- Exactly one version for each data object is committed, and belongs to the *global DB* – the "true" database which other transactions can see.
- The others are local copies (for read and write) of transactions.
- The local copies are transferred to the global DB at transaction commit.



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 global DB only if no data object x which it writes has been committed, since its snapshot was created, to the global 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 ⇒ 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.

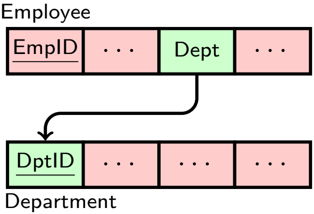
Question: Does SI provide serializable-level isolation?

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, constraint 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.

Constraint-Free Nonserializable SI

- It is also possible to have non-serializability without any constraint violations.

Example: Two transactions, two data objects.

- $T_1 : \; x_1 \leftarrow x_2$ $\qquad$ $T_2 : \; x_2 \leftarrow x_1$.

If executed serially: $x_1 = x_2$ when finished.

If executed concurrently under SI: The values are swapped.

Extension to n variables $x_0, \ldots x_{n-1}$ and n transactions $T_0, \ldots T_{n-1}$:

- $T_i : \; x_i \leftarrow x_{(i+1) \bmod n}$.

If executed serially: One value is always lost.

If all transactions executed concurrently: Rotation of values.

If any transaction removed: Execution is always serializable.

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:

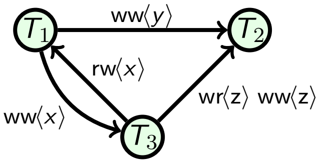
$T_i \xrightarrow{\mathrm{rw}\langle x\rangle} T_j$: T_i reads x and T_j is the next writer of x.

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

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

Example: The DSG for

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



Theorem: Cycle-free DSG ⇔ *conflict serializability* ⇒ *view serializability*. □

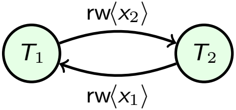
- Stronger than view serializability but the differences are anomalous.
- Useful for testing because the computational complexity is low.

Serializable Snapshot Isolation for Two Transactions

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

Simple case: First consider the case of just two concurrent transactions.

Theorem: If a schedule for SI for two concurrent transactions T_1 and T_2 is not view serializable, the DSG must contain a cycle of the following form for some data objects x_1, x_2.



Observation: The two simple examples of non-serializable SI (write skew and swap) have this property.

Strategy: If such a cycle occurs, abort one transaction, allowing the other to finish.

- Then re-run the aborted transaction.

Question: How can this be extended to more than two transactions?

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 transactions running under SI is not view serializable, the DSG must contain a dangerous structure. □

Observation: If $T_i = T_k$, this reduces to the case of the previous slide.

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: Since version 9.1 (late 2011), SSI has been 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.

Remark: The SSI algorithm works even if some transactions run at the lower READ COMMITTED level.

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, this may be a severe drawback.

Long-Running and Interactive Transactions

Long-running transaction: Impractical to abort and rerun because of their length .. .. hours or days or more in running time.

Interactive transaction: Human input (in response to transaction output) is part of the process.

Rich source of examples: Business processes.

Example: Employee request for travel.

- Requires financial resources and time away from the office.
  - Approval by management and accounting as interactive process.
- Requires travel resources (transportation, lodging).
  - Travel agent involved interactively.

Consequences of abort: All of these interactive sessions would be required to start over, from scratch.

Conclusion: It is far preferable to avoid such aborts, if at all possible.

Practical aspect: Because the transactions run much more slowly, it is reasonable to use more time-consuming, sophisticated strategies in order to avoid the need to abort and rerun.

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 $x \leftarrow x - w$.

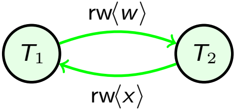
Integrity context: y is the *guard* of the transaction; it must be read in order to verify that the update will satisfy the integrity constraint.

Grounding context: 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.

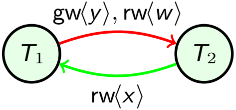
Only the snapshot value of w is important for constraint satisfaction: A change to the value of w by another concurrent transaction will not affect whether or not the constraint is satisfied.

Example: Constraints: $x + y \geq 500; \quad w > 0$
$T_1 : \; x \leftarrow x - w$
$T_2 : \; w \leftarrow w - x$



There is no conflict since both reads are grounding.

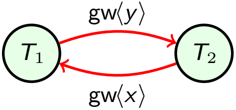
Example: Constraints: $x + y \geq 500; \quad w > 0$
$T_1 : \; x \leftarrow x - w$
$T_2 : \; w \leftarrow w + |x|; \quad y \leftarrow y + 1$



There is no conflict since the read of x by T_2 is grounding.

- The read of y is shown as g⟨y⟩ for *guard*.

Example: Constraints: $x + y \geq 500; \quad w > 0$
$T_1 : \; x \leftarrow x - w$
$T_2 : \; y \leftarrow y - 1$



Both reads are guard (integrity) reads, so there is a potential conflict.

Value-Level Modelling of Transactions

Object-level modelling: $\mathrm{r}_T\langle x\rangle$ and $\mathrm{w}_T\langle x\rangle$, but no information on what is read or written.

Value-level modelling: In addition to identifying which data objects are read or written, information about the actual values may also be involved.

Examples: Read values may be constrained to lie within a *tolerance* and write values guaranteed within a *range*.

- $\mathrm{r}_T\langle x\rangle{:}\mathrm{Tol}\langle S\rangle$ Transaction T reads data object x, with the requirement that any changes to the value must lie within S.
- $\mathrm{w}_T\langle x\rangle{:}\mathrm{Rng}\langle S\rangle$ Transaction T writes data object x, with the guarantee that the new value will lie in S.
  - For writes, S is taken to be a single value in this work.

- Support for value-level modelling requires more time and resources, but for interactive transactions, the tradeoff is reasonable.

Tolerant CPSI for Two Transactions and Two Data Objects

Main idea: For two *concurrent* transactions T_1 and T_2, and data object x

$$\mathsf{r}_{T_1}\langle x\rangle{:}\mathsf{Tol}\langle S_r\rangle \wedge \mathsf{w}_{T_2}\langle x\rangle{:}\mathsf{Rng}\langle S_w\rangle \quad\Rightarrow\quad S_w \subseteq S_r$$

Example: Constraint: $x + y \geq 500$; Initial state: $\langle x, y\rangle = \langle 400, 300\rangle$.

$T_1: \; x \leftarrow x - 100$ $\qquad$ $T_2: \; y \leftarrow y - 50$

$\mathsf{r}_{T_1}\langle y\rangle{:}\mathsf{Tol}\langle [200, \infty)\rangle$ $\qquad$ $\mathsf{r}_{T_2}\langle x\rangle{:}\mathsf{Tol}\langle [250, \infty)\rangle$

$\supseteq$

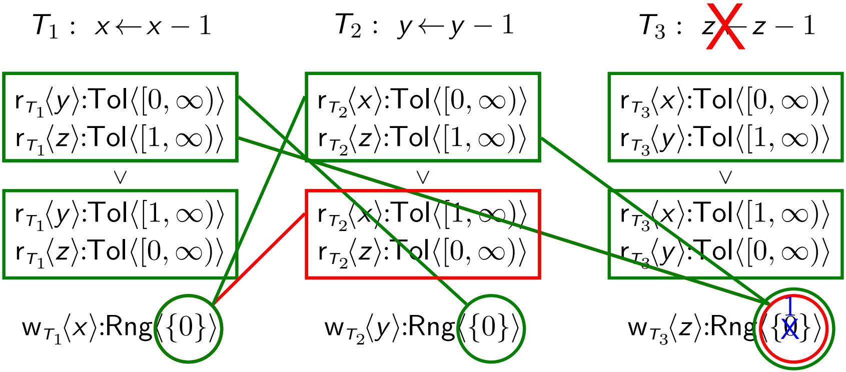
$\subseteq$

$\mathsf{w}_{T_1}\langle x\rangle{:}\mathsf{Rng}\langle \{300\}\rangle$ $\qquad$ $\mathsf{w}_{T_2}\langle y\rangle{:}\mathsf{Rng}\langle \{250\}\rangle$

- T_1 and T_2 are not in conflict under TCPSI.
- They are in conflict under ordinary CPSI (and SSI).

Tolerant CPSI for >2 Transactions and/or >2 Data Objects

Example: Constraints: $x + y + z \geq 1$, $x, y, z \geq 0$; Init: $\langle x, y, z \rangle = \langle 1, 1, \overset{2}{\not 1} \rangle$.



Problem: Results in complex system of constraints which must be solved.

Solution: Require each transaction to select one of the disjuncts.

Sketch of Details of Tolerant CPSI

- There are many details which are necessary to address in a successful deployment of TCPSI.

Calculus of combining tolerances: When several transactions have read tolerances on the same data object, the tolerances must be integrated.

Dynamic declaration of read tolerances: by transactions during execution.

- A transaction may choose which disjunct to use dynamically.
- It may observe the current read tolerances and write ranges of concurrent transactions in so doing.

Waiting of blocked transactions: for the necessary tolerances and ranges in order to continue.

- This may be preferable to abort and restart in some cases.

Grouping data objects for tolerance: Example: $\mathrm{r}_T\langle x_1\rangle \mathrm{r}_T\langle x_2\rangle$:$\mathrm{Tol}\langle x_1 x_2\rangle$.

- $x_1 x_2$ is considered as a single data object for declaring tolerances.
- Concurrent transactions must (for now) use the same groupings.

Conclusions and Further Directions

Conclusions:

CPSI & TCPSI: New levels of transaction isolation.

- Full constraints preservation: Both internal and extended constraints are fully preserved.
- $\geq$ SI isolation: Guarantees at least snapshot isolation.
- Simple checks: Two-at-a-time verification $\Rightarrow$ adaptable to dynamic changes of transaction reads and cooperation between transactions.

Further Directions:

Extend to write tolerances: Several transactions wish to withdraw from the same account concurrently.

$T_1: x \leftarrow x - 100$; $T_2: x \leftarrow x - 50$; Initial balance: $x = 500$.

- A protocol to support such concurrency is under development.

Use within a cooperative model: Rather than aborting transactions which conflicts occur, they may communicate and cooperate in order to proceed.

- It is a particularly attractive solution for interactive transactions.

More Information

Comprehensive slides: Slides (129 of them) entitled *Transaction models and concurrency control* from the course *Database System Principles* at Umeå University:
`http://www8.cs.umu.se/kurser/5DV120/V16/Slides/09_trans_5dv120_h.pdf`

- Also used at UdeC in the DB 1 course.

Research paper: Hegner, Stephen J., Constraint-preserving snapshot isolation, *Annals of Mathematics and Artificial Intelligence*, (**76**)2016, pp. 281–326.
`http://www8.cs.umu.se/~hegner/Publications/PDF/amai15.pdf`
`http://www8.cs.umu.se/~hegner/Publications/PDF/amai15_corr.pdf`

Research paper: Hegner, Stephen J., Tolerant constraint-preserving snapshot isolation: extended concurrency for interactive transactions, submitted for publication, 2017 (available upon request).