**the morning paper**

an interesting/influential/important paper from the world of CS every weekday morning, as selected by Adrian Colyer

# A Critique of ANSI SQL Isolation Levels

FEBRUARY 24, 2016

*tags:* Consistency, Transaction processing

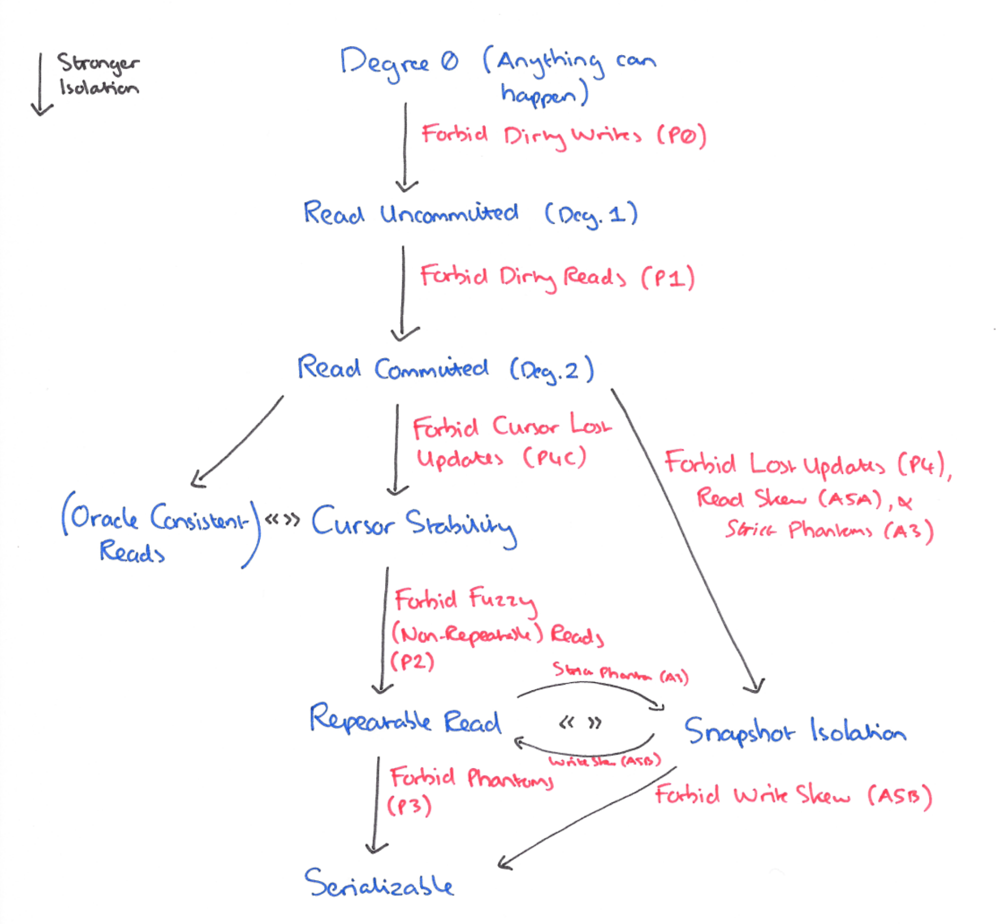
**A Critique of ANSI SQL Isolation Levels (http://arxiv.org/pdf/cs/0701157.pdf)** – Berenson et al. 1995

*udpate: 2 minor corrections in the section on A5A per the comment from 'banks' – thanks!*

The ANSI SQL isolation levels were originally defined in prose, in terms of three specific anomalies that they were designed to prevent. Unsurprisingly, it turns out that those original definitions are somewhat ambiguous and open to both a strict and a broad interpretation. It also turns out that they are not sufficient, since there is one even more basic anomaly not mentioned in the standard that needs to be prevented in order to be able to implement rollback and recovery. Looking even more deeply, the paper uncovers eight different phenomena (anomalies) that can occur, and six different isolation levels. I'm going to focus in this write-up on explaining those phenomena and isolation levels, and for those interested in the detailed comparison to the original ANSI SQL isolation levels I refer you to the full paper.

(*I hope you find all the history sketches in this post useful – it took me an age to draw and scan them all! You might find it easier to view slightly bigger versions of the images – simply click on the image to see an enlarged view.*)

Here's an overview map that shows how they all relate, that may be helpful as we work our way through.



**(https://adriancolyer.files.wordpress.com/2016/02/ansi-sql-isolation-levels.png)**

## Phenomena / Anomalies

In the description that follows, long duration lock are those held until the end of the transaction once taken, and short duration locks are released once the action involved completes. Furthermore,

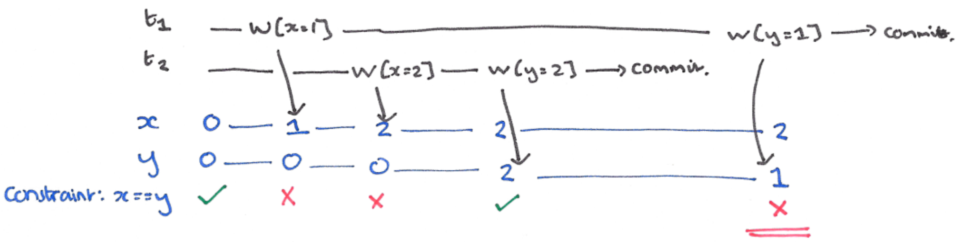
> *A transaction has well-formed writes (reads) if it requests a Write (Read) lock on each data item or predicate before writing (reading) that data item, or set of data items defined by a predicate. The transaction is well-formed if it has well-formed writes and reads. A transaction has two-phase writes (reads) if it does not set a new Write (Read) lock on a data item after releasing a Write (Read) lock. A transaction exhibits two-phase locking if it does not request any new locks after releasing some lock.*

## P0: Dirty Write

A *Dirty Write* occurs when one transaction overwrites a value that has previously been written by another still in-flight transaction.

> *One reason why Dirty Writes are bad is that they can violate database consistency. Assume there is a constraint between x and y (e.g., x = y), and T1 and T2 each maintain the consistency of the constraint if run alone. However, the constraint can easily be violated if the two transactions write x and y in different orders, which can only happen if there are Dirty writes.*

Suppose T1 writes x=y=1 and T2 writes x=y=2, the following history violates the integrity constraint.



**(https://adriancolyer.files.wordpress.com/2016/02/dirty-write.png)**

Another pressing reason to protect against dirty writes is that without protection from them the system can't automatically rollback to a before image on transaction abort.

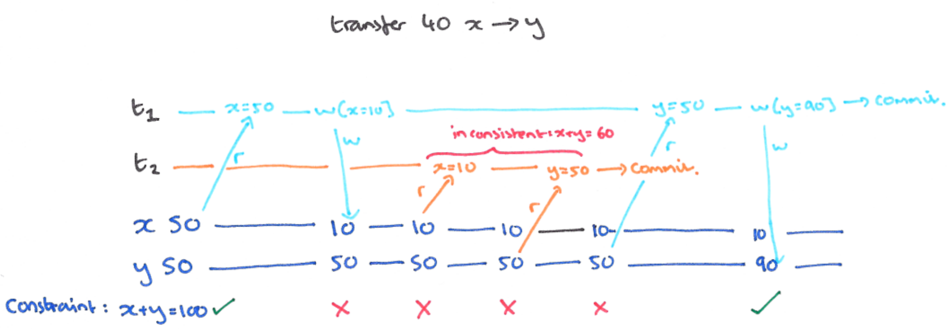
> *Even the weakest locking systems hold long-duration write locks…*

(Well-formed writes and long duration write locks prevent dirty writes).

Other than at the serializable level, ANSI SQL isolation levels do not exclude dirty writes. However, "ANSI SQL should be modified to require P0 (protection) for all isolation levels."

## P1: Dirty Read

A *Dirty Read* occurs when one transaction reads a value that has been written by another still in-flight transaction. It is not enough to prevent only reads of values written by transactions that eventually rollback, you need to prevent reads of values from transactions that ultimately commit too. Consider a transfer of balance between x and y, with an integrity constraint that the balance remains unchanged at 100:

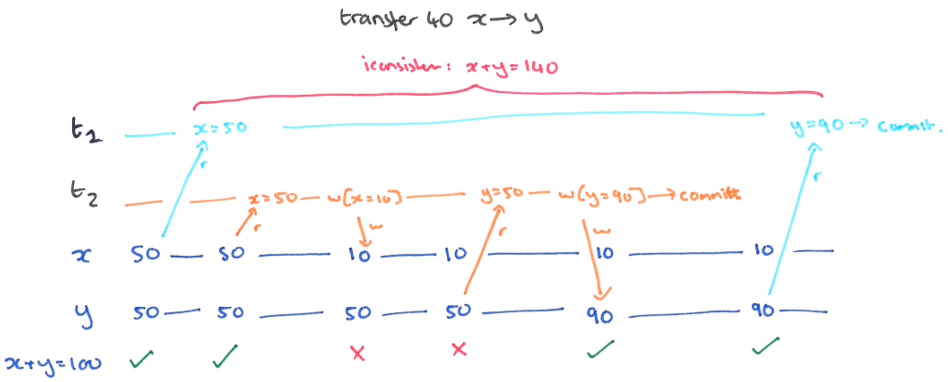

transfer 40 x → y
t1 — x=50 — w[x=10] — y=50 — w[y=90] → commit.
t2 — x=10 — y=50 → commit.
inconsistent: x+y = 60
r
w
x 50 — 10 — 10 — 10 — 10 — 10
y 50 — 50 — 50 — 50 — 50 — 90
Constraint: x+y=100 ✓ ✗ ✗ ✗ ✗ ✓

**(https://adriancolyer.files.wordpress.com/2016/02/dirty-read.png)**

In a locking-based implementation, well-formed reads and writes combined with holding short-duration read-locks and long-duration write-locks will prevent dirty reads.

## P2: Fuzzy Read (Non-Repeatable Read)

A *Fuzzy* or *Non-Repeatable* Read occurs when a value that has been read by a still in-flight transaction is overwritten by another transaction. Even without a second read of the value actually occurring this can still cause database invariants to be violated. Consider another history from our transfer example:

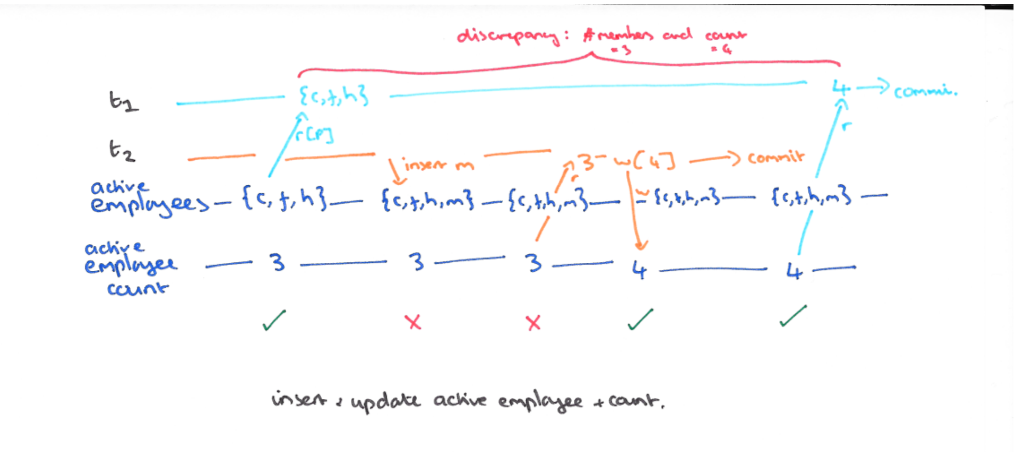


**(https://adriancolyer.files.wordpress.com/2016/02/fuzzy-read.png)**

In a locking implementation, well-formed reads and writes combined with long duration read and write locks for data items, but only short-duration locks for predicate-based reads will prevent non-repeatable reads.

## P3: Phantom

A *Phantom* occurs when a transaction does a predicate-based read (e.g. SELECT… WHERE P) and another transaction writes a data item matched by that predicate while the first transaction is still in flight. In the original ANSI SQL language only matching inserts were forbidden (the new entries being the phantoms), but in fact to be safe we need to prohibit any write: updates, deletes, and inserts. Consider the history below:

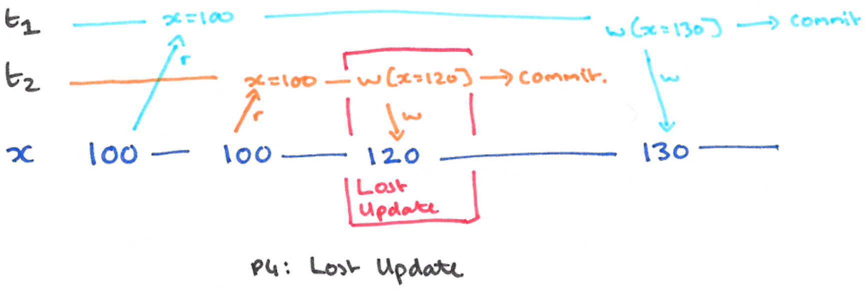


**(https://adriancolyer.files.wordpress.com/2016/02/phantom.png)**

In a locking-based implementation, well-formed reads and writes coupled with long-duration read locks (both data item and predicate), and long-duration write locks will prevent phantoms.

## P4: Lost Update

It turns out that there is an anomaly that is not prevented by systems that defend against Dirty Reads and Dirty Writes, but that *is* prevented by systems that also protect against Fuzzy reads. It's known as a Lost Update. Consider the following history:

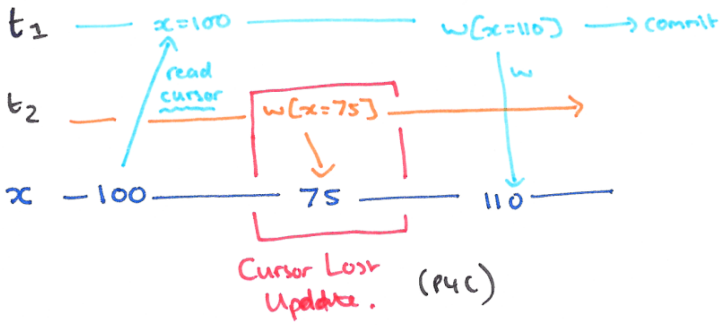


**(https://adriancolyer.files.wordpress.com/2016/02/lost-update.png)**

There is no Dirty Write since $T_2$ commits before $T_1$ writes x. There is no Dirty Read since there is no read after a write. Nevertheless, at the end of this sequence $T_2$‘s update will be lost, even if T2 commits.

## P4C: Cursor Lost Update

P4**C** is a variation of the Lost Update phenomenon that involves a SQL cursor. In the history below, let rc(x) represent a read of the data item x under the cursor, and wc(x) a write of the data item x under the cursor. If we allow another transaction T2 to write to x in between the read-cursor and write-cursor actions of T1, then its update will be lost.

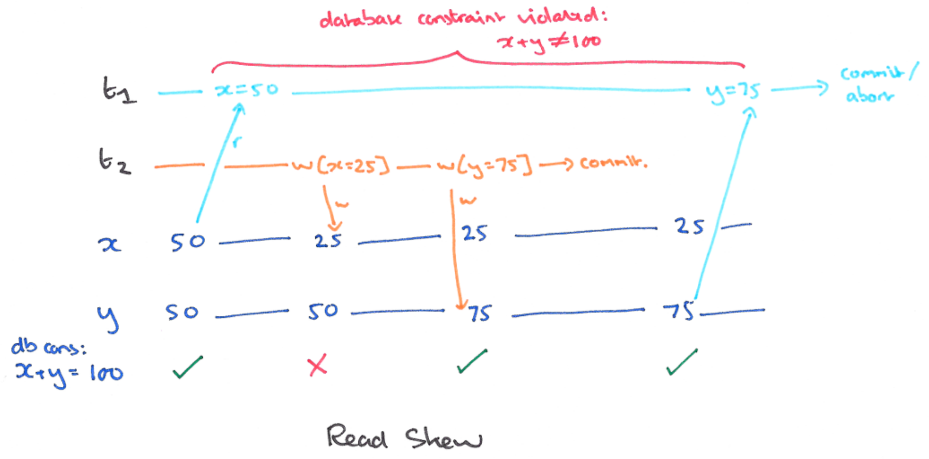


**(https://adriancolyer.files.wordpress.com/2016/02/cursor-lost-update.png)**

In a locking implementation, over above the requirements to prevent P0 and P1, holding a lock on the current item of the cursor until the cursor moves or is closed will prevent Cursor Lost Updates.

## A5A: Read Skew

Read skew can occur when there are integrity constraints between two or more data items. Consider the constraint x + y = 100, and the history:

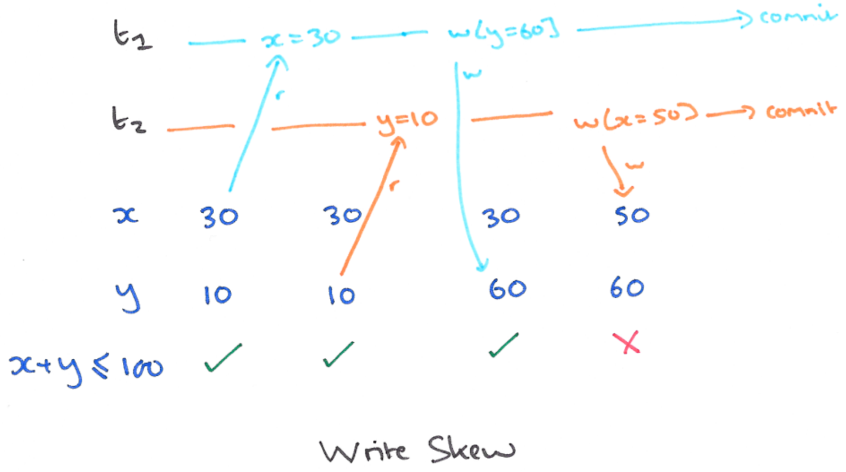


**(https://adriancolyer.files.wordpress.com/2016/02/read-skew.png)**

When T1 reads y it sees an inconsistent (skewed) state, and may therefore generate an inconsistent state as output.

## A5B: Write Skew

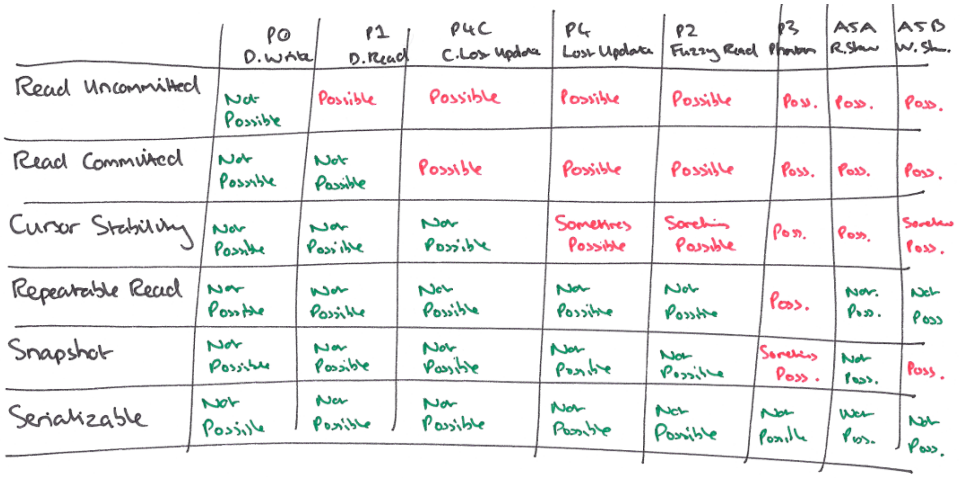
Write skew is very similar. Suppose our constraint is x+y ≤ 100. And this time we interpose a transaction before T1 issues a write:



**(https://adriancolyer.files.wordpress.com/2016/02/write-skew1.png)**

## Isolation Levels

We can characterise isolation levels by the phenomena they allow (prevent):



| | P0 D.Write | P1 D.Read | P4C C.Lost Update | P4 Lost Update | P2 Fuzzy Read | P3 Phantom | A5A R.Skew | A5B W.Skew |
|---|---|---|---|---|---|---|---|---|
| Read Uncommitted | Not Possible | Possible | Possible | Possible | Possible | Poss. | Poss. | Poss. |
| Read Commited | Not Possible | Not Possible | Possible | Possible | Possible | Poss. | Poss. | Poss. |
| Cursor Stability | Not Possible | Not Possible | Not Possible | Sometimes Possible | Sometimes Possible | Poss. | Poss. | Sometimes Poss. |
| Repeatable Read | Not Possible | Not Possible | Not Possible | Not Possible | Not Possible | Poss. | Not. Poss. | Not Poss |
| Snapshot | Not Possible | Not Possible | Not Possible | Not Possible | Not Possible | Sometimes Poss. | Not Poss. | Poss. |
| Serializable | Not Possible | Not Possible | Not Possible | Not Possible | Not Possible | Not Possible | Not Poss. | Not Poss. |

**(https://adriancolyer.files.wordpress.com/2016/02/isolation-levels-table.png)**

We can compare isolation levels based on the *non*-serializable histories that they allow:

> L1 is *weaker* than L2 if L1 permits non-serializable histories that L2 does not, and every non-serializable history under L2 is also a non-serializable history under L1. We write L1 << L2.
> L1 and L2 are equivalent if the sets of non-serializable histories permitted by them both are identical. We write L1 == L2.
> L1 and L2 may also be incomparable. If L1 permits a non-serializable history that L2 does not, and vice-versa, then L1 is not weaker than L2, but L2 is also not weaker than L1. We write L1 <> L2.

We can partially order isolation levels according to << :

Degree 0 (everything goes) << Read Uncommitted << Read Committed << Cursor Stability << Repeatable Read << Serializable.

There are two other isolation levels discussed in the paper, Oracle Consistent Read, and Snapshot Isolation. Like Cursor Stability, Read Committed << Oracle Consistent Read. Oracle Read Consistency isolation gives each SQL statement the most recent committed database value at the time the statement began.It is as if the start-timestamp of the transaction is advanced at each SQL statement. The members of a cursor set are as of the time of the Open Cursor. The underlying mechanism recomputes the appropriate version of the row as of the statement timestamp. Row inserts, updates, and deletes are covered by Write locks to give a first-writer-wins rather than a first committer-wins policy.

In contrast, in *snapshot isolation*,

> *…each transaction reads reads data from a snapshot of the (committed) data as of the time the transaction started, called its Start-Timestamp. This time may be any time before the transaction's first Read. A transaction running in Snapshot Isolation is never blocked attempting a read as long as the snapshot data from its Start-Timestamp can be maintained. The transaction's writes (updates, inserts, and deletes) will also be reflected in this snapshot, to be read again if the transaction accesses (i.e., reads or updates) the data a second time. Updates by other transactions active after the transaction StartT timestamp are invisible to the transaction.*

Both snapshot isolation and Oracle Consistent Read are types of multi-version concurrency control mechanisms, in which we have to reason about multiple different values of the same data item at any given point in time. It turns out that all Snapshot Isolation histories can be mapped to single-valued histories while preserving data dependencies.

Read Committed << Snapshot Isolation << Serializable.

But Snapshot Isolation and Repeatable Read are incomparable. Snapshot Isolation permits Write Skew, which Repeatable Read does not, and Repeatable Read permits some Phantoms which are not permitted under Snapshot Isolation.

*from* → Uncategorized

23 Comments leave one →

1. **Dan Koren PERMALINK**
   **February 24, 2016 7:11 am**
   While this is one of the key research papers in serializability theory and its importance cannot be underestimated, it is also outdated. More recent research over the past decade has found several snapshot isolation methods that satisfy serializability as well. These include serializable snapshot algorithms developed by Alan Fekete, Michael Cahill, Ryan Johnson and Steven Revilak.

   REPLY

   - **adriancolyer PERMALINK***
     **February 24, 2016 9:08 am**
     Hi Dan, many thanks for the pointers to follow-on research. I didn't know about the SSN paper so that's a great discovery for me :). So much of what follows refers back to today's paper (e.g. Adya) that I needed to cover it as a necessary foundation. I plan to cover a few more papers on these lines over the coming weeks (starting with Adya tomorrow!).

     For anyone interested in the papers Dan is referring to, here are some links:
     * Fekete: http://db.cs.berkeley.edu/cs286/papers/ssi-tods2005.pdf
     * Cahill: http://www.cs.nyu.edu/courses/fall09/G22.2434-001/p729-cahill.pdf
     * Revilak: http://www.cs.umb.edu/~srevilak/srevilak-dissertation.pdf
     * Wang & Johnson: dl.acm.org/ft_gateway.cfm?id=2771949&type=pdf

     REPLY

     - **banks PERMALINK**
       **February 24, 2016 10:29 am**
       In fact the design of CocroachDB is based on the Cahill SSI paper, but also references another one not included in that list: "A Critique of Snapshot Isolation" by Ferro and Yebandeh at Yahoo in 2012: https://drive.google.com/file/d/0B9GCVTp_FHJIMjJ2U2t6aGpHLTFUVHFnMTRUbnBwc2pLa1RN/edit

     - **dankoren PERMALINK**
       **February 24, 2016 6:21 pm**
       Oops! I meant "its importance cannot be overestimated"!
       I shouldn't be typing past midnight!
       Sorry,
       dk

2. **banks PERMALINK**
   **February 24, 2016 10:26 am**
   Fantastic overview as always Adrian – thank you.

   Yes the history diagrams were very useful; your hard work is appreciated :).

   I *think* I've spotted a couple of typos in your description of A5A: Read Skew that are minor but might just make it a little harder for someone to follow:

   – "when where are integrity" => "when there are integrity"
   – "When T2 reads y" => "When T1 reads y"

   Forgive the nitpick – would hate to see such a succinct and clear overview confuse future readers due to these minor issues!

   REPLY

- **adriancolyer PERMALINK***
  **February 24, 2016 10:31 am**
  Thank you – it's important to be accurate when dealing with topics such as this! I'll take a look when I get back to proper connectivity and post an update.

  REPLY
- **dankoren PERMALINK**
  **February 24, 2016 6:24 pm**
  Here's another diagram of serializability classes that might be useful:

  I always keep a copy in my cube next to my display! 😉

  REPLY
  - **dankoren PERMALINK**
    **February 24, 2016 6:25 pm**
    Not sure why the link did not stick, here's one more try:

3. **dankoren PERMALINK**
   **February 24, 2016 6:29 pm**
   Yoav Raz' commitment ordering method is also a
   key step in the evolution of serializability theory:
   https://en.wikipedia.org/wiki/Commitment_ordering

   REPLY
4. **Sen PERMALINK**
   **July 2, 2017 12:15 am**
   Thanks!

   PS: Could you please tell me what kind of tool you had been used to paint these beautiful diagram? (I guess maybe surface pen or apple pencil?)

   [diagram example](https://adriancolyer.files.wordpress.com/2016/02/ansi-sql-isolation-levels.png)

   REPLY
   - **adriancolyer PERMALINK***
     **July 2, 2017 7:02 am**
     Hi Sen, these particular diagrams were hand-drawn with pen and paper and then scanned. Nowadays I mostly use an app called Notability on an iPad Pro to achieve a similar effect. Regards, A.

     REPLY
5. **Etki PERMALINK**
   **August 29, 2017 11:55 pm**
   I'm having a feeling that short-duration read / long-duration write locks would not protect T2 in P1 example from reading y = 50 (before t1 commit) and then x = 10 (after t1 commit), still resulting in inconsistency.

   REPLY

## Trackbacks

1. 【はてなブックマーク】A Critique of ANSI SQL Isolation Levels | the morning paper – 気になるニュースをお届け♪
2. A Critique of ANSI SQL Isolation Levels | the morning paper – Temporary index
3. Generalized Isolation Level Definitions | the morning paper

4. ZeroDB и Nomad 0.3 — Episode 0081 « DevZen Podcast
5. Находки в сети за март 2016 | Записки программиста
6. Serializable, Lockless, Distributed: Isolation in CockroachDB | 神刀安全网
7. The SNOW theorem and latency-optimal read-only transactions | the morning paper
8. The many faces of consistency | the morning paper
9. Omid reloaded: scalable and highly-available transaction processing | the morning paper
10. ACIDRain: concurrency-related attacks on database backed web applications | the morning paper
11. Locks used in Database Management Systems | ao