Testing Isolation Levels of Relational Database Management Systems

Dimitrios Liarokapis

(dimitris@cs.umb.edu)

University of Massachusetts Boston

Overview

Isolation Levels have been introduced in RDBMS in order to increase performance when absolute concurrency correctness is not necessary or when correctness can be guaranteed at the application level.

The ANSI SQL standard has provided definitions for four isolation levels: **READ UNCOMITTED, READ COMMITED, REPEATABLE READ** and **SERIALIZABLE**.

There has been some critique in the literature, about the clarity and generality of these definitions. This raises concerns about the quality of the implementation of isolation levels by database vendors since it becomes more probable that the implementation of concurrency control could be sometimes incorrect or over-restrictive.

By incorrect we mean that a database system could allow executions that should be proscribed by a given isolation level, leading to an unintentional corruption of the database or to the return of incorrect information. By over-restrictive we mean that the database system would not allow executions that are not proscribed by the isolation level at use. This would generally lead to reduced performance.

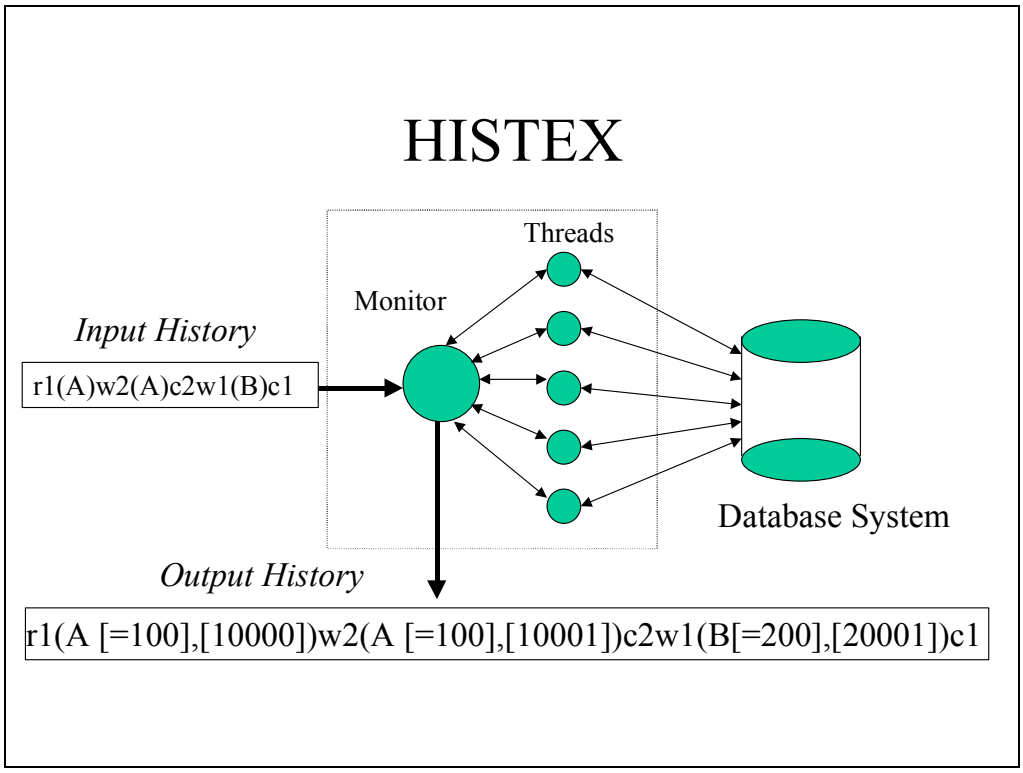
The general questions we try to answer are:

Does a given database system implement Isolation Levels correctly?

Can we design a tool and methodologies to test the support of isolation levels by database management systems?

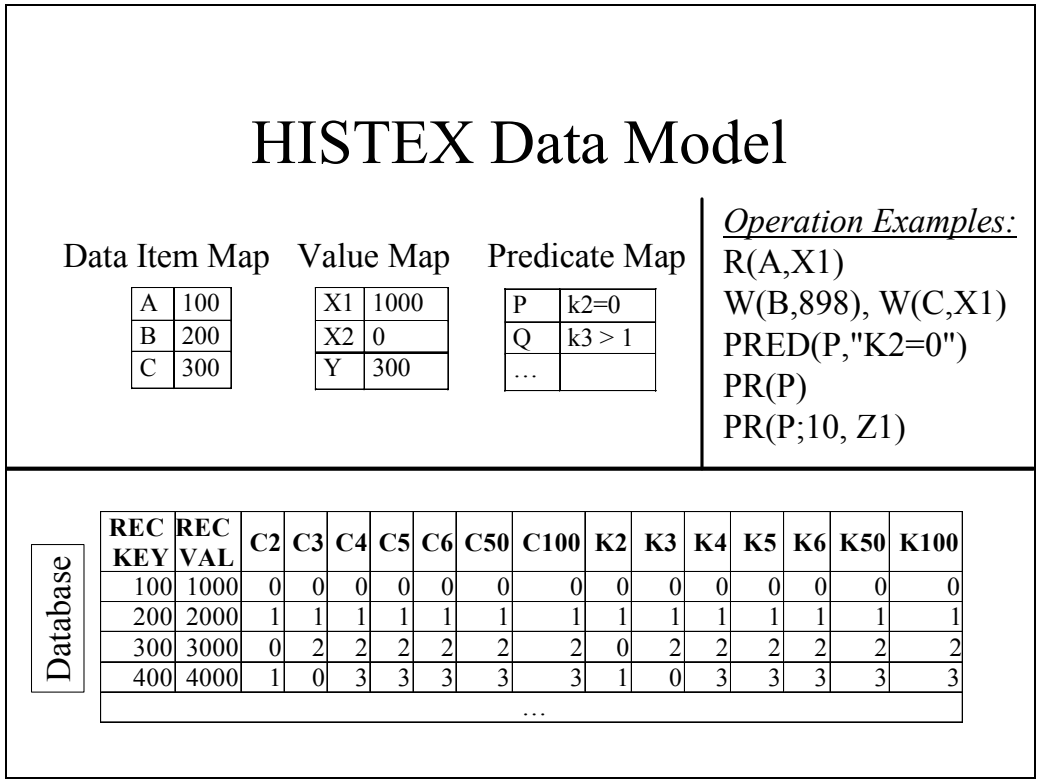
Tool & Methodologies

We have devised a notation similar to the one used in the Serializability Theory and a tool that executes sequences of operations (input histories) against commercial database applications and produces output histories.



In addition to the traditional operations for reading (R) and writing (W) data items and committing (C) or aborting (A) transactions, HISTEX supports a predicate read operation (PR) for accessing the rows that satisfy a predicate. This can be used for testing if a database system guards against phantoms (updates causing a row to enter or leave a set of rows satisfying a predicate read by another transaction).

Operations can include variable names to represent data items, values and predicates. HISTEX translates the input notation to SQL statements for accessing an underlying database table. Internal structures are used to map the variable names to specific rows in the table and hold the corresponding values.



Comparative Testing

One way for examining the correctness and precision of isolation level implementations is by using comparative testing: The same testing scenarios can be executed by different database products. The results are then compared and clustered according to their similarity.

When the results are different, it is an indication that one of the products might have demonstrated a problem. There can also be some special cases (single-version vs. multi-version systems) where even though the results are literally different the behavior of both the examined products is correct.

When the results are similar it is most likely that the testing scenarios do not reveal any problem. This is because the probability of many products demonstrating the same error is relatively small.

Gray Box Testing (Conditional Testing)

In general we could test the implementation of concurrency control by trying to force a database system to produce histories that should be proscribed by some isolation level. Such histories are those that contain a cycle in their serialization graph, when using the SERIALIZABLE level, or cycles of certain types at lower levels.

In order to simplify the creation and analysis of testing scenarios, we decided to utilize assumptions about the underlying implementation and test certain aspects of the underlying system. We call this method gray box testing.

We have focused on gray-box testing of database systems that are known to use single-version concurrency control algorithms based on preventing concurrent execution of conflicting operations. This is usually achieved by locking.

We have proven a theorem showing that for testing these types of schedulers it is adequate to test whether each isolation level proscribes the execution of certain pairs of conflicting operations.

Isolation Level	**Proscribed Pairs of Concurrent Operations**
Read Uncommitted	Transactions are READ ONLY. There should not be concurrent conflicting operations.
Read Committed	$W_1(A)W_2(A)$ $W_1(A)R_2(A)$ W_1(A changes P) $PR_2(A)$
Repeatable Read	All above and $R_1(A)$ $W_2(A)$
Serializable	All above and $PR_1(A)$ W_2(A changes P)

Results

We have executed histories including all different types of conflicting pairs of operations against 3 commercial database systems A, B and C (A and B were successive versions of the same database system) and we have observed several interesting results.

One major finding is related to the results received for the following histories:

History Name	Type of Conflict
h.14.w_pr	W_1(A out of P) …PR_2(P)
h.17.w_pr	D_1(A in P)…PR_2(P)

(D indicates a Delete operation)

For the database system A, the isolation level that corresponds to the READ COMMITTED level in the ANSI SQL specification, would allow a transaction to observe an uncommitted state of the database.

This was happening in cases where a transaction T_1 would force a row out of a predicate P, and before this transaction committed another transaction T_2 accessing the rows in predicate P would not see this row.

Output file	prim-key index	prim-key no-index	no-prim-key index	no-prim-key no-index
h.14.w_pr.RC_RC	EXECUTED*	TIMEOUT	EXECUTED*	TIMEOUT
h.14.w_pr.RC_RR	EXECUTED*	TIMEOUT	EXECUTED*	TIMEOUT
h.14.w_pr.SR_RC	EXECUTED*	TIMEOUT	EXECUTED*	TIMEOUT
h.14.w_pr.SR_RR	EXECUTED*	TIMEOUT	EXECUTED*	TIMEOUT
h.14.w_pr.RR_RC	EXECUTED*	TIMEOUT	EXECUTED*	TIMEOUT
h.14.w_pr.RR_RR	EXECUTED*	TIMEOUT	EXECUTED*	TIMEOUT
h.17.w_pr.RC_RC	EXECUTED*	EXECUTED*	EXECUTED*	EXECUTED*
h.17.w_pr.RC_RR	EXECUTED*	EXECUTED*	EXECUTED*	EXECUTED*
h.17.w_pr.SR_RC	EXECUTED*	EXECUTED*	EXECUTED*	EXECUTED*
h.17.w_pr.SR_RR	EXECUTED*	EXECUTED*	EXECUTED*	EXECUTED*
h.17.w_pr.RR_RC	EXECUTED*	EXECUTED*	EXECUTED*	EXECUTED*
h.17.w_pr.RR_RR	EXECUTED*	EXECUTED*	EXECUTED*	EXECUTED*

This behavior has not been observed in the successive version of the same database product (B).

We have also noticed that a transaction running at the READ UNCOMMITTED isolation level on systems A and B can perform updates. In general this is disallowed by the ANSI SQL specification, in order to eliminate the risk of performing an update based on non-committed information.

In addition to incorrect behavior our analysis also detected cases where the underlying database systems behaved over-restrictively.

This work was done in cooperation with Professors: Elizabeth O'Neil (Thesis Advisor) and Patrick O'Neil.

For more info please visit: www.cs.umb.edu/~dimitris/thesis
Or contact : dimitris@cs.umb.edu