Testing Isolation Levels of Relational Database Management Systems

A Dissertation Presented

by

Dimitrios Liarokapis

Δημήτριος Αλεξάνδρου Λιαροκάπης

Submitted to the Office of Graduate Studies and Research, University of Massachusetts Boston, in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2001

Computer Science Program

Testing Isolation Levels of Relational Database Management Systems

A Dissertation Presented

by

Dimitrios Liarokapis

Approved as to style and content by:

ABSTRACT

Testing Isolation Levels of Relational Database Management Systems

December 2001

Dimitrios Liarokapis,

B.S., University of Patras, Greece
M.S., University of Massachusetts Boston
Ph.D., University of Massachusetts Boston

Directed by Professor Elizabeth O'Neil

This thesis studies approaches for testing and understanding the implementation of Isolation Levels by Relational Database Management Systems. Isolation is the property of these systems that ensures that concurrent transactions do not interfere. The Isolation Levels feature has been developed to improve performance when full correctness is not desirable or when correctness could be assured at the application level. The definitions of Isolation Levels, below the serializable level, are evolving and there are not yet well-accepted general definitions.

A tool and methodology for testing whether a system behaves according to some definition, or for understanding the exact behavior of a given system, is very important because very critical applications use this technology, and problems in implementations or misuse of Isolation Levels could have very adverse side-effects (e.g. money being lost, collisions in reservations, bad estimates etc). The tool we have developed processes specifications of concurrency scenarios, called input histories and produces output histories at various database Isolation Levels. The input histories are made up of a series of generically specified transactional operations for several concurrent transactions. By analyzing the results in the output history, it can be determined whether the execution was correct under a given isolation level or levels.

We introduce two methodologies: comparative testing and gray box testing and we focus 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 prove 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. We have executed histories including all different types of conflicting pairs of operations. Among interesting results, we have detected that the isolation level of a particular version of a database system that corresponds to the READ COMMITTED ANSI SQL would allow execution of a certain type of conflicting operations (write/predicate-read) that should be proscribed. The results we have generated demonstrate the utility of our methodology.

TABLE OF CONTENTS

1. Introduction

Τιμή σ' εκείνους όπου στην ζωή των
όρισαν και φυλάγουν Θερμοπύλες.
Ποτέ από το χρέος μη κινούντες·
δίκαιοι κ' ίσιοι σ' όλες των τες πράξεις,
αλλά με λύπη κιόλας κ' ευσπλαχνία·
γενναίοι οσάκις είναι πλούσιοι, κι όταν
είναι πτωχοί, πάλ' εις μικρόν γενναίοι,
πάλι συντρέχοντες όσο μπορούνε·
πάντοτε την αλήθεια ομιλούντες,
πλην χωρίς μίσος για τους ψευδομένους.
Και περισσότερη τιμή τους πρέπει
όταν προβλέπουν (και πολλοί προβλέπουν)
πως ο Εφιάλτης θα φανεί στο τέλος,
κ' οι Μήδοι επι τέλους θα διαβούνε.

Κωνσταντίνος Π. Καβάφης (1903)

Honor to those who in their lives
have defined and guard their Thermopylae.
Never stirring from duty;
just and upright in all their deeds,
yet with pity and compassion too;
generous when they are rich, and when
they are poor, again a little generous,
again helping as much as they can;
always speaking the truth,
yet without hatred for those who lie.
And more honor is due to them
when they foresee (and many do foresee)
that Ephialtes will finally appear,
and that the Medes in the end will go through[1].

Constantine P. Cavafy (1903)

Relational Database Management Systems have become the predominant technology of the majority of information systems today. They are used to store, retrieve and update information for a variety of applications such as banking accounts, airline reservation systems, inventories, stock exchanges etc.

A very important aspect of Database Management Systems is *concurrency control*. This is considered to be part of what is known as *transaction management* and *crash recovery* is considered to be the other part. Concurrency control is responsible for assuring that concurrent access of the database by different users will be done properly. Metaphorically, the importance of concurrency control for database systems is similar to the importance of traffic lights in a traffic system. Their absence would cause either collisions in case drivers were very optimistic or jams and deadlocks if they were very pessimistic.

[1] In 480 BC, Xerxes led the Persians (*Medes*) and invaded Greece. The Spartan king Leonidas, in charge of 7000 Greeks, was ordered to cut the advance of the Persian army at *Thermopylae* (in central Greece), a narrow strip of land between the sea and impassable mountains. The Persian army, 250,000 strong, attacked twice and was forced to retreat, due to the fact that the passage was so narrow that they could not fully deploy their force. However, an avaricious local farmer, *Ephialtes*, led a force of Persian infantry through a mountain passage and next morning they appeared behind the Greek lines. Leonidas ordered the rest of the army withdraw, and held the passage with 300 Spartans. As a true Spartan, he chose death over retreat; all 300 Spartans, including Leonidas, died, but held the Persians long enough to ensure the safe withdrawal of the rest of the Greek army. (Poem and notes were retrieved from: http://alexandria.sae.gr/html/thermopylae.html)

Originally, concurrency control was envisioned to provide complete *isolation* to transactions accessing a database but there was a desire in the industry for providing the ability of reducing this restriction in a controlled fashion in order to increase performance. This need brought in the introduction of different *isolation levels* that a database system could provide. Metaphorically, this is similar to the way traffic lights are sometimes switching to a blinking mode, indicating that responsibility for the decision on whether to pass or not is transferred to the drivers.

The definitions of isolation levels are still in evolution. The ANSI (American National Standards Institute) SQL[2] [SQL92, SQL99] standard, which has also become an ISO (International Standards Organization) standard, has provided definitions that have been highly criticized for their clarity and generality [BBMGOO95, ALO00]. 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 in this work are: "Does a given database system implement concurrency control correctly?"; or more specifically, "Does it implement Isolation Levels according to the ANSI SQL [SQL99] specification?"; "Can we design a tool and a methodology to test the support of isolation levels by database management systems?"

For developing the tool and the testing methodologies we are describing, we have borrowed the basic ideas from the theory that is being developed to examine the correctness of concurrency control provided by database management systems [GLTP77, GR93, BHG87, PAPA86, BBGMOO95, ALO00]. The general approach for theorizing about concurrency is utilizing an abstract data and operations model consisting only of the essential parts of a database system that pertain to concurrency control. These were mainly a set of data items and a set of basic operations for reading and updating the data items and committing or aborting transactions. Operations for processing the data values read or computing the data values to be written were considered less essential. The *scheduler* of a database management system was responsible to receive concurrent operations of several transactions and produce a sequence of the same operations in such a way that the execution would be equivalent to a serial execution of the same transactions. There were criteria developed to decide whether an output history was correct or not.

One way to develop a testing methodology would be to devise a notation for creating testing scenarios and a system that could try to exercise the concurrent behavior of a

[2] SQL (Structured Query Language) is the standard language that is used by all the modern RDBMSs for defining and manipulating data.
[3] This work started as part of the [ISO97] research project.

database system. The notation could be similar to the notation used in the concurrency control literature and the system would try to map the data and operations deployed by that notation to actual data representations and commands of a real database system. A preliminary specification of such a system was introduced in [L98] and several extensions have been proposed in [O98]. The main extensions in [O98] had to do with introducing a predicate read operation (in addition to data item operations) and several types of memory variables that could hold record identifiers, data values and information about predicates. A working version of this tool, called HISTEX (HISTory EXerciser) is already implemented and is based on some of the ideas appearing in [L98, O98].

Originally, the tool has been used to perform a good deal of ad-hoc examination of the correctness and precision of isolation levels in commercial database systems[4]. This experimentation ended up introducing a testing methodology [L98b] that we can call *comparative* testing. The idea is quite simple. The same testing scenarios can be executed by different database products. The results could then be compared and clustered according to their similarity. When the results are similar it is very probable that the testing scenarios would not reveal any problem. This is because the probability of many products demonstrating the same error is relatively small. In addition, if the results are identical the clustering will considerably reduce the amount of outputs that will need to be analyzed.

When the results are different, it is an indication that one of the products might have demonstrated a problem. There can also be cases where even though the results are literally different the behavior of both the examined products is correct. In the testing done at [L98b] such cases were due to the fact that the tested products are using very different approaches in implementing concurrency control (single-version vs. multi-version). For example, in a single-version system which utilizes locking for concurrency control[5], a transaction would try to acquire a read lock on some data item in order to prevent some other transaction from updating the same data (before the first transaction commits), and this would cause the second transaction to wait. For the same scenario, in a database system that implements a form of multi-version concurrency control[6], the first transaction, in general, would not acquire any lock, and the concurrency problem is solved because the second transaction will not directly update the data item that was read, but will create a new version of it.

To explain the difference between single-version systems and multi-version systems in a figurative way, it is to say that single version systems are using traffic lights where multi-version systems are using overpasses. Traffic lights are usually used where the road system is restricted to a single-plane and overpasses where it is possible to use an additional dimension for avoiding collisions.

In this dissertation we will be describing the design and implementation of HISTEX and of the experience we have gained up to now including a systematic methodology to

[4] IBM DB2 UDB version 5.0 and ORACLE version 8.1.6
[5] Such a system is IBM DB2 (versions 7.0 and earlier)
[6] Such s system is Oracle (versions 8.x)

validate SQL concurrency behavior at various isolation levels in database systems that use a locking approach to isolation.

The methodology we have used in this dissertation could be classified as *gray box* testing compared to the more standard methodologies such as *black box* testing and *white box* testing that can be found in the software testing literature . The objective of black box testing is to consider the software system as a black box without any knowledge of the internal implementation, and to test whether it behaves according to the functional requirements. In the white box (or glass box) testing, the approach is to verify the correctness of the software by analyzing the source code. This means that for using white box testing someone should have access to the sources. This could not happen in our case because the software we are testing is commercial applications and the sources are not available.

The term *gray box* testing has appeared already in the literature, but it is not well defined. In our case, we mean that when developing a testing methodology we are also considering assumptions about the underlying implementation. Such knowledge is available because for several database products there are publications that provide descriptions about the basic algorithms that are used for concurrency control. In this way we are able to focus on testing certain aspects of the underlying system and this could significantly reduce the combinatorial space of the testing cases that should be developed for a general black box approach. This makes the process more manageable and reproducible and more pertinent for implementing regression tests because these should be usually executed in a limited time period, and it is desirable that the test cases should be as representative as possible.

We have used the tool and our methodology quite successfully. One major finding was that in the database system IBM DB2 UDB V5.0 the isolation level Cursor Stability, which corresponds to the READ COMMITTED isolation 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 the updated row. This behavior is not correct according to the ANSI specification because T_2 should observe only a committed state of the database.

We have noticed also that it is possible that a transaction running at the Uncommitted Read isolation level on an IBM DB2 database system can perform updates. According to ANSI SQL specification, transactions that run at this level should be disallowed to perform updates in order to eliminate the risk of creating an update based on non-committed information.

In addition, HISTEX could be used by Database Administrators and students to study the concurrency behavior of database systems. This is important because currently there is not a wide understanding about the implications of using isolation levels. Towards this end a new tool and service has been envisioned, namely HistexOnline [L00] and a current release has already been implemented[7]. The intent of this tool is to make HISTEX

[7] The implementation of HistexOnline was done as a Master's Software Engineering Project at UMASS

accessible through the world wide web and to provide the scientific community the ability to experiment and learn about database concurrency control with a minimal effort, since HistexOnline obliterates the tedious steps of multiple software installations that are required and the consequent maintenance cost and effort.

An example of an area that should be understood by database system practitioners are the implications of the default isolation level that are supported by several RDBMSs[8]. These are usually set to levels lower than the strictest one in order to improve performance. However it can be demonstrated that under these circumstances it is possible that an application will not work correctly. This could have serious implications because database systems currently constitute the back bone of many commercial and governmental information systems and this type of application errors could result in bad transactions causing money being lost etc.

The tool we have devised could be also used for regression testing of the concurrency control algorithms utilized by database systems. These algorithms are usually quite complex and often of a heuristic nature [MOHAN90]. The complexity involved increases the risk of inadvertent mistakes or omissions in the implementation. Furthermore we expect that the concurrency control mechanisms supported by database vendors will undergo changes and refinements. Some indications of this are the differences we have noticed when we run our tests in two successive versions of the same database product, namely IBM UDB DB2 v5.0 and IBM UDB DB2 v6.1. A good and systematic regression test suite could help in coping with the risks that accompany software modifications and thus assist in the software improvement.

A legitimate question about our methodology could be: “How are we sure that the problems we have discovered are real and not due to possible errors in the tool we have utilized?” What we would recommend as an additional verification would be to recreate similar scenarios not by using our methodology and tool, but rather the tested database system directly. Based on the histories (i.e. testing scenarios) we have used to reveal some problem, it is possible to describe operational steps that can be followed by directly accessing a database management system in order to reproduce the same behavior.

Finally, we would like to mention that in addition to testing we have also used some features of HISTEX to take performance measurements related to concurrency control for a joint paper, which is currently in preprint form, [FLOOS00].

Boston, by Hui Gong, Zhiming Mai, Jian Pan, Sasaki Sui; their instructor was Prof. K. Newman.

[8] For example the default isolation level in ORACLE 8.1.X is the Read Committed rather than the Serializable and for IBM DB2 it is the Cursor Stability rather than the Repeatable Read (the name DB2 is used for what is called Serializable by ANSI SQL.

Σα βγεις στον πηγαιμό για την Ιθάκη,
να εύχεσαι νάναι μακρύς ο δρόμος,
γεμάτος περιπέτειες, γεμάτος γνώσεις.
Τους Λαιστρυγόνας και τους Κύκλωπας,
τον θυμωμένο Ποσειδώνα μη φοβάσαι,
τέτοια στον δρόμο σου ποτέ σου δεν θα βρείς,
αν μέν' η σκέψις σου υψηλή, αν εκλεκτή
συγκίνησις το πνεύμα και το σώμα σου αγγίζει.
Τους Λαιστρυγόνας και τους Κύκλωπας,
τον άγριο Ποσειδώνα δεν θα συναντήσεις,
αν δεν τους κουβανείς μες στην ψυχή σου,
αν η ψυχή σου δεν τους στήνει εμπρός σου.
Να εύχεσαι νάναι μακρύς ο δρόμος.
Πολλά τα καλοκαιρινά πρωϊά να είναι
που με τι ευχαρίστησι, με τι χαρά
θα μπαίνεις σε λιμένας πρωτοειδωμένους·
να σταματήσεις σ' εμπορεία Φοινικικά,
και τες καλές πραγμάτειες ν' αποκτήσεις,
σεντέφια και κοράλλια, κεχριμπάρια κ' έβενους,
και ηδονικά μυρωδικά κάθε λογής,
όσο μπορείς πιο άφθονα ηδονικά μυρωδικά·
σε πόλεις Αιγυπτιακές πολλές να πας,
να μάθεις και να μάθεις απ' τους σπουδασμένους.
Πάντα στον νου σου νάχεις την Ιθάκη.
Το φθάσιμον εκεί είν' ο προορισμός σου.
Αλλά μη βιάζεις το ταξίδι διόλου.
Καλλίτερα χρόνια πολλά να διαρκέσει·
και γέρος πια ν' αράξεις στο νησί,
πλούσιος με όσα κέρδισες στον δρόμο,
μη προσδοκώντας πλούτη να σε δώσει η Ιθάκη.
Η Ιθάκη σ' έδωσε το ωραίο ταξίδι.
Χωρίς αυτήν δεν θάβγαινες στον δρόμο.
Αλλο δεν έχει να σε δώσει πια.
Κι αν πτωχική την βρεις, η Ιθάκη δεν σε γέλασε.
Ετσι σοφός που έγινες, με τόση πείρα,
ήδη θα το κατάλαβες η Ιθάκες τι σημαίνουν.

Κωνσταντίνος Π. Καβάφης (1911)
http://alexandria.sae.gr/html/ithaca-gr.html

When setting out upon your way to Ithaca,
wish always that your course be long,
full of adventure, full of lore.
Of the Laestrygones and of the Cyclopes,
of an irate Poseidon never be afraid;
such things along your way you will not find,
if lofty is your thinking, if fine sentiment
in spirit and in body touches you.
Neither Laestrygones nor Cyclopes,
nor wild Poseidon will you ever meet,
unless you bear them in your soul,
unless your soul has raised them up in front of you.
Wish always that your course be long;
that many there be of summer morns
when with such pleasure, such great joy,
you enter ports now for the first time seen;
that you may stop at some Phoenician marts,
to purchase there the best of wares,
mother-of-pearl and coral, amber, ebony,
hedonic perfumes of all sorts--
as many such hedonic perfumes as you can;
that you may go to various Egyptian towns
to learn, and learn from those schooled there.
Your mind should ever be on Ithaca.
Your reaching there is your prime goal.
But do not rush your journey anywise.
Better that it should last for many years,
and that, now old, you moor at Ithaca at last,
a man enriched by all you gained upon the way,
and not expecting Ithaca to give you further wealth.
For Ithaca has given you the lovely trip.
Without her you would not have set your course.
There is no more that she can give.
If Ithaca seems then too lean, you have not been deceived.
As wise as you are now become, of such experience,
you will have understood what Ithaca stands for.

Constantinos Kavafis (1863-1933)
Translated by Memas Kolaitis

2. Concepts and Background

A database system maintains information in each database about a real-world enterprise, and provides functionality to query and update this information. It interacts with the users through atomic units of work called transactions. A *transaction* consists of a group of operations to read and update a database that the programmer wants to have succeed or fail as a unit, isolated from operations of concurrent transactions that might interfere. *Concurrency* is the ability of the database system to allow the execution of multiple transactions with interleaved operations. One of the main advantages of concurrency is performance improvement. The performance gains are due to the fact that some components in the memory hierarchy (i.e. disks) have very slow access time compared to others (i.e. main memory). If only one transaction is active at a time then the CPU will remain unutilized whenever a disk access occurs. If concurrency is allowed, then the CPU could be utilized by a different transaction that is starting or that has completed a disk access and is ready to run once more.

Uncontrolled concurrent execution of transactions can lead to a database state that is not consistent with what would happen if the transactions were executed in serial order. A classic example to illustrate how this can frustrate the intention of the programmer is a scenario of two transactions T_1 and T_2 accessing two database accounts A and B. T_1 transfers \$10 from A to B and T_2 deposits \$50 in account B. Initially the accounts A and B contained the amounts \$100 and \$50 respectively. We expect that after the two transactions are complete, account A would contain \$90 and account B \$160. We can assume that T_1 operates in the following steps:

S_{11}: Read(A, V1)
S_{12}: Write(A, V1 - 10)
S_{13}: Read(B, V2)
S_{14}: Write(B, V2 + 10)

And T_2 operates in the following steps:

S_{21}: Read(B, V3)
S_{22}: Write(B, V3 + 50)

It is easy to verify that if all operations of transaction T_1 and T_2 were executed in a serial order, i.e. all operations of T_1 followed by all operations of T_2 or vice-versa, then the result of the execution would be the expected one. But in an uncontrolled concurrent execution, the steps of the two transactions could be executed in the following interleaved order:

S_{11}: Read(A, V1=100)
S_{12}: Write(A, V1 - 10 = 90)
S_{13}: Read(B, V2 = 50)
S_{21}: Read(B, V3 = 50)
S_{22}: Write(B, V3 + 50 = 100)

S_{14}: Write(B, V2 + 10 = 60)

The final state of the database would be such that the amounts in accounts A and B were \$90 and \$60 respectively. In other words, the effects of transaction T_2 setting B to 100 were lost. Proper isolation requires the database system to provide some control to disallow scenarios such as this.

2.1 Serializability

A database system contains a component, called a *scheduler*, which governs the interleaved execution of transaction operations. A scheduler receives operations from user transactions and ensures that they will be executed in such a way that the execution will be *correct.* We call the sequence of the operations as they are received by the scheduler the *input history*. The operations of the input histories we study here are more general than the generic history operations introduced in such classic works as [BHG87].

A *locking scheduler*, which we will study closely in what follows, has the capability to ensure correctness by changing the order of the input operations by delaying them before they are executed, and even to abort some transactions that become involved in a *deadlock*. It is generally assumed that such aborted transactions will later be *retried* by the user application, so all required work will eventually be carried out. Correctness has been taken to mean that the series of operations executed - the *output history* - constitutes a *serializable* execution, one that demonstrates the same effects with a serial execution of the same transactions.

Serializability theory has been developed in order to provide more compact criteria for deciding whether a history is serializable. Otherwise, the intuitive definition we have just provided would imply that in order to determine whether a history is serializable someone would have to execute all possible combinations of serial histories until one (if any) that produces the same results is reached.

The concept of *equivalence* was introduced in order to provide syntactical rules that could be used in transforming one history to another that would demonstrate the same effects. If by applying these rules we can reach a serial history, then the original history would be serializable.

There are actually two main and well-accepted theories that have been developed in the database concurrency control literature for defining history equivalence and for deciding whether a history is serializable, namely: *conflict serializability* [BHG87] and *view serializability* [PAPA79]. Conflict serializability is mainly used for single version database systems whereas view serializability is mainly used for multi-version database systems.

In conflict serializability, adjacent and non-conflicting operations belonging to different transactions could be swapped in the history, and yet the history would produce the same result. Two operations are considered to *conflict* if they both access the same data item and one of them is a write. If the order of execution of two conflicting operations changes, the end result will be different.

The order of conflicting operations establishes precedence *dependencies* between the transactions that contain those operations. A *Serialization Graph* [BHG87, OO00] for a history H is a directed graph SG(H) that captures these dependencies. The graph contains one node for every committed transaction T_i in the history, and there is a directed edge from node T_i to node T_j if and only if there are two conflicting operations P_i and Q_j in H, such that P_i comes prior to Q_j.

It can be shown that two histories are conflict-equivalent if their serialization graphs are identical. A history is serializable if there is no cycle in its serialization graph.

A locking scheduler that receives an operation Q_j conflicting with a prior operation R_i of an uncommitted transaction Ti will attempt to block Q_j from executing until T_i commits or aborts.

Classical transactional notation contains two main operations for accessing the database *Reads* (R) and *Writes* (W). In HISTEX (the tool we have developed for testing the implementation of database concurrency control) however, we are extending the possible operations to include Insert (I), Delete (D), and ReadWrite (RW) – Read followed indivisibly by Write. Basically we still classify all operations that perform updates as performing Writes (W, I, D, RW), and say that all other data access operations (R, PR, see Section 3) perform Reads.

2.2 Predicates and the Phantom Problem

Operations on single data items are not the only ones that a database system can support. There are operations that operate on a group of items that satisfy a *predicate*. In relational database systems supporting SQL, the predicates that determine the set of items to be operated on are provided in the WHERE clause of the SELECT or UPDATE statements.

When an application accesses a group of items that satisfy a predicate, this can lead to new consistency violations. This category of violations is known as the *phantom problem* [EGLT76, GR93], and result from conflicts that are not captured by the notion of the conflicting pairs of operations on single data items. These are called *predicate conflicts*. A predicate operation PR, which determines the set of rows satisfying a WHERE predicate, conflicts with any other operation that modifies the set of rows satisfying the predicate by causing some row to enter or leave this set.

An example of a predicate conflict can occur between two transactions T_1 and T_2, where T_1 computes the sum of balances for all the rows in an accounts table for a given bank branch, and T_2 enters a new account row into that branch. An insert of a new account does not conflict with any operation that has been applied to an existing row in accounts, but it conflicts with the operation that computes the sum of the balances in the branch. This is because it affects the range over which the sum of balances is calculated. If the insert and the evaluation of the sum were executed in the reverse order, the result would be different.

[EGLT76] introduced the concept of *predicate read* and *predicate write* locks in order for a locking scheduler to guard against predicate conflicts. These were to be acquired by operations that would select or update the set of rows specified by a predicate. It was shown later in [JBB81] that a database system does not need to support two types of predicate locks (read and write) in order to avoid the phantom problem. Only predicate read locks are essential among predicate locks. In discussions of predicate locks, we are always assuming data-item locks are fully in use. The predicates in [JBB81] should follow the restriction that given a tuple *t* and a predicate *P*, it must be possible to decide whether *t* satisfies *P* without using any other information. A predicate write (we will also call it a *set update*) can be viewed as a sequence of operations making up a predicate read and followed by updates of individual rows that are contained in the set of rows that satisfy the predicate. No modern database system uses the old style of predicate write locks since they were shown to be inappropriately restrictive of concurrency [CHETAL81].

The operations supported by HISTEX, which will be described in Section 3, were chosen to reflect modern practice. In addition to the operations that act on data items, HISTEX supports an operation PR(P), which performs a predicate evaluation of the predicate P.

We could express predicate conflicts by using the abstract notation:

$$W_1(\text{A changes P})\ PR_2(P)$$

The first operation – W_1(A changes P) – denotes an update of the data item A such that either the data item was *satisfying* the predicate P before the update and it does not satisfy it after the update, or vice versa. Such changes in a Relational Database could result either by inserting or updating a row to make it enter the set of rows satisfying the predicate P, or by deleting or updating a row so that it is removed from the set of rows satisfying P. These particular update operations could be expressed by the notation W(A into P) and W(A outof P), respectively. The Delete and Insert operations could be expressed by using the notations I(A in P) and D(A in P), respectively.

However, the actual HISTEX notation is not this abstract. It is up to the author of the histories to arrange the conflicts appropriately.

2.3 Isolation Levels

As it has already been indicated, a correct scheduler is one that produces serializable output histories: histories that are equivalent to some serial schedule. In a locking scheduler, this can be achieved by blocking conflicting operations to delay them until all transactions with conflicting operations have either aborted or committed, and by aborting transactions to deal with deadlocks. But these remedies to guarantee serializability can adversely affect performance by lowering transactional throughput. It is pointed out in [BHG87], Section 3.12, that the blocking effect of locking, as the number of contending transactions increases, causes most of the performance loss, rather than transactional aborts, which are relatively rare by comparison.

There has been a desire in the database industry [GLPT77, SQL92, SQL99] to weaken the requirement that a system must enforce perfect serializability, in order to increase the performance of common multi-user applications in cases where: (1) absolute correctness is not critical, or (2) the expected transactional workload is such that the application threads will execute correctly even if the underlying system does not employ all the mechanisms for ensuring correct behavior.

The ANSI SQL Standard [SQL92, SQL99] defines four isolation levels. The definition is expressed in terms of the following phenomena that different isolation levels should proscribe in output histories.

1. Dirty Read (P1). A transaction T1 modifies some row and another transaction T2 reads that row before T1 commits. The implication of this phenomenon is that if transaction T1 issues a ROLLBACK statement (abort), it will be as if transaction T2 read values that have never existed.

2. Non-Repeatable Read (P2). A transaction T1 reads some row. Another transaction T2 modifies that row and performs a commit. If T1 attempts to re-read that row, it can observe the changes done by transaction T2.

3. Phantom (P3). A transaction T1 reads a set of rows that satisfy some condition. Another transaction T2 executes a statement that causes new rows to be added or removed from the search condition. If T1 repeats the read, it will obtain a different set of rows.

The following table [SQL92, SQL99] specifies the phenomena that are possible and not possible for a given isolation level:

Isolation Level	**P1**	**P2**	**P3**
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

In addition, the standard requires that:

1. The Serializable level should allow only serializable executions.
2. There are no visible uncommitted actions except for the RU level.
3. There are no update operations allowed at the RU level.

[BBGMOO95] provided a critique on the definition of ANSI SQL Isolation Levels, showing that the definitions can be interpreted ambiguously. They redefined phenomena P1-P3 by using operational patterns, and introduced an additional phenomenon P0 (Dirty writes) that all isolation levels should proscribe. This approach was close to the original definitions that appeared in [GLPT77].

The following are the patterns of histories that correspond to the phenomena P0-P3 specified in [BBGMOO95]:

P0 : w1(x) ... w2(x) ...(c1 or a1)
P1 : w1(x) ... r2(x) ...(c1 or a1)
P2 : r1(x) ... w2(x) ...(c1 or a1
P3 : r1(P) ... w2(y in P) ...(c1 or a1)

In these patterns, c1 and a1 indicate the COMMIT and ABORT (ROLLBACK) operations respectively. The "(c1 or a1)" clause at the end of each phenomenon basically means that the transaction T_1 performing the first operation is still active when the second operation of T_2 is executed. Even this set of phenomena is not sufficient, however, since we also need to restrict other possible sequences of operations, such as:

P3': w1(y in P) … r2(P) … (c1 or a1)

Basically, [BBGMOO95] pointed out that these phenomena were trying to define a transactional locking protocol! A simplified and corrected version of the locking definitions in [BBGMOO95] is given in Table 2.1.

TABLE 2.1

Locking Isolation Level	**Read Locks on Rows and Predicates (the same unless noted)**	**Write Locks on Rows and Predicates (always the same)**
READ UNCOMMITTED	None required (Read Only)	None required (Read Only)
READ COMMITTED	Short term Read locks (both)	Long term Write locks
REPEATABLE READ	Long term data-item Read locks Short term Read Predicate locks	Long term Write locks
SERIALIZABLE	Long term Read locks (both)	Long term Write locks

In Table 2.1, a *short term* (*read* or *write*) *lock* is one that is taken prior to a (Read or Write) operation, and then released immediately after the operation is complete. A *long term lock*, on the other hand, is held until transaction commit time, the typical lock release time in 2-Phase Locking to guarantee Serializable behavior. While locks on rows are taken on individual rows, Predicate locks are taken on the "set" of rows specified in a SQL WHERE clause.

In [BBGMOO95, page 3], it is discussed how write predicate locks can conflict with read predicate locks. We have said that our model has no write predicate locks, but this earlier definition can be interpreted properly if we assume that a "write predicate lock" is taken on an individual row A whenever the row A is updated, so that the row operation W(A changes P) will conflict with PR(P).

An alternative definition of isolation levels was given in [ALO00]. Those definitions are more portable because they can apply to database systems that implement concurrency control by other methods than locking, such as multi-version and optimistic systems. This approach mainly defines isolation levels based on the types of cycles that would be allowed in the serialization graph of a history. It extends the classical data model used for conflict serializability [BHG87] with operations to model predicate evaluations and provides a unified way to treat single version and multi-version systems. Traditionally correctness for single version systems would be determined by using conflict-serializability, and view-serializability would be used for mutli-version systems.

There are several types of dependencies that are defined in [ALO00] for transactions participating in a history and they are established based on the existence of conflicting operations in the committed projection of the history. The following summarizes those definitions:

A *write-dependency* is similar to the dependency caused in conflict serializability when two operations perform a write on the same data item. We will also call this a *w-w dependency*.

An *item-read-dependency* is similar to the dependency caused by a write-read conflict. We will call it a *w-r dependency.*

A *predicate-read-dependency* captures the notion of the conflict that exists between a write operation and a predicate read operation when the write operation precedes the predicate read operation and *changes its matches*. A write operation changes the matches of a predicate read operation when it changes the set of rows that satisfy the predicate. A conflict exists in this case because if the execution order of the two operations were to be reversed the cumulative effect would be different. We will call this a *w-pr dependency.*

An *item-anti-dependency* is similar to the dependency caused by a read-write conflict in conflict serializability. We will call this a *r-w dependency*.

A *predicate-anti-dependency* captures the notion of the conflict that exists between a predicate read operation and a write operation when the write operation changes the matches of the predicate. A conflict exists because if the order of the operations were to be reverted, the cumulative effect would be different. We will call this a *pr-w dependency*.

The concept of the Serialization Graph we have already introduced is extended so that edges will be categorized according to the dependency type to which they correspond.

[ALO00] redefines the ANSI isolation levels based on the following phenomena:

G0 (Write cycles): The serialization graph contains a cycle consisting only of edges due to w-w dependencies.

G1: Phenomenon G1 consists of any of the following sub-phenomena (all must be proscribed if G1 is required to be Not Possible):

G1a (Aborted Reads): A committed transaction performs an item-read or a predicate-read that conflicts with a write performed by an uncommitted transaction which eventually aborts. For a single version database system this is equivalent to allowing the execution of a history that contains the following patterns of operations:

$W_1(A)$ ... $R_2(A)$ ... (A_1 and C_2)
W_1(A changes P) ... $R_2(P)$ ... (A_1 and C_2)

G1b (Intermediate Reads): A committed transaction performs an item-read or a predicate-read, and as a result observes a non final value written by another transaction. For a single version database system this is equivalent to allowing the execution of a history that contains the following patterns of operations.

$W_1(A)$ ... $R_2(A)$ .... $W_1(A)$ ... C_2
W_1(A changes P) ... $R_2(P)$ ... W_1(A changes P) ... C_2

G1c : The serialization graph contains a cycle consisting only of edges due to w-w, w-r, w-pr dependencies.

G2: The serialization graph contains a cycle with one or more edges due to r-w or pr-w dependencies.

G2-item: The serialization graph contains a cycle with one or more edges due to r-w dependencies.

The following table defines the isolation levels based on these phenomena

Isolation Level	**GO**	**G1**	**G2-item**	**G2**
READ UNCOMMITTED	NA	NA	NA	NA
READ COMMITTED	Not Possible	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible	Not Possible

2.4 Correctness and Precision of the Isolation Levels

We define the locking implementation of an isolation level to be correct if it produces only output histories that are permitted by our interpretation of ANSI SQL appearing in Table 2 in the previous section. In order to evaluate correctness we need to check in our testing method if the database system produces incorrect histories under a given isolation level. In [BBGMOO] the following definition appears:

Definition. Isolation level L1 is weaker than isolation level L2 (or L2 is stronger than L1), denoted L1 << L2, if all non-serializable histories that obey the criteria of L2 also satisfy L1, and there is at least one non-serializable history that can occur at level L1 but not at level L2. [In our case, the criteria of interest will turn out to be the locking behaviors of Table 1.] Two isolation levels L1 and L2 are equivalent, denoted L1 == L2, when the sets of non-serializable histories satisfying L1 and L2 are identical. L1 is no stronger than L2, denoted L1 $\underline{<<}$ L2 if either L1 << L2 or L1 == L2. Two isolation levels are incomparable, denoted L1 >><< L2, when each isolation level allows a non-serializable history that is disallowed by the other.

In comparing the strength of isolation levels in the above definition, we differentiate them only in terms of the non-serializable histories that can occur in one but not the other. But two implementations of the same isolation level can also differ in terms of the serializable histories they permit. We say Locking SERIALIZABLE == Conflict Serializable, even though it is well known that a locking scheduler does not admit all possible Conflict Serializable histories. To capture this idea, we define the *precision* of an implementation to be the fraction of valid histories of an isolation level that are permitted by the implementation. The definitions of isolation levels in Table 2.1 are locking-based, so it might seem that locking implementations of these isolation levels cannot be more restrictive than what is defined, but precision would be adversely affected if, for example, an implementation used page locking instead of row locking.

In addition, the SERIALIZABLE isolation level is well understood to include many histories that are not permitted in a Locking SERIALIZABLE implementation, since the Serialization Graph definition of Serializable permits pairs of conflicting operations in a history that would be forbidden by locking, as long as conflict circuits do not arise in the Serialization Graph. The definitions in [ALO00] generalize the Serialization Graph definition to other isolation levels, demonstrating that Locking interpretations of isolation levels are not as precise as we would like. I will be concentrating on locking schedulers in my thesis, so such considerations will not be central to my work. However, the definition of Precision is of independent interest.

3. Tool and Notation

3.1 Notation

3.1.1 Introduction to Notation

Our project has developed a software module called HISTEX (HISTory EXerciser), which executes input histories written in a generic transactional notation on commercial DBMS platforms. The HISTEX notation can be used by a researcher in much the same way as the classical transactional notation found in [BHG87] to write down sequences of operations that model concurrent histories. However, there are a number of differences:

(1) Our HISTEX notation can represent a number of operational situations that occur in commercial products but that up to now could not be represented by researchers attempting to specify arbitrary histories in a rigorous way. The new operations HISTEX provides include simple ones such as Inserts, Deletes, and indivisible Read-Write Update operations ($RW_i(A)$), and more complex ones such as predicate evaluation, PR(P . . .), which represent an Open Cursor operation and/or a sequence of Fetch operations from the Cursor.

(2) While our HISTEX notation extends the number of operations from classical notation, it is nevertheless *generic*, meaning that it leaves details of operations as undefined as possible. By avoiding SQL-level specification, it allows researchers to concentrate on expressing a history, rather than becoming fixated on unimportant details. The HISTEX program module interprets generic parameters such as i, A, and X in $R_i(A, X)$, assigning an operation with subscript i to a particular transaction thread, performing a Read (Select) operation of a particular row it consistently assigns the name A, and reading the (default) column value[s] of A into a value-variable it associates with the name X, one of multiple memory variables maintained by HISTEX.

(3) The HISTEX notation makes no assumption about the Concurrency Control (CC) protocol under which the history will be executed; this distinguishes HISTEX notation from that of [ALO00], for example, where a multi-version CC protocol is assumed and version numbers must be assigned to data items read, fixing the execution down to the detail level.

(4) The HISTEX notation has an *output history* format to represent the results of an input history execution on a specific database platform (a specific DBMS and Isolation Level). In the HISTEX output history form, values will be provided for rows/columns read and written, and types of failures in history execution will be noted.

3.1.2 HISTEX Default Table

By default, HISTEX interprets input histories in terms of a canonical relational table described as follows:

```
T(reckey,recval,c2,c3,c4,c5,c6,c50,c100,k2,k3,k4,k5,k6,k50,k100)
```

The table T contains a parameterized number of rows (by default, the number is 200, but this can be altered to any multiple of 100). Columns k2 through k100 are indexed integer columns, where column kN has values 0, 1, 2, . . ., N-1 for successive rows, starting with the first row and extending through the last. Columns c2 through c100 have identical values with the corresponding k2 through k100 columns, but are not indexed (this will make a difference in predicate evaluation execution). The column named reckey is a primary key for the table, used to identify each individual row A, B, . . . used in an operation, and the column named recval is the default "value" of the row, which will be incremented by 1 when unspecified updates are performed. The values for reckey will be successively assigned to rows in T with values 100, 200, . . ., and recval values will be successively assigned with values 10000, 20000, ...

reckey	recval	c2	c3	c4	c5	c6	c50	c100	k2	k3	k4	k5	k6	k50	k100
100	10000	0	0	0	0	0	0	0	0	0	0	0	0	0	0
200	20000	1	1	1	1	1	1	1	1	1	1	1	1	1	1
300	30000	0	2	2	2	2	2	2	0	2	2	2	2	2	2
400	40000	1	0	3	3	3	3	3	1	0	3	3	3	3	3
500	50000	0	1	0	4	4	4	4	0	1	0	4	4	4	4
600	60000	1	2	1	0	5	5	5	1	2	1	0	5	5	5
. . .	. . .	..	..	..	..	..	...	...	..	..	..	..	..	...	...

The input histories are considered to be a sequence of the following operations[9]:

3.1.3 HISTEX Operation Definitions

In this section we list the operations that are currently supported by the HISTEX notation. The declarative operations are listed first. These operations will not cause any database access. They are used to initialize variables that can be referenced in subsequent operations.

Predicate Declaration

This operation will associate the predicate variable P with a predicate expression suitable for a SQL Where Clause.

Syntax:

PRED(_<predicate_name>, ["]<predicate_expression>["])_

<predicate_name> := <variable_name>

[9] NOTE: We will call the notation we describe here the "textbook" notation in order to distinguish it from the "implemented" notation. The two notations are isomorphic. For example an operation that appears as O_1(A,A0) in the textbook notation will be written as the tuple (1,O,A,A0) in an input history.

<variable_name> := ***A sequence of letters and digits starting with a letter***

<*predicate_expression> :=* ***An expression that can appear in a SQL WHERE clause***

Example: PRED (P, "k2=1 and k3<2")

Typically predicate variables will have names beginning with P or Q, such as P1, P2, Q, etc.

<u>Row Declaration</u>

This operation associates a row variable with a specific row. The row id is expected to be a value of the reckey column of the underlying table.

Syntax:

MAP(*<row_name>, <row_id>****)***

<row_name> := <variable_name>

<row_id> := ***<u>An integer value that can appear in the column reckey of the underlying table</u>***

Example:

MAP(A, 100)

<u>Isolation Level Set Operation</u>

This operation is used to set the transaction isolation level for a particular transaction. It must be the first operation of a transaction. Currently the isolation levels that can be specified by this command depend on the underlying DBMS executing the history.

ORACLE provides two isolation levels, READ COMMITTED (RC) and SERIALIZABLE (SR).

DB2 provides four isolation levels, UNCOMMITTED READ (UR), CURSOR STABILITY (CS), READ STABILITY (RS) and REPEATABLE READ (RR). DB2 does not follow the ANSI convention for naming the isolation levels, and this can be somewhat confusing. The following table shows the correspondence between the DB2 isolation levels and the ANSI isolation levels:

DB2 Isolation Level	**ANSI Isolation Level**
UNCOMMITTED READ	READ UNCOMMITTED
CURSOR STABILITY	READ COMMITTED
READ STABILITY	REPEATABLE READ
REPEATABLE READ	SERIALIZABLE

INFORMIX provides the four ANSI isolation levels READ UNCOMMITTED (RU), READ COMMITTED (RC), REPEATABLE READ (RR) and SERIALIZABLE (SR). As of INFORMIX Universal Server 9.1, the RR level has been implemented to be identical with the SR level.

Syntax:

IL$_{<tid>}$*(<isolation_level>)*

Where:

<tid> := ***non_negative_integer***

<isolation_level> := *<ora_isolation_level>* | *<db2_isolation_level>* | *<inf_isolation_level>*

<ora_isolation_level> := ***RC*** | ***SR***

<db2_isolation_level> := ***UR*** | ***CS*** | ***RS*** | ***RR***

<inf_isolation_level> := ***RU*** | ***RC*** | ***RR*** | ***SR***

Example: IL_1 (SR)

When interpreting this operation on an ORACLE database, HISTEX will execute the SQL statement:

set transaction isolation level serializable

A special mechanism is used for setting the isolation level on a DB2 database because the versions we have experimented with (5.0, 6.1) did not provide such a SQL command. More details on this are included in the section describing the HISTEX implementation.

<u>Write Operation</u>

This operation models a blind update of a row.

Syntax:

$\boldsymbol{W}_{<tid>}$***(<data_item_spec> [, <expression>])***

Where:

<tid> := ***non_negative_integer***

*<data_item_spec> := <row_name> [**; column_name**]*

<row_name> := ***variable_name***

<expression> := {<simple_expression | <complex_expression }[10]

*<simple_expression> := {***integer** | *<value_name>}*

<complex_expression> := <atomic_expression> <operator> <atomic_expression>

<operator> := { **+** | **-** | * | */}*

<value_name> := ***variable_name***

Example:

W_1(A, 1001)

When interpreting this operation, HISTEX will identify the row variable A with a row (identified internally by its reckey). If A has already been identified with a row at some prior point in the history (even in an operation of a different transaction), then A will continue to be identified with the same row. To execute the operation W1(A, 1001), HISTEX is going to execute an SQL statement of the form:

update T set recval = 1001 where reckey = 100

Further, any <column_name> of the underlying table can be written in this operation. See the <data_item_spec> definition above. Such an operation could be used in order to produce a conflict with a predicate P. For example, if a predicate P were defined as k2=1, the operation W_2(B;k2, 0) would guarantee that the row B would not be in P.

Read Operation

This operation performs a read of a data item (row).

Syntax:

[10] The current implementation of HISTEX supports only simple expressions.

$\boldsymbol{R}_{<tid>}$*(<data_item_spec> [, <value_name>])*

For syntax element definitions of this operation, see the Write operation above. In particular, note that <data_item_spec> can specify any given column of a row-variable.

Example:

R_1(A, X)

When interpreting this operation, HISTEX will associate the row-variable A with a specific row in the underlying table in the same way as the Write operation described above.

It will then open an SQL cursor for the prepared statement:

select <column_name> into :value from T where reckey=100

The value retrieved by this operation will be assigned to the variable X.

Read-Write Operation

Syntax:

$\boldsymbol{RW}_{<tid>}$*(<data_item_spec> [, <update_expression>])*

<*update_expression*> *:=* ***An expression that can appear at the right side of a SET clause in a SQL UPDATE statement.***

Example: RW_1(A;k2, k2+k3)

HISTEX will associate the variable *A* with a row of the underlying table (if necessary). Then it will execute an SQL statement of the form:

update T set k2 = k2 + k3 where reckey = 100

Insert Operation

This operation will insert a new data item row in the underlying table.

Syntax:

I$_{<tid>}$*(<data_item_i> [, <data_value_i>])*

*<data_item_i> := <data_item_name> [**; column_name** [**; column_name** …]]*

*<data_value_i> := **integer** [**; integer** …]*

Examples:

I_1(A)

HISTEX, by executing this operation, will associate the row-variable *A* to a new reckey value. Then it will insert a new row in the underlying table, which will contain the new reckey value and some default values for the rest of the columns[11].

I_2(B;recval;k2;k3, 3000;0;2)

In this example, HISTEX will associate variable B to a new reckey value. It will then execute the SQL statement[12]:

insert into T (reckey,recval,k2,k3) values (350,3000,0, 2)

Delete Operation

This operation will delete a data item (row) from the underlying table.

Syntax:

D$_{<tid>}$*(<data_item_name>)*

Examples:

D_1(A)

This operation will delete the row that is associated with the data item variable A.

Predicate Read Operation

The purpose for introducing this operation is to provide a way of producing predicate read/item write conflicts.

[11] In the current implementation when there are no columns specified in the Insert operation, the inserted row will contain values for the reckey and recval columns.

[12] The current implementation could be extended so that default values would be included for the columns that are not specified in the operation.

Syntax:

PR$_{<tid>}$***(<predicate_spec>, <data_value_name>)***

<*predicate_spec> := <predicate_name>* ***;*** *<column_spec>* ***;*** *<iterator>[****;***
<data_item_name>]

*<column_spec> := {****column_name*** **|** *<aggregate_operation>****(column_name)*** *}*

<aggregate_operation> := *{****count*** **|** ***sum****}*

<iterator> := { ***non_negative_integer*** **|** ***all*** *}*

Examples:

PR$_1$(P;recval;1;A, X)

This operation will attempt to read one (1) row that matches the predicate P. In the current implementation, this operation will cause the opening of a cursor for selecting the recval column of the rows that match the predicate P. When the operation includes just a column name, as in this example, the reckey column is retrieved as well. The value of the reckey column will be registered in the variable A[13]. The value of the recval will be registered in the variable X.

In a case where the operation specified more than one rows to be retrieved, the reckey and recval of the last row that was retrieved would be assigned to the variables A and X, respectively.

PR$_1$(P;reckey;all)

This operation is going to retrieve all rows that satisfy the predicate P by reading the reckey value. Note that this is the most economical operation for accessing a predicate. If an index has been created for the reckey column, the operation can execute without having to retrieve the rows that satisfy the predicate; it only has to access the index.

PR$_1$(P;count(*);1)

This operation will cause the execution of a SQL statement of the form: *select count(*) from T where P*. The database will return the number of rows that satisfy the predicate.

<u>Commit Operation</u>

Syntax:

C$_{<tid>}$

[13] Note that the PR operation can change the value of a data item variable.

Example:

C_1

For the operation C_1, HISTEX will issue a COMMIT statement in the thread associated with transaction 1.

<u>Abort Operation</u>

Syntax:

$A_{<tid>}$

Example:

A_1

For the operation A_1, HISTEX is going to issue a ROLLBACK statement in the thread associated with transaction 1.

<u>Executing macro expanded SQL statements</u>

EXECSQLI$_{<tid>}$(<immediate_sql_statement_with_predicate_macros>)

<immediate_sql_statement_with_predicate_macros> := ***A SQL statement that can be executed by the EXECUTE command and which does not return any value. The statement can include macro references of the form %<predicate_name>.***

Examples:

PRED(P, k2=0 and k1=1)
EXECSQLI$_1$(update T set recval = recval + 1 where %P)

In this history, HISTEX is going to substitute %P with "k2=0 and k1=1" and then execute the derived statement as an operation of transaction 1.

This operation can be used for executing what we call a set update operation – an operation that will update all the rows that satisfy a predicate.

EXECSQLS$_{<tid>}$(<select_sql_statement_with_predicate_macros>)

<select_sql_statement_with_predicate_macros> := ***A select SQL statement that selects a single column. The statement can include macro references of the form %<predicate_name>***[14]***.***

Examples:

PRED(P, k2=0 and k1=1)
EXECSQLS$_1$(select sum(recval) from T group by k1, k2 having %P)

For this example HISTEX is going to substitute %P with the corresponding predicate, and then execute the select statement by opening a cursor and fetching all the selected rows. Currently the operation does not return any values. It can be used to test whether SQL select statements that have not been mapped to other HISTEX operations acquire the necessary locks.

[14] Note: The reference of a predicate variable in a macro definition might change from %P to %%P because the character % could be used in SQL statements as a wild character

3.2 The Tool Architecture

3.2.1 Overview

HISTEX is a multi-process application. It consists of two major modules: **monitor** and **thread**. At runtime there is a process executing the monitor module, and there can be several processes, each one executing a thread module.

The monitor process is the one that creates the thread processes. It is responsible for scanning and loading the input history, maintaining structures containing the state of the variables that are used in the history, and producing the output.

The thread module is responsible for interacting with a database system and provides the embedded SQL implementation of the HISTEX operations.

There is a communication protocol between the monitor and the threads. The monitor sends messages terminated with a newline character and the thread responds with SUCCESS or FAILURE. The thread's response also contains any value requested by the monitor, orsome error message.

The current implementation of HISTEX is done for the UNIX operating system (specifically Sun Solaris 2.7, but portable to most other UNIX platforms) and the communication between the monitor and the thread processes is done by using pipes. The threads themselves are UNIX processes.

In what follows, we describe some of the more complex parts of the implementation.

3.2.2 Communication between the monitor and the thread processes

We have developed a rather generic mechanism for implementing a system of a monitor process communicating with a group of thread processes. The system is developed in such a way that it could be utilized by other applications that follow a similar pattern.

In order to create a new thread the monitor uses the following function:

```
Thread create_thread (void *call_thread(), void *parameters)
```

The first argument is a pointer to the function that will be called by the newly created thread. In the current application this function is the one that contains the Embedded SQL implementation of the HISTEX operations. The second argument is a pointer to a generic structure that can hold the parameters that the monitor would pass to the thread. Currently the only parameter passed is the default isolation level.

The function will return the thread id encapsulated in the type `Thread.` Currently this type is implemented as an integer (`int`).

The communication between the monitor and the thread processes is implemented with UNIX pipes. When a new thread process is created, a pair of pipes is created between the monitor and the thread process, one for each direction of data flow.

A message can be sent to a thread by using the following function:

```
Boolean send_to_thread (Thread threadId, char *buff)
```

`ThreadId` is the thread to receive the message. The parameter `buff` points to the message which must be a newline terminated string.

The function:

```
Boolean wait_for_thread(Thread threadId, int timeout, Boolean *toflag)
```

is used by the monitor process in order to wait for a particular thread to respond. It will return FALSE if a system error occurred while waiting, and TRUE otherwise. The parameter `threadId` specifies the thread. The parameter `timeout` specifies a maximum number of seconds that the function will wait for the thread to respond. The parameter `toflag` will be set to TRUE if a timeout occurred while waiting, and to FALSE otherwise.

The following function

```
Boolean wait_for_any_thread(Thread *threadId, int timeout, Boolean *toflag)
```

is used to wait for any thread to respond. This is used when HISTEX is executed in the *asynchronous* mode. Under this mode it is possible that after HISTEX has submitted a history operation to an execution thread, it could proceed in processing the next operation even though the earlier one has not completed yet. After HISTEX has submitted all the possible operations, it calls this function to wait for a thread to send a response. The argument `threadId` will identify the thread.

For receiving a response from a thread the following function is used:

```
Boolean receive_from_thread (Thread threadId, char *buff)
```

`Finally the following function`

```
void finalize_threads ()
```

is used to terminate all thread processes.

3.2.3 Setting the isolation level dynamically

The ANSI SQL-99 [SQL99] standard provides the SET TRANSACTION ISOLATION level statement for setting a desired isolation level when a transaction starts. This statement could be utilized by HISTEX for implementing a corresponding operation so that input histories could consist of transactions running at different isolation levels.

One of the products we wanted to experiment with, namely IBM DB2, did not provide such a command. The ISOLATION LEVEL must be set when the module containing embedded SQL is compiled or bound[16]. Originally, in order to run histories at different isolation levels we had to create a separate executable for each supported isolation level. However this would not be enough for examining histories containing transactions running at mixed isolation levels. In order to solve this problem we reorganized the thread module so that it would consist of two levels.

The higher level would contain the function `thread()` that would be called by the monitor module. When this function receives an operation, it examines whether it is an operation that sets the isolation level. If it is, then the function just stores the isolation level into a variable. When other types of operations are received, this variable is examined and the message is sent to the corresponding secondary thread function (e.g. to the CS-version of thread code if the current isolation level is Cursor Stability). The implementations of the secondary functions are identical except for their name. The name contains an indication of the isolation level that was used in the compilation. All these functions are linked together to form a single executable.

3.2.4 Implementing the Predicate Read Operation

We have implemented the predicate read (PR) HISTEX operation by using SQL cursors. Another approach could have been to just use a SELECT statement. Choosing a cursor implementation allows a wider range of testing scenarios such as the partial evaluation of a predicate. This is important in order to observe how a database system that implements Key-Value Locking behaves (e.g. we expect the locking to incrementally advance across a predicate set).

In such cases it is possible that at some point in time a transaction T_1 has already opened a cursor and fetched some of the rows. At that moment the database system receives an update operation from transaction T_2, which changes the matches of the predicate used by T_1. In a system that implements KVL locking [MOHAN96] it is not certain that this operation will be blocked. It is possible that T_1 has already locked a range of values that contain some of the column values of the row updated by transaction T_2. In this case the update of T_2 will be blocked. Another possibility is that transaction T_1 has not yet acquired any lock that could conflict with the update done by T_2. In that case T_1 could perform the update, and the scheduler will consider T_1 serialized before T_2. Later on when the

[16] This restriction applies to DB2 Version 5. DB2 version 6 introduced a new command "CHANGE ISOLATION LEVEL" for selecting a specific isolation level.

scanning of the predicate operation reaches the range that could conflict with the update, it is possible that the operation will be blocked (in case T_1 has not committed yet).

The PR operation of HISTEX provides the ability to form such scenarios. Different instances of this command can be used by the same transaction in order to access a predicate in several steps. Each instance needs to reference the same predicate variable and can specify the number of rows that will be accessed each time.

In order to implement this feature we are associating a predicate variable with some SQL cursor. The number of cursors that can be opened simultaneously is fixed. By convention, we use integer numbers to identify each cursor.

The first time a predicate variable is used by a PR operation, a cursor has not been opened yet. The message sent by the monitor process contains the value 0 as the required cursor id. When the thread processes the request, it will identify the next available cursor ID and will attempt to open a cursor.

To open cursors we use a switch statement where the case labels are the supported cursor ids. The following figure shows the C code for handling the opening of cursors:

```
switch(cursor_id) {

           case 1:
                 EXEC SQL PREPARE S1 FROM :sql_stmt;
                 EXEC SQL DECLARE C1 CURSOR FOR S1;
                 EXEC SQL OPEN C1;
                 break;
           case 2:
                 EXEC SQL PREPARE S2 FROM :sql_stmt;
                 EXEC SQL DECLARE C2 CURSOR FOR S2;
                 EXEC SQL OPEN C2;
                 break;

           ...
}
```

The code associates a cursor id with variables Sx and Cx for the corresponding prepared statement and cursor[17].

The cursor id of a recently opened cursor will be included in the thread's response. The monitor process will map the predicate variable used in the PR operation to this cursor id.

When the monitor encounters a subsequent PR operation referencing the same Predicate variable, it will extract the cursor id that was already mapped to this variable, and it will included it in the message sent to the thread. The following is the logic executed by the thread when a specific cursor id is provided:

[17] The reason we use this technique is that it appears that we could not store cursor identifiers into an array and thus be able to dynamically access them.

```
switch (cursor_id) {

       /* arg3 indicates the rows to scan                        */
       /* aggrfl is set when an aggregate operation is processed */

       case 1:

         if (!strcmp(arg3, "all")) {
           EXEC SQL WHENEVER NOT FOUND GOTO label_1_1;
           I = 0; for(;;) {
             if (aggrfl) {
               EXEC SQL FETCH C1 INTO :value;
             } else {
               EXEC SQL FETCH C1 INTO :key, :value;
             }
             i++;
           }
           label_1_1 :
           EXEC SQL CLOSE C1;
           free_cursorid(cursor_id);
           cursor_id = -1;

         } else {
           EXEC SQL WHENEVER NOT FOUND GOTO label_1_2;
           j = atoi(arg3);
           i = 0; while (i<j) {
             if (aggrfl) {
               EXEC SQL FETCH C1 INTO :value;
             } else {
               EXEC SQL FETCH C1 INTO :key, :value;
             }
             i++;
           }
           label_1_2:
           ;
         }
         break;

       case 2:     /* same as in case 1 (substitute C2 for C1 */

         if (!strcmp(arg3, "all")) {
           EXEC SQL WHENEVER NOT FOUND GOTO label_2_1;
           i = 0; for(;;) {
             if (aggrfl) {
               EXEC SQL FETCH C2 INTO :value;
             } else {
               EXEC SQL FETCH C2 INTO :key, :value;
             }
             i++;             /* Count the actual fetches */
           }
           label_2_1:
           EXEC SQL CLOSE C2;
           free_cursorid(cursor_id);
           cursor_id = -1;
```

```
          } else {
            EXEC SQL WHENEVER NOT FOUND GOTO label_2_2;
            j = atoi(arg3);
            i = 0; while (i<j) {
              if (aggrfl) {
                EXEC SQL FETCH C2 INTO :value;
              } else {
                EXEC SQL FETCH C2 INTO :key, :value;
              }
              i++;               /* Count the actual fetches */
            }
            label_2_2:
            ;
          }
          break;

...
```

In the preceding code, the different cases of the switch statement are identical except for the use of different names for some variables. If the PR operation requests that **all** rows are read, then the logic in the first conditional block will be executed and the whole rowset will be scanned. Having completed this, the cursor will be closed and the corresponding cursor id will be freed, becoming available for use by a different PR statement. Otherwise, only the specified number of rows will be retrieved.

Note that in the case that a specified number of rows is requested, the cursor will not be recycled even after all rows have been fetched. This behavior has been chosen so that it can be easily determined when a cursor has been closed.

3.2.5 Synchronous vs asynchronous execution mode

By default HISTEX executes histories in a *synchronous* (*serial*) *mode* (i.e. an operation is processed after any preceding operation has been executed). This mode is the one used for the experiments we present in this dissertation. HISTEX also provides an *asynchronous* (*concurrent*) *mode* of execution, where operations can be submitted simultaneously to different threads. This is necessary when it is desirable to create concurrent workloads in a database. We have used this feature for implementing the performance measuring experiments in [FLOOS00]. The current level of concurrency is that of *one outstanding operation per transaction* (i.e. the monitor submits any operation it encounters to the corresponding thread until it reaches an operation of a transaction with an unfinished operation). The asynchronous mode is enabled by using the HISTEX option (-c).

4. Experimentation and Results

4.1 Developing a testing methodology

We developed HISTEX with the intent of testing whether database vendors correctly implement isolation levels. A general approach that occurred to us was to create a large number of random input histories and execute them under different database systems and the isolation levels provided. The output histories could then be analyzed to determine whether those histories should have been produced by a given isolation level.

An approach to generate stochastic tests to determine when different database systems provided different answers to identical SQL statements has already been successful [SLUTZ98].

We considered using the characterization that appeared in [ALO00] to determine when output histories were legal under various isolation levels. This work provides implementation-independent definitions of the isolation levels so that ORACLE multi-version concurrency can be treated similarly to the way locking concurrency is treated for other database systems. The paper uses a multi-version notation for defining histories and it defines the isolation levels based on the type of cycles that could be allowed in the output history and a few additional constraints.

However, there were several reasons that suggested we could not readily rely on the approach used in [ALO00]. The main reason was that histories appearing in the paper also needed to specify the versions of the items that a system had chosen for every operation. Clearly the operations in our input histories could not specify particular versions, and a multi-version system, like ORACLE, did not report what versions were used. It could only report the values of the data items and the analyzer of the output history would have to determine the version.

The [ALO00] approach also relied on the existence of a version order of the data items included in the history, information that was not available in an output history from ORACLE, and there was no general way of deriving it.

In addition, to determine the predicate dependencies in [ALO00], we would require knowledge of the whole database state at the moment the predicate operation occured. In order to avoid performing a total read of the relations involved in the predicate, which is an action that would alter the meaning of the examined history, we would need to implement a complex mechanism for mirroring the database so that the versions of the rows in a table could be identified by querying this mirror.

Finally, in order to detect a predicate conflict, when examining the output history we should be able to reproduce the state of every row updated before and after the update takes place. Even though the update might specify only the value of a column being modified, in order to determine if the row update affects some predicate, the values of the rest of the columns must be available.

While we were still considering ways of addressing the issues of dealing with multi-version concurrency, we decided to concentrate on testing isolation levels acting under locking concurrency. All commercial databases other than ORACLE use locking, including DB2, Informix, and Microsoft's SQL Server.

For reasons that will become apparent in the sequel, we have decided to define a methodology that utilizes assumptions of concurrency control mechanisms about the underlying database system. In this way we can considerably simplify the testing cases.

Instead of creating histories that would try to produce the phenomena described above (i.e. either the output patterns in the [BBMGOO] approach or the cycles in [ALO00]), we can show that it would be sufficient to check whether the locking protocol is implemented according to Table 4.1.

Table 4.1 provides definitions of isolation levels based on the pairs of concurrent conflicting operations that should be avoided. These are actually the effects that a locking scheduler should have, as it was specified in Table 2.1.

Table 4.1. Isolation Levels defined in terms of prohibited concurrent pairs of conflicting operations

Locking Isolation Level	**Concurrent pairs of conflicting operations that should be avoided**
READ UNCOMMITTED	There are no conflicting operations. Transactions are READ ONLY
READ COMMITTED	$W_1(A)$ $W_2(A)$[18] $W_1(A)$ $R_2(A)$ W_1(A changes P) $PR_2(P)$
REPEATABLE READ	All the above and: $R_1(A)$ $W_2(A)$
SERIALIZABLE	All the above and: $PR_1(A)$ W_2(A changes P)

We will rely on this table to define a plan for testing the correctness of isolation levels by database systems.

[18] A W operation in this table stands for any operation that performs a Write (i.e. W, RW, I, D)

Theorem 1 Adequacy of testing pairs of operations.

If a database system prevents the concurrent operations in Table 4.1, then it implements the isolation levels correctly. By "correctly", we mean that at a given isolation level there will be no phenomenon occurring that is proscribed by that level.

Proof:

We will prove this considering the definitions given in [ALO00] (See section 2.3).

Case for READ UNCOMMITTED:

This level should disallow a cycle consisting of w-w edges in the serialization graph. A database system is meant to execute Read Only transactions. Since no write operation will be allowed execution, it will not be possible for a cycle to be formed in the Serialization Graph.

Case for READ COMMITTED:

If a database system obeys the restrictions in TABLE 4.1, it will also disallow phenomenon G1. This is because the sub-phenomena G1a, G1b and G1c will all be disallowed.

G1a will be disallowed because otherwise the DBMS would allow the concurrent execution of a pair $W_1(A)$ $R_2(A)$. This however is not allowed according to Table 4.1.

G1b will be disallowed for a similar reason.

In order to prove that G1c will be disallowed as well, we will show that in the serialization graph of any history there cannot be a cycle consisting of w-w, w-r and w-pr edges. Let's assume that there is such a history and the cycle consists of the following transactions:

T1 -> T2 -> ... -> Tn

where each edge is due to a w-w, w-r or w-pr dependency.

The edges in the cycle imply that a transaction T_k must commit after transaction T_{k-1} has committed. If T_k committed before T_{k-1}, then a proscribed pair of operations (the ones responsible for the edge in the graph) of concurrent transactions would have been allowed to execute. This implies that T_n must commit after T_1. In the cycle however, there is an edge T_n->T_1 and this implies that transaction T_1 must commit after transaction T_n. In this way we have reached a contradiction, and so a cycle could not have occurred if the database system proscribed the concurrent execution of the operations in Table 4.1.

In the case of a database system, this cycle would have been prevented because a deadlock would have occurred.

Case for REPEATABLE READ

This level should disallow phenomena G1 and G2-item. Phenomenon G1 is disallowed because the level would not allow the concurrent execution of the operations that are disallowed by the READ COMMITTED level (see proof above).

In addition, phenomenon G2-item is disallowed because if there were a cycle in a serialization graph containing an item anti-dependency, a deadlock would occur. The proof is very similar to the READ COMMITTED case.

Case for SERIALIZABLE

This level should disallow phenomena G1 and G2. G1 is disallowed for the same reason that it is disallowed for the previous levels. G2 will be disallowed because if there was a cycle in the serialization graph, a deadlock would have already occurred by executing the transactions.

4.2 Testing plan of isolation levels for a locking scheduler

In this section we describe a testing plan. The first subsection describes an idea for testing the READ UNCOMMITTED level only. The implementation and results for this case are included in section 4.4. The remaining subsections are derived from the cases included in Table 4.1. Section 4.3 describes a system for implementing these test cases. The results we have obtained are described in section 4.4.

4.2.1 Testing for correctness at the READ UNCOMMITTED level

This level should be used by READ ONLY transactions [SQL92, SQL99], and there are no locks required. It appears that we do not need to perform any locking test. However, we need to test that transactions running at this level are prevented from performing updates.

Our experience on some commercial database products indicates that transactions that have started at the READ UNCOMMITTED level can perform updates. Such a behavior is considered to be a deviation from the standard. It is possible that the database system escalates a transaction running at READ UNCOMMITTED to a higher isolation level at the moment the transaction executes a write. If this is the case, we need to ensure that the escalated transaction doesn't perform an update based on uncommitted data. The following HISTEX history can be used for performing this test:

$R_1(A)R_1(B)C_1W_2(A)\ R_3(A, A0)\ W_3(B, A0)\ C_3\ A_2\ R_4(A)\ R_4(B)\ C_4$

In this history transaction T_3, executing at the READ UNCOMMITTED level, reads the value of a row that has been updated by an uncommitted transaction T_2. T_3 then attempts to perform a write based on that value and commits. If transaction T_2 were to abort, T_3 would have performed its update based on a value that never existed. When transaction T_4 later reads the values of *A* and *B* for testing purposes, if *B* contains the value that was written by T_2, it means that T_3 performed an update based on uncommitted data, an act that is proscribed by the ANSI SQL standard.

4.2.2 Testing for correctness at the READ COMMITTED level

At this level we expect that long write locks are acquired for data items, and short read locks are acquired for data items and predicates. Recall that we assume that all operations that perform updates take write locks, and all other operations take read locks.

We can test whether the appropriate locks are acquired by testing whether we can observe their effects. The long write locks on data items acquired by a transaction T_1 are expected to block any other transaction T_2 from acquiring a write or read lock on a data item.

The following are the possible combinations of operations that would cause a write-write lock conflict[19]:

$W_1(A)$	$W_2(A)$
$W_1(A)$	$RW_2(A)$
$W_1(A)$	$D_2(A)$
$RW_1(A)$	$W_2(A)$
$RW_1(A)$	$RW_2(A)$
$RW_1(A)$	$D_2(A)$
$I_1(A)$	$W_2(A)$
$I_1(A)$	$RW_2(A)$
$I_1(A)$	$D_2(A)$

If we detect that the second operation was not blocked, it is an indication that the locking protocol did not work properly.

The following are the possible pair of operations that would cause a write-read lock conflict:

$W_1(A)$	$R_2(A)$
$RW_1(A)$	$R_2(A)$
$I_1(A)$	$R_2(A)$

$W_1(A \text{ into } P)$	$PR_2(P)$

[19] We need to clarify that pairs that start with $D_1(A)$ operations are not considered for specifying item conflicts because the deleted item will not exist after the delete operation.

W_1*(A outof P)*	PR_2*(P)*
RW_1*(A into P)*	PR_2*(P)*
RW_1*(A outof P)*	PR_2*(P)*
D_1*(A in P)*	PR_2*(P)*
I_1*(A in P)*	PR_2*(P)*

The notation "*A {into/outof} P"* is not supported by the current implementation of HISTEX. It means that the update of the data item *A* should be done in such a way that the item changes the set of rows that satisfy the predicate P. In order to achieve the same effects the writer of a history needs to choose the data items in such a way that they would cause a predicate conflict with the corresponding predicate read operation. Another way would be to have an initialization transaction execute a *PR(P;reckey;1;A)* operation and associate the variable *A* with a row that matches the predicate. In order to choose a row that does not match a predicate *P*, we could form a predicate *Q* as the negation of *P* and issue a predicate operation *PR(Q;reckey;1;A).*

4.2.3 Testing for correctness at the REPEATABLE READ level

At this level, the locking protocol should show all the same conflicts as in the previous level, and in addition there should be long term read locks acquired for data items. To detect whether long term read locks are acquired properly, we could check whether a read operation on a data item conflicts with a write operation that will be issued later by another transaction. This is because a write operation will always try to acquire an incompatible lock. So we need to perform the following additional tests:

R_1*(A)*	W_2*(A)*
R_1*(A)*	RW_2*(A)*
R_1*(A)*	D_2*(A)*

4.2.4 Testing for correctness at the SERIALIZABLE level

In addition to the tests performed at the previous levels, for this level we need to check whether the locking protocol holds long term read predicate locks. Long term read predicate locks conflict with write operations that change the set of rows that match the predicate. This can happen by a transaction that inserts or deletes a row that matches the predicate read by a different transaction or that updates a row in such a way that the row would match the predicate before the update but would not do so after the update or vice versa. In order to test this level, we need to perform the following additional tests:

PR_1*(P)*	W_2*(A into P)*
PR_1*(P)*	W_2*(A outof P)*
PR_1*(P)*	RW_2*(A into P)*
PR_1*(P)*	RW_2*(A outof P)*
PR_1*(P)*	D_2*(A in P)*
PR_1*(P)*	I_2*(A in P)*

4.2.5 Testing additional SQL statements

While normal HISTEX operations execute typical SQL statements that can read predicates and update rows, it is possible that some of the more unusual SQL clauses will not be exercised in normal HISTEX: e.g., Select statements with GROUP BY and HAVING clauses, or set oriented updates. HISTEX therefore has been designed to provide these alternative SQL operational forms by supporting a command *execsqli,* which can be used to execute any SQL statement that does not return a value. There is an additional command *execsqls* that can be used to execute general SQL select statements.

We can now treat these operations in the same way we treated the Predicate Read operations in creating item-write/predicate-read pairs. So if *S* is an SQL statement that uses predicate *P*, we need to test the following additional pairs for all isolation levels at READ COMMITTED and above.

$W_1(A\ into\ P)$	$S_2(P)$
$W_1(A\ outof\ P)$	$S_2(P)$
$RW_1(A\ into\ P)$	$S_2(P)$
$RW_1(A\ outof\ P)$	$S_2(P)$
$D_1(A\ in\ P)$	$S_2(P)$
$I_1(A\ in\ P)$	$S_2(P)$

In addition, all pairs we can form by reversing the operations above need to be tested at the SERIALIZABLE isolation level. *S* stands for a general SQL statement executed by the *execsqli* or *execsqls* command.

4.3 Implementing the Testing Plan

4.3.1 Generator

We have devised a way to facilitate writing the histories we specified in the testing plan. Instead of creating all the possible histories, we create a template that contains groups of operations. The groups are defined so that their members could cause the same type of a conflict. A **generator** program processes this template and produces histories by combining operations from several groups. A user can provide the pairs of groups to be used for creating the desired combinations. The generator has been implemented as a PERL program.

The template contains several sections, each one starting with a "%BEGIN" keyword, followed by the name of the section in the same line and its body in subsequent lines. The end of the section is specified by the keyword "%END". The name of a section next to "%END" is optional.

The template we have used to implement our testing plan is shown in APPENDIX 1. The character (#) is used as a comment indicator.

The first section "%BEGIN INIT" contains declarations of predicate and item variables that will be used in the input histories.

```
%BEGIN INIT

pred,P,k2=0
pred,Q,"not (k2=0)"               # Q = not P
pr,P;recval;1;A,                  # A will be a row in P
pr,Q;recval;1;B,                  # B will be a row NOT in P
pr,P;recval;all,                  # This will close the cursor for P
pr,Q;recval;all,                  # This will close the cursor for Q
c,,

%END INIT
```

The data item variable A is associated with a row that satisfies the predicate P, and the data item variable B is associated with a row that satisfies predicate Q, which is the negation of P.

The next section "%BEGIN MATRIX" specifies the way the generator will combine the groups of operations for producing the required histories[21]. Each line in this section contains the names of the groups to be combined, followed by an optional string that will eventually be embedded in the name of the files, which will contain the input histories that are generated by following the given combination. The reason for using the tag in this template is to indicate the type of conflict that the operations in the groups ought to produce. As we will see later, this will make it easy to analyze the output histories and to determine whether an anomaly occurred.

```
%BEGIN MATRIX

1 1a w_w
1 2 w_r
4 3 w_pr
4 5 w_pr
2 1a r_w
3 4 pr_w
5 4 pr_w

%END MATRIX
```

The sections starting with "%BEGIN COMMON 1" and "%BEGIN COMMON 2" contain operations that will be placed in front of every operation of each group in the generated histories. What appears in COMMON 1 will be placed in front of the first operation of the generated pairs, and what appears in COMMON 2 will be placed in front of the second operation of the generated pairs.

[21] The GROUP declarations will follow.

```
%BEGIN COMMON 1
il,$IL1,
%END COMMON 1

%BEGIN COMMON 2
il,$IL2,
%END COMMON 2
```

In our case these sections contain operations that will set the isolation levels. Notice that the operands in these operations ($IL1 and $IL2) are HISTEX macros. In this way the generated histories will be such that they can be executed by using different combinations of isolation levels. This is done by providing a definition for each macro when executing HISTEX.

The following sections contain the definitions of the groups. Every group starts with the keywords “%BEGIN GROUP”, followed by its name. The operations contained in this section are similar to the HISTEX operations, but they do not contain the leading transaction id. The generator will fill in the transaction id when it produces the histories. This is because the operations in a given group could belong either to the first or to the second transaction of the output histories, and so their id is not fixed.

```
%BEGIN GROUP 1
w,D,111
rw,D,111
#D,D,
I,D,
%END GROUP 1

%BEGIN GROUP 1a

w,D,111
rw,D,111
D,D,

%END GROUP

%BEGIN GROUP 2
r,D,
%END GROUP 2


%BEGIN GROUP 3

pr,P;recval;all,

%END GROUP 3

# This group performs operations that change the matches of
# the predicate P : k2=0
%BEGIN GROUP 4
```

```
w,B;k2,0        # w(B into P)
w,A;k2,1        # w(A outof P)
rw,B;k2,k2-1    # rw(B into P)
rw,A;k2,k2+1    # rw(A outof P)
D,A,            # D(A in P)
I,C;recval;k2,15000;0  # I(C in P)

%END GROUP 4


%BEGIN GROUP 5

execsqli,update T set recval = recval + 1 where %P,

%END GROUP 5
```

The generator places the histories in files with the following name format:

<prefix>.<xx>.<pattern>.in

where <prefix> is the name of the template file, <xx> is a numeric id for every history produced, and <pattern> indicates the type of conflict that is contained in the history. The <pattern> is derived from the third column of the MATRIX in the template.

The following table shows the patterns we are currently using:

Pattern	Type of Conflict
w_w	item write / item write
w_r	item write / item read
r_w	item read / item write
w_pr	item write / predicate read
pr_w	predicate read / item write

The complete listing of the histories produced by this template is given in APPENDIX 2.

4.3.2 Executing the histories

The template we described above generates 39 input histories. We use HISTEX to execute these histories for all the database systems we are testing.

We are executing the histories in all possible combinations of isolation levels. In addition, we use various ways to create the underlying table, basically by either defining or not defining a primary key constraint for the reckey column and indexes for the k1-k100 columns. The reason for introducing these variations is to study the implication of indices in concurrency control, since we know from the literature that index structures are used extensively for acquiring high granularity locks. These are used as an attempt to implement the concept of a predicate lock.

The following table summarizes the parameters we have used:

Database System	**Combination of Isolation levels**	**Indices and Constraints**
IBM DB2 V5.2	CS_CS CS_RS CS_RR RS_CS RS_RS RS_RR RR_CS RR_RS RR_RR	prkey_index prkey_noindex noprekey_index noprekey_noindex
IBM DB2 V6.1	CS_CS CS_RS CS_RR RS_CS RS_RS RS_RR RR_CS RR_RS RR_RR	prkey_index prkey_noindex noprekey_index noprekey_noindex
INFORMIX V9.1	RC_RC RC_SR SR_RC SR_SR	prkey_index prkey_noindex noprekey_index noprekey_noindex

The output history is placed in a file with the following pattern name:

<prefix>.<xx>.<pattern>.<dbsys>.<IL1>_<IL2>

The first three parts of the name are derived from the filename of the input history. <dbsys> indicates the database system that was used. We use **db2** for the IBM DB2 database system and **inf** for the[23] Informix one. <IL1> and <IL2> specify the isolation levels that were used.

4.3.4 Analyser

Having executed all the histories, we analyze the output histories to see if an anomaly has occurred. This is when the operations of the second transaction are executed even though this should be proscribed by a given isolation level.

When a history is executed, we expect that one of the following conditions will occur:

1. All operations will be executed without a problem (EXECUTED).
2. Some operation will be blocked, causing the history to timeout (TIMEOUT).
3. An error will be raised if the database detects that it couldn't serialize the execution (SQL_ERROR).

An **analyzer** program scans the output histories and produces a list with the terminating condition (EXECUTED, TIMEOUT, SQL_ERROR). In addition, the analyzer detects

[23] A database product version is not included in this name. This is because we decided to test several versions after the setup was created. We will be explicitly mentioning the version of a database product when necessary.

whether the execution of a history is an anomaly or the database system is over-restrictive in not allowing a history to execute.

The analysis is done by the following method: The conflict patterns w_w, w_r, r_w, w_pr and pr_w that are included in the history names divide the histories into classes.

For every database product, we define a table that specifies the combinations of the isolation levels that could allow the execution of the classes mentioned above.

In the general case (i.e., for a product that provides the exact isolation levels as they are specified by the ANSI SQL), the table would be:

History Class	**Isolation Levels (RC or above) allowing execution**
w_w	None
w_r	None
w_pr	None
r_w	RC_RC, RC_RR, RC_SR
pr_w	RC_RC, RR_RR RC_RR RR_RC RC_SR RR_SR

For the IBM DB2 product, where the naming of the isolation levels deviates from the standard, we are using the following table:

History Class	**Isolation Levels (RC or above) allowing execution**
w_w	None
w_r	None
w_pr	None
r_w	CS_CS CS_RS CS_RR
pr_w	CS_CS RS_RS CS_RS RS_CS CS_RR RS_RR

For the INFORMIX database product, where the Repeatable Read Isolation level behaves identically as the Serializable isolation level, we use the table:

History Class	**Isolation Levels allowing execution**
w_w	None
w_r	None
w_pr	None
r_w	RC_RC RC_SR
pr_w	RC_RC RC_SR

As we have mentioned, the conflict pattern of every history is recorded by the generator in the name of the history. In addition, the file name of the output history is extended with the names of the isolations levels under which the history was executed. The analyzer uses this information to examine whether the EXECUTED histories are included in the classes that are determined from the tables above. In that case it uses a (*) to flag this event in the output. For histories that were not executed, the analyzer, based on the same table, examines if it was valid for those histories to be executed. In that case it uses a (+) to flag this event in the output.

4.4 Results

4.4.1 Examining the READ UNCOMMITTED Isolation Level

IBM DB2 UDB V5.0

We have observed that it is possible for a transaction that runs at the UNCOMMITED READ isolation level to read an uncommitted value and then perform an update. This behavior is not allowed by the SQL standard since READ ONLY transactions are to be executed under this level.

It is interesting to note that there are cases where a transaction running at the UNCOMMITED READ isolation level will block when trying to read a row written by an uncommitted transaction. This is because only read operations performed by a **read only cursor** are not blocked. It is only in that case that a transaction is considered to execute in the UNCOMMITED READ isolation level. Otherwise, even though a user specifies the UR isolation level, the database system will execute the transaction in a higher isolation level.

The following are excerpts from the DB2 5.0 documentation [DB2.1]:

Definition of UNCOMMITTED READ:

For a SELECT INTO, FETCH with a read-only cursor, fullselect used in an INSERT, row fullselect in an UPDATE, or scalar fullselect (wherever used), level UR allows:

- Any row that is read during the unit of work to be changed by other application processes.
- Any row that was changed by another application process to be read even if the change has not been committed by that application process.

For other operations, the rules of level CS apply.

A cursor is *read-only* if it is not deletable.

A cursor is *deletable* if all of the following are true:

- Each FROM clause of the outer fullselect identifies only one base table or deletable view (cannot identify a nested or common table expression)
- The outer fullselect does not include a VALUES clause
- The outer fullselect does not include a GROUP BY clause or HAVING clause
- The outer fullselect does not include column functions in the select list
- The outer fullselect does not include SET operations (UNION, EXCEPT, or INTERSECT, with or

without the ALL option)

- The select list of the outer fullselect does not include DISTINCT
- The select-statement does not include an ORDER BY clause
- The select-statement does not include a FOR READ ONLY clause
- One or more of the following is true:
    - the FOR UPDATE clause is specified
    - the cursor is statically defined
    - the LANGLEVEL bind option is MIA or SQL92E

In our case we have explicitly used the "FOR READ ONLY" clause in a select statement.

The following is the result of a history we have executed for examining this behavior:

```
(map, A, 100)
(map, B, 200)
(1, r, A [=100],  [=10000])
(1, r, B [=200],  [=20000])
(1, c)
(2, w, A [=100],  [=1730691225])
(3, execsqls, select recval from T where reckey = 100 FOR READ ONLY, A0 [=1730691225])
(3, w, B [=200], A0 [=1730691225])
(3, c)
(2, a)
(4, r, A [=100],  [=10000])
(4, r, B [=200],  [=1730691225])
(4, c)
```

Transaction 1 just reads the values of rows A and B. Transaction 2 writes a random value (1730691225) in row A. Transaction 3 reads the value of row A, writes it in row B, and commits. Note that at this point the value read is not committed. In what follows, transaction 2 aborts. The last transaction re-reads the values of rows A and B where we can verify that an update based on uncommitted read values has occurred.

IBM DB2 UDB 6.1

We have observed that this version of IBM DB2 allows a transaction that executes at the READ UNCOMMITTED isolation level to perform an update based on an uncommitted value. The ISO SQL standard in general does not allow transactions that execute at the READ UNCOMMITTED level to perform updates.

INFORMIX

The informix database product we tested would not allow a transaction executing at the READ UNCOMMITTED level to perform an update in general.

4.4.2 Examining the remaining Isolation levels

IBM DB2 UDB V5.0

For IBM DB2 5.0, as it appears in the following excerpt of the analyzer output, we observe that DB2 allows execution of some histories in the **w_pr** class where this should be proscribed. This can be considered an anomaly under our interpretation of this isolation level.

```
Output file           prkey       prkey       noprkey     noprkey
                      index       noindex     index       noindex
-----------------------------------------------------------------
h.14.w_pr.db2.CS_CS : EXECUTED* : TIMEOUT   : EXECUTED* : TIMEOUT
h.14.w_pr.db2.CS_RS : EXECUTED* : TIMEOUT   : EXECUTED* : TIMEOUT
h.14.w_pr.db2.RR_CS : EXECUTED* : TIMEOUT   : EXECUTED* : TIMEOUT
h.14.w_pr.db2.RR_RS : EXECUTED* : TIMEOUT   : EXECUTED* : TIMEOUT
h.14.w_pr.db2.RS_CS : EXECUTED* : TIMEOUT   : EXECUTED* : TIMEOUT
h.14.w_pr.db2.RS_RS : EXECUTED* : TIMEOUT   : EXECUTED* : TIMEOUT
h.16.w_pr.db2.CS_CS : EXECUTED* : TIMEOUT   : EXECUTED* : TIMEOUT
h.16.w_pr.db2.CS_RS : EXECUTED* : TIMEOUT   : EXECUTED* : TIMEOUT
h.16.w_pr.db2.RR_CS : EXECUTED* : TIMEOUT   : EXECUTED* : TIMEOUT
h.16.w_pr.db2.RR_RS : EXECUTED* : TIMEOUT   : EXECUTED* : TIMEOUT
h.16.w_pr.db2.RS_CS : EXECUTED* : TIMEOUT   : EXECUTED* : TIMEOUT
h.16.w_pr.db2.RS_RS : EXECUTED* : TIMEOUT   : EXECUTED* : TIMEOUT
h.17.w_pr.db2.CS_CS : EXECUTED* : EXECUTED* : EXECUTED* : EXECUTED*
h.17.w_pr.db2.CS_RS : EXECUTED* : EXECUTED* : EXECUTED* : EXECUTED*
h.17.w_pr.db2.RR_CS : EXECUTED* : EXECUTED* : EXECUTED* : EXECUTED*
h.17.w_pr.db2.RR_RS : EXECUTED* : EXECUTED* : EXECUTED* : EXECUTED*
h.17.w_pr.db2.RS_CS : EXECUTED* : EXECUTED* : EXECUTED* : EXECUTED*
h.17.w_pr.db2.RS_RS : EXECUTED* : EXECUTED* : EXECUTED* : EXECUTED*
h.23.w_pr.db2.CS_CS : TIMEOUT   : TIMEOUT   : TIMEOUT   : EXECUTED*
h.23.w_pr.db2.CS_RS : TIMEOUT   : TIMEOUT   : TIMEOUT   : EXECUTED*
h.23.w_pr.db2.RS_CS : TIMEOUT   : TIMEOUT   : TIMEOUT   : EXECUTED*
h.23.w_pr.db2.RS_RS : TIMEOUT   : TIMEOUT   : TIMEOUT   : EXECUTED*
```

All the above cases are related to the fact that some histories containing an item write/predicate read conflict are executed when the second transaction (the one issuing the predicate operation) runs at the CS or RS isolation level (these correspond to the ANSI READ COMMITTED and ANSI REPEATABLE READ).

According to our interpretation of the ANSI SQL definitions, this behavior should be proscribed because transactions running at these levels should only observe a committed state of the database. But in the cases above they see the effects of uncommitted transactions.

The following is an analysis of our observations:

(a) The pairs *W(A outof P)PR(P)* and *RW(A outof P)PR(P)* are allowed execution at the CS and RS levels <u>when an index for the column mentioned in the predicate exists</u>. (Anomaly)

This is shown by the output of histories 14 and 16

Under the assumption that the CS and RS levels should hold short read predicate locks, this behavior can be considered an anomaly.

We would also like to explain why the histories are not executed if an index for the k2 column is absent.

We could identify the following possible scenarios:

i. Since indexes are absent, DB2, when trying to perform a write operation, should lock the whole table in order to prevent a possible concurrent transaction running at the Serializable Isolation level from seeing a phantom.

If this scenario holds, then we expect that for all histories that are executed at the CS or RS level, and for which the first operation is a write, the second operation should block. This, in general, is true, except in the case where the first operation is a delete.

It is interesting to notice that if the second operation is a SetUpdate, it will not block. This is actually serializable behavior because if the row is deleted, it does not matter whether it is deleted before or after.

ii. The first transaction locks the row, and because the second predicate read needs to scan the table for predicate evaluation, it is possible that it will try to read the locked row to see if it matches the predicate, and in that case it will block.

If this is the case, we would expect that any predicate operation should conflict with any item write operation, regardless whether there is an item-write/predicate-read conflict.

(b) A pair D(A in P) PR(P) is allowed execution at the CS and RS levels (Possible Anomaly).

This is also related to the previous case. It appears that for a deletion of a row there is no lock acquired.

This is shown in the outputs of history 17.

(c) A pair D(A in P) SetUpdate(P) is allowed execution at the CS and RS levels, when <u>no indexes and no prkey are used</u>.

This is shown by the results of history 23.

Although this behavior shows that an item-write/predicate-read conflict exists due to the embedded predicate read operation of the Set Update, it can be argued that there is not a conflict because the result of the operation will be the same regardless of the order in which the operations will be executed.

(d) Interesting Observations for PR(P)WRITE(A changes P) category.

The RS level, even though it allows a W(A into P), a RW(A into P), or an I(A in P) to be executed after a PR(P), does not allow a W(A outof P), a RW(A outof P), or a D(A in P).

This is shown by the results of the histories 21-26.

This behavior is not considered an anomaly since the RS level is not meant to hold long term read predicate locks; but it is interesting to notice the difference in behavior depending on the type of the write operation.

An explanation here is that when an update is such that it inserts a row into a predicate, the blocking occurs because the predicate operation will attempt to read those rows, and this is not permitted under the RS isolation level.

(e) The pairs SetUpdate(P)W(A into P) and SetUpdate(P) RW(A into P) are allowed execution at the CS and RS levels only <u>at the presence of a primary key and the absence of indexes</u>.

This is shown in the output of history 36.

(f) A pair SetUpdate(P) I(A in P) is allowed execution at the CS and RS levels ONLY at the absence of indexes.

This is shown in the output of history 39, and is not considered an anomaly because a set update is not required to hold a long-term predicate read lock.

IBM DB2 UDB V6.1

We have executed the same histories under a more recent version of DB2, namely 6.1. We have noticed that all the behavior that we have considered an anomaly in the previous version is not present any more. This is also a strong indication that our interpretation of the ANSI definition was correct. In addition, we have also noticed that this version increased its precision because some histories that were not executed before under a given isolation, without this being considered an anomaly, are executed now.

INFORMIX

We have executed the histories mentioned above by using the INFORMIX database system. This system does not allow any conflicting pair of operations that should be proscribed by a given isolation level to be executed under that level.

Also, we have not observed any difference in the executed histories, regardless of whether there were indexes or primary key constrains used for the underlying table.

4.5 Other applications and suggested extensions

Extensions to the current approach

The test cases we have described in this work are not an exhaustive test for commercial database systems. A natural extension to this work would be to create test cases by using more complex predicates. One challenge in doing this would be to identify the data items that would produce the necessary conflicts. An automatic way could be devised for creating complex predicates and for identifying the data items and the update information for causing predicate conflicts.

The output of this process could generate templates similar to the one contained in the thesis. The generator could be used to execute those histories and the analyzer to detect any anomaly. The execsqli and execsqls statements could be used to exercise other SQL constructs that are not being covered by the supported commands.

Testing multi-version systems

The type of testing we described above is not pertinent if the underlying system uses multiple versions. In such a case, if we specified a pair of conflicting operations such as a WRITE followed by a READ of the same data item, a multi-version system could execute the read operation by using a previous version. So both operations could execute without actually causing a problem. Even though we have not defined a systematic way for testing a multi-version system, an investigator can create interesting histories which might demonstrate an anomaly. We have created histories to demonstrate cases where the SERIALIZABLE level of the Oracle Database product (version 8.1.6) does not provide full serializability.

One way of developing a systematic way for testing multi-version database systems could rely on detecting cycles in output histories, similar to the ones described in [ALO00]. As we have mentioned earlier, our output histories do not contain all the information required for detecting such cycles because such information is not provided at the application level. Some of these problems could be resolved by utilizing assumptions of the underlying database systems. For example, if we would like to develop some testing for an Oracle database system, we could assume that the version order is the same as the order in which the update statements occurred. In order to detect what version of data is read any time, we could extend our data model by including in every row a version field that could contain the transaction id. When a row is read, the value of the version field can be read and included in the output history.

Other applications

We have also used HISTEX as a tool for driving performance measurement tests for a concurrency control protocol that fixes some of the problems in the Snapshot Isolation level provided by database systems like Oracle [FLOOS00]. For these experiments we have utilized a feature of HISTEX that allows an asynchronous[25] execution of the operations contained in a history.

[25] The default mode is a synchronous execution where an operation will be executed after any previous operation was completed.

APPENDICES

APPENDIX 0 – HISTEX OPERATIONS AND DATA MODEL

Data Model

RECKEY	RECVAL	C2	C3	C4	C5	C6	C50	C100	K2	K3	K4	K5	K6	K50	K100
100	1000	0	0	0	0	0	0	0	0	0	0	0	0	0	0
200	2000	1	1	1	1	1	1	1	1	1	1	1	1	1	1
300	3000	0	2	2	2	2	2	2	0	2	2	2	2	2	2
400	4000	1	0	3	3	3	3	3	1	0	3	3	3	3	3
							…								

HISTEX Operations

Text Book Notation	Functionality	Examples of Implemented Notation
A_i	Abort	1,a,,
I_i(A[;col1[;col2 ...]] [,num[;num ...]])	Insert Row	1,I,A;c2,4
C_i	Commit	1,c,,
D_i(A)	Delete Row	1,D,A,
EXECSQLI$_i$(statement)	Execute a SQL statement (immediate mode)	1,execsqli,"update T set recval = recval + 1 where %P",
EXECSQLS$_i$(statement)	Execute a SQL statement (open cursor)	1,execsqls,"select sum(recval) from T group by k1, k2 having %P",
IL$_i$(UR|CS|RS|RR)	Isolation Level (DB2)	1,il,UR,
IL$_i$(RU|RC|RR|SR)	Isolation Level (Informix)	1,il,RC,
IL$_i$(RC|SR)	Isolation Level (Oracle)	1,il,SR,
MAP(A,ID)	Map a row	0,map,A,100
PRED(P,predicate)	Predicate Declaration	0,pred,P,*" k2=1 and k3<2"*
PR$_i$(P;col;i[;A] [,X])	Predicate Read	1,pr,P;recval;1;A,A0
R$_i$(A[;column][,X])	Read a row	1,r,A,A0
RW$_i$(A[;column][,exp]	Read & update a row	1,rw,A;k2,k2+k3
W$_i$(A[;column][,{X|num}])	Update a row	1,w,A,1001

APPENDIX 1 – The template for generating the histories

```
%BEGIN INIT

pred,P,k2=0
pred,Q,"not (k2=0)"                 # Q = not P
pr,P;recval;1;A,                    # A will be a row in P
pr,Q;recval;1;B,                    # B will be a row NOT in P
pr,P;recval;all,                    # This will close the cursor for P
pr,Q;recval;all,                    # This will close the cursor for Q
c,,

%END INIT

%BEGIN MATRIX

1 1a w_w
1 2 w_r
4 3 w_pr
4 5 w_pr
2 1a r_w
3 4 pr_w
5 4 pr_w

%END MATRIX


%BEGIN COMMON 1
il,$IL1,
%END COMMON 1

%BEGIN COMMON 2
il,$IL2,
%END COMMON 2

%BEGIN GROUP 1
w,D,111
rw,D,111
#D,D,
I,D,
%END GROUP 1

%BEGIN GROUP 1a

w,D,111
rw,D,111
D,D,

%END GROUP

%BEGIN GROUP 2
r,D,
%END GROUP 2
```

```
%BEGIN GROUP 3

pr,P;recval;all,

%END GROUP 3

# This group performs operations that change the matches of the predicate
# P : k2=0

%BEGIN GROUP 4

w,B;k2,0        # w(B into P)
w,A;k2,1        # w(A outof P)
rw,B;k2,k2-1    # rw(B into P)
rw,A;k2,k2+1    # rw(A outof P)
D,A,            # D(A in P)
I,C;recval;k2,15000;0  # I(C in P)

%END GROUP 4


%BEGIN GROUP 5

execsqli,update T set recval = recval + 1 where %P,

%END GROUP 5
```

APPENDIX 2 – The input histories

```
::::::::::::::
h.01.w_w.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,w,D,111
2,il,$IL2,
2,w,D,111
::::::::::::::
h.02.w_w.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,w,D,111
2,il,$IL2,
2,rw,D,111
::::::::::::::
h.03.w_w.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,w,D,111
2,il,$IL2,
2,D,D,
::::::::::::::
h.04.w_w.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,rw,D,111
2,il,$IL2,
2,w,D,111
::::::::::::::
h.05.w_w.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,rw,D,111
2,il,$IL2,
2,rw,D,111
::::::::::::::
h.06.w_w.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,rw,D,111
2,il,$IL2,
2,D,D,
::::::::::::::
h.07.w_w.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,I,D,
2,il,$IL2,
2,w,D,111
::::::::::::::
h.08.w_w.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,I,D,
2,il,$IL2,
2,rw,D,111
::::::::::::::
h.09.w_w.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,I,D,
2,il,$IL2,
2,D,D,
::::::::::::::
h.10.w_r.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,w,D,111
```

```
2,il,$IL2,
2,r,D,
::::::::::::::
h.11.w_r.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,rw,D,111
2,il,$IL2,
2,r,D,
::::::::::::::
h.12.w_r.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,I,D,
2,il,$IL2,
2,r,D,
::::::::::::::
h.13.w_pr.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,w,B;k2,0
2,il,$IL2,
2,pr,P;recval;all,
::::::::::::::
h.14.w_pr.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,w,A;k2,1
2,il,$IL2,
2,pr,P;recval;all,
::::::::::::::
h.15.w_pr.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,rw,B;k2,k2-1
2,il,$IL2,
```

```
2,pr,P;recval;all,
::::::::::::::
h.16.w_pr.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,rw,A;k2,k2+1
2,il,$IL2,
2,pr,P;recval;all,
::::::::::::::
h.17.w_pr.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,D,A,
2,il,$IL2,
2,pr,P;recval;all,
::::::::::::::
h.18.w_pr.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,I,C;recval;k2,15000;0
2,il,$IL2,
2,pr,P;recval;all,
::::::::::::::
h.19.w_pr.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,w,B;k2,0
2,il,$IL2,
2,execsqli,update T set recval = recval + 1
where %P,
::::::::::::::
h.20.w_pr.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,w,A;k2,1
2,il,$IL2,
2,execsqli,update T set recval = recval + 1
where %P,
::::::::::::::
h.21.w_pr.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,rw,B;k2,k2-1
2,il,$IL2,
2,execsqli,update T set recval = recval + 1
where %P,
::::::::::::::
h.22.w_pr.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,rw,A;k2,k2+1
2,il,$IL2,
2,execsqli,update T set recval = recval + 1
where %P,
::::::::::::::
h.23.w_pr.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,D,A,
2,il,$IL2,
2,execsqli,update T set recval = recval + 1
where %P,
::::::::::::::
h.24.w_pr.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,I,C;recval;k2,15000;0
2,il,$IL2,
2,execsqli,update T set recval = recval + 1
where %P,
::::::::::::::
h.25.r_w.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,r,D,
2,il,$IL2,
2,w,D,111
::::::::::::::
h.26.r_w.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,r,D,
2,il,$IL2,
2,rw,D,111
::::::::::::::
h.27.r_w.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,r,D,
2,il,$IL2,
2,D,D,
::::::::::::::
h.28.pr_w.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,pr,P;recval;all,
2,il,$IL2,
2,w,B;k2,0
::::::::::::::
h.29.pr_w.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,pr,P;recval;all,
2,il,$IL2,
2,w,A;k2,1
::::::::::::::
h.30.pr_w.in
::::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,pr,P;recval;all,
2,il,$IL2,
2,rw,B;k2,k2-1
:::::::::::::
h.31.pr_w.in
:::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,pr,P;recval;all,
2,il,$IL2,
2,rw,A;k2,k2+1
:::::::::::::
h.32.pr_w.in
:::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,pr,P;recval;all,
2,il,$IL2,
2,D,A,
:::::::::::::
h.33.pr_w.in
:::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,pr,P;recval;all,
2,il,$IL2,
2,I,C;recval;k2,15000;0
:::::::::::::
h.34.pr_w.in
:::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,execsqli,update T set recval = recval + 1
where %P,
2,il,$IL2,
2,w,B;k2,0
:::::::::::::
h.35.pr_w.in
:::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,execsqli,update T set recval = recval + 1
where %P,
2,il,$IL2,
2,w,A;k2,1
:::::::::::::
h.36.pr_w.in
:::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,execsqli,update T set recval = recval + 1
where %P,
2,il,$IL2,
2,rw,B;k2,k2-1
:::::::::::::
h.37.pr_w.in
:::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,execsqli,update T set recval = recval + 1
where %P,
2,il,$IL2,
2,rw,A;k2,k2+1
:::::::::::::
h.38.pr_w.in
:::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,execsqli,update T set recval = recval + 1
where %P,
2,il,$IL2,
2,D,A,
:::::::::::::
h.39.pr_w.in
:::::::::::::
0,pred,P,k2=0
0,pred,Q,"not (k2=0)"
0,pr,P;recval;1;A,
0,pr,Q;recval;1;B,
0,pr,P;recval;all,
0,pr,Q;recval;all,
0,c,,
1,il,$IL1,
1,execsqli,update T set recval = recval + 1
where %P,
2,il,$IL2,
2,I,C;recval;k2,15000;0
```

APPENDIX 3 – Analyzed output for IBM DB2 UDB Version 5.0

```
h.01.w_w.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.01.w_w.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.01.w_w.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.01.w_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.01.w_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.01.w_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.01.w_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.01.w_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.01.w_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.14.w_pr.db2.CS_CS : EXECUTED* : TIMEOUT : EXECUTED* : TIMEOUT
h.14.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.14.w_pr.db2.CS_RS : EXECUTED* : TIMEOUT : EXECUTED* : TIMEOUT
h.14.w_pr.db2.RR_CS : EXECUTED* : TIMEOUT : EXECUTED* : TIMEOUT
h.14.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.14.w_pr.db2.RR_RS : EXECUTED* : TIMEOUT : EXECUTED* : TIMEOUT
h.14.w_pr.db2.RS_CS : EXECUTED* : TIMEOUT : EXECUTED* : TIMEOUT
h.14.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.14.w_pr.db2.RS_RS : EXECUTED* : TIMEOUT : EXECUTED* : TIMEOUT
h.15.w_pr.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.15.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.15.w_pr.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.15.w_pr.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.15.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.15.w_pr.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
```

```
h.15.w_pr.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.15.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.15.w_pr.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.16.w_pr.db2.CS_CS : EXECUTED* : TIMEOUT : EXECUTED* : TIMEOUT
h.16.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.16.w_pr.db2.CS_RS : EXECUTED* : TIMEOUT : EXECUTED* : TIMEOUT
h.16.w_pr.db2.RR_CS : EXECUTED* : TIMEOUT : EXECUTED* : TIMEOUT
h.16.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.16.w_pr.db2.RR_RS : EXECUTED* : TIMEOUT : EXECUTED* : TIMEOUT
h.16.w_pr.db2.RS_CS : EXECUTED* : TIMEOUT : EXECUTED* : TIMEOUT
h.16.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.16.w_pr.db2.RS_RS : EXECUTED* : TIMEOUT : EXECUTED* : TIMEOUT
h.17.w_pr.db2.CS_CS : EXECUTED* : EXECUTED* : EXECUTED* : EXECUTED*
h.17.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.17.w_pr.db2.CS_RS : EXECUTED* : EXECUTED* : EXECUTED* : EXECUTED*
h.17.w_pr.db2.RR_CS : EXECUTED* : EXECUTED* : EXECUTED* : EXECUTED*
h.17.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.17.w_pr.db2.RR_RS : EXECUTED* : EXECUTED* : EXECUTED* : EXECUTED*
h.17.w_pr.db2.RS_CS : EXECUTED* : EXECUTED* : EXECUTED* : EXECUTED*
h.17.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.17.w_pr.db2.RS_RS : EXECUTED* : EXECUTED* : EXECUTED* : EXECUTED*
h.18.w_pr.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.18.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.18.w_pr.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.18.w_pr.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.18.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.18.w_pr.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.18.w_pr.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.18.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.18.w_pr.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.23.w_pr.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : EXECUTED*
h.23.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.23.w_pr.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : EXECUTED*
h.23.w_pr.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
```

```
h.23.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.23.w_pr.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.23.w_pr.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : EXECUTED*
h.23.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.23.w_pr.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : EXECUTED*
h.24.w_pr.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.24.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.24.w_pr.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.24.w_pr.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.24.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.24.w_pr.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.24.w_pr.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.24.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.24.w_pr.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.25.r_w.db2.CS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.25.r_w.db2.CS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.25.r_w.db2.CS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.25.r_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.25.r_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.25.r_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.25.r_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.25.r_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.25.r_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.26.r_w.db2.CS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.26.r_w.db2.CS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.26.r_w.db2.CS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.26.r_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.26.r_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.26.r_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.26.r_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.26.r_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.26.r_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.27.r_w.db2.CS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.27.r_w.db2.CS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.27.r_w.db2.CS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.27.r_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.27.r_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.27.r_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.27.r_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.27.r_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.27.r_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.28.pr_w.db2.CS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.28.pr_w.db2.CS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.28.pr_w.db2.CS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.28.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.28.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.28.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.28.pr_w.db2.RS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.28.pr_w.db2.RS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.28.pr_w.db2.RS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.29.pr_w.db2.CS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.29.pr_w.db2.CS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.29.pr_w.db2.CS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.29.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.29.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.29.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.29.pr_w.db2.RS_CS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.29.pr_w.db2.RS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.29.pr_w.db2.RS_RS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.30.pr_w.db2.CS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.30.pr_w.db2.CS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.30.pr_w.db2.CS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.30.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.30.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.30.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.30.pr_w.db2.RS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.30.pr_w.db2.RS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.30.pr_w.db2.RS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.31.pr_w.db2.CS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.31.pr_w.db2.CS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
```

```
h.31.pr_w.db2.CS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.31.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.31.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.31.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.31.pr_w.db2.RS_CS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.31.pr_w.db2.RS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.31.pr_w.db2.RS_RS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.32.pr_w.db2.CS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.32.pr_w.db2.CS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.32.pr_w.db2.CS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.32.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.32.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.32.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.32.pr_w.db2.RS_CS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.32.pr_w.db2.RS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.32.pr_w.db2.RS_RS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.33.pr_w.db2.CS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.33.pr_w.db2.CS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.33.pr_w.db2.CS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.33.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.33.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.33.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.33.pr_w.db2.RS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.33.pr_w.db2.RS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.33.pr_w.db2.RS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.34.pr_w.db2.CS_CS : TIMEOUT+ : EXECUTED : TIMEOUT+ : TIMEOUT+
h.34.pr_w.db2.CS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.34.pr_w.db2.CS_RS : TIMEOUT+ : EXECUTED : TIMEOUT+ : TIMEOUT+
h.34.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.34.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.34.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.34.pr_w.db2.RS_CS : TIMEOUT+ : EXECUTED : TIMEOUT+ : TIMEOUT+
h.34.pr_w.db2.RS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.34.pr_w.db2.RS_RS : TIMEOUT+ : EXECUTED : TIMEOUT+ : TIMEOUT+
h.35.pr_w.db2.CS_CS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.35.pr_w.db2.CS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.35.pr_w.db2.CS_RS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.35.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.35.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.35.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.35.pr_w.db2.RS_CS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.35.pr_w.db2.RS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.35.pr_w.db2.RS_RS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.36.pr_w.db2.CS_CS : TIMEOUT+ : EXECUTED : TIMEOUT+ : TIMEOUT+
h.36.pr_w.db2.CS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.36.pr_w.db2.CS_RS : TIMEOUT+ : EXECUTED : TIMEOUT+ : TIMEOUT+
h.36.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.36.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.36.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.36.pr_w.db2.RS_CS : TIMEOUT+ : EXECUTED : TIMEOUT+ : TIMEOUT+
h.36.pr_w.db2.RS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.36.pr_w.db2.RS_RS : TIMEOUT+ : EXECUTED : TIMEOUT+ : TIMEOUT+
h.37.pr_w.db2.CS_CS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.37.pr_w.db2.CS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.37.pr_w.db2.CS_RS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.37.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.37.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.37.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.37.pr_w.db2.RS_CS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.37.pr_w.db2.RS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.37.pr_w.db2.RS_RS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.38.pr_w.db2.CS_CS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.38.pr_w.db2.CS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.38.pr_w.db2.CS_RS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.38.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.38.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.38.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.38.pr_w.db2.RS_CS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.38.pr_w.db2.RS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.38.pr_w.db2.RS_RS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.39.pr_w.db2.CS_CS : TIMEOUT+ : EXECUTED : TIMEOUT+ : EXECUTED
h.39.pr_w.db2.CS_RR : TIMEOUT+ : EXECUTED : TIMEOUT+ : EXECUTED
h.39.pr_w.db2.CS_RS : TIMEOUT+ : EXECUTED : TIMEOUT+ : EXECUTED
h.39.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.39.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.39.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.39.pr_w.db2.RS_CS : TIMEOUT+ : EXECUTED : TIMEOUT+ : EXECUTED
h.39.pr_w.db2.RS_RR : TIMEOUT+ : EXECUTED : TIMEOUT+ : EXECUTED
h.39.pr_w.db2.RS_RS : TIMEOUT+ : EXECUTED : TIMEOUT+ : EXECUTED
```

APPENDIX 4 – Analyzed Output for IBM DB2 Version 6.1

```
h.01.w_w.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.01.w_w.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.01.w_w.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.01.w_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.01.w_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.01.w_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.01.w_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.01.w_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.01.w_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.14.w_pr.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.14.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.14.w_pr.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.14.w_pr.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.14.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.14.w_pr.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.14.w_pr.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.14.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.14.w_pr.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.15.w_pr.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.15.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.15.w_pr.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.15.w_pr.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.15.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.15.w_pr.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.15.w_pr.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.15.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
```

```
h.15.w_pr.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.16.w_pr.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.16.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.16.w_pr.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.16.w_pr.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.16.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.16.w_pr.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.16.w_pr.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.16.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.16.w_pr.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.17.w_pr.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.17.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.17.w_pr.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.17.w_pr.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.17.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.17.w_pr.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.17.w_pr.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.17.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.17.w_pr.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.18.w_pr.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.18.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.18.w_pr.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.18.w_pr.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.18.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.18.w_pr.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.18.w_pr.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.18.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.18.w_pr.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.23.w_pr.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.23.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.23.w_pr.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.23.w_pr.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.23.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.23.w_pr.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
```

```
h.23.w_pr.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.23.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.23.w_pr.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.24.w_pr.db2.CS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.24.w_pr.db2.CS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.24.w_pr.db2.CS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.24.w_pr.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.24.w_pr.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.24.w_pr.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.24.w_pr.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.24.w_pr.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.24.w_pr.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.25.r_w.db2.CS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.25.r_w.db2.CS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.25.r_w.db2.CS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.25.r_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.25.r_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.25.r_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.25.r_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.25.r_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.25.r_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.26.r_w.db2.CS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.26.r_w.db2.CS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.26.r_w.db2.CS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.26.r_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.26.r_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.26.r_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.26.r_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.26.r_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.26.r_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.27.r_w.db2.CS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.27.r_w.db2.CS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.27.r_w.db2.CS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.27.r_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.27.r_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.27.r_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.27.r_w.db2.RS_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.27.r_w.db2.RS_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.27.r_w.db2.RS_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.28.pr_w.db2.CS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.28.pr_w.db2.CS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.28.pr_w.db2.CS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.28.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.28.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.28.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.28.pr_w.db2.RS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.28.pr_w.db2.RS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.28.pr_w.db2.RS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.29.pr_w.db2.CS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.29.pr_w.db2.CS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.29.pr_w.db2.CS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.29.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.29.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.29.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.29.pr_w.db2.RS_CS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.29.pr_w.db2.RS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.29.pr_w.db2.RS_RS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.30.pr_w.db2.CS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.30.pr_w.db2.CS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.30.pr_w.db2.CS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.30.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.30.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.30.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.30.pr_w.db2.RS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.30.pr_w.db2.RS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.30.pr_w.db2.RS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.31.pr_w.db2.CS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.31.pr_w.db2.CS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.31.pr_w.db2.CS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.31.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
```

```
h.31.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.31.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.31.pr_w.db2.RS_CS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.31.pr_w.db2.RS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.31.pr_w.db2.RS_RS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.32.pr_w.db2.CS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.32.pr_w.db2.CS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.32.pr_w.db2.CS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.32.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.32.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.32.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.32.pr_w.db2.RS_CS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.32.pr_w.db2.RS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.32.pr_w.db2.RS_RS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.33.pr_w.db2.CS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.33.pr_w.db2.CS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.33.pr_w.db2.CS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.33.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.33.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.33.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.33.pr_w.db2.RS_CS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.33.pr_w.db2.RS_RR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.33.pr_w.db2.RS_RS : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.34.pr_w.db2.CS_CS : TIMEOUT+ : EXECUTED : TIMEOUT+ : TIMEOUT+
h.34.pr_w.db2.CS_RR : TIMEOUT+ : EXECUTED : TIMEOUT+ : TIMEOUT+
h.34.pr_w.db2.CS_RS : TIMEOUT+ : EXECUTED : TIMEOUT+ : TIMEOUT+
h.34.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.34.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.34.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.34.pr_w.db2.RS_CS : TIMEOUT+ : EXECUTED : TIMEOUT+ : TIMEOUT+
h.34.pr_w.db2.RS_RR : TIMEOUT+ : EXECUTED : TIMEOUT+ : TIMEOUT+
h.34.pr_w.db2.RS_RS : TIMEOUT+ : EXECUTED : TIMEOUT+ : TIMEOUT+
h.35.pr_w.db2.CS_CS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.35.pr_w.db2.CS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.35.pr_w.db2.CS_RS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.35.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.35.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.35.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.35.pr_w.db2.RS_CS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.35.pr_w.db2.RS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.35.pr_w.db2.RS_RS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.36.pr_w.db2.CS_CS : TIMEOUT+ : EXECUTED : TIMEOUT+ : TIMEOUT+
h.36.pr_w.db2.CS_RR : TIMEOUT+ : EXECUTED : TIMEOUT+ : TIMEOUT+
h.36.pr_w.db2.CS_RS : TIMEOUT+ : EXECUTED : TIMEOUT+ : TIMEOUT+
h.36.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.36.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.36.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.36.pr_w.db2.RS_CS : TIMEOUT+ : EXECUTED : TIMEOUT+ : TIMEOUT+
h.36.pr_w.db2.RS_RR : TIMEOUT+ : EXECUTED : TIMEOUT+ : TIMEOUT+
h.36.pr_w.db2.RS_RS : TIMEOUT+ : EXECUTED : TIMEOUT+ : TIMEOUT+
h.37.pr_w.db2.CS_CS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.37.pr_w.db2.CS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.37.pr_w.db2.CS_RS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.37.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.37.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.37.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.37.pr_w.db2.RS_CS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.37.pr_w.db2.RS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.37.pr_w.db2.RS_RS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.38.pr_w.db2.CS_CS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.38.pr_w.db2.CS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.38.pr_w.db2.CS_RS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.38.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.38.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.38.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.38.pr_w.db2.RS_CS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.38.pr_w.db2.RS_RR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.38.pr_w.db2.RS_RS : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.39.pr_w.db2.CS_CS : TIMEOUT+ : EXECUTED : TIMEOUT+ : EXECUTED
h.39.pr_w.db2.CS_RR : TIMEOUT+ : EXECUTED : TIMEOUT+ : EXECUTED
h.39.pr_w.db2.CS_RS : TIMEOUT+ : EXECUTED : TIMEOUT+ : EXECUTED
h.39.pr_w.db2.RR_CS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.39.pr_w.db2.RR_RR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.39.pr_w.db2.RR_RS : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.39.pr_w.db2.RS_CS : TIMEOUT+ : EXECUTED : TIMEOUT+ : EXECUTED
h.39.pr_w.db2.RS_RR : TIMEOUT+ : EXECUTED : TIMEOUT+ : EXECUTED
h.39.pr_w.db2.RS_RS : TIMEOUT+ : EXECUTED : TIMEOUT+ : EXECUTED
```

APPENDIX – 5 Analyzed output for the Informix database

```
h.01.w_w.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.01.w_w.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.01.w_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.01.w_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.02.w_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.03.w_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.04.w_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.05.w_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.06.w_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.07.w_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.08.w_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.09.w_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.10.w_r.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.11.w_r.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.12.w_r.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.13.w_pr.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.14.w_pr.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.14.w_pr.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.14.w_pr.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.14.w_pr.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.15.w_pr.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.15.w_pr.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.15.w_pr.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.15.w_pr.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.16.w_pr.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.16.w_pr.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.16.w_pr.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.16.w_pr.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.17.w_pr.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.17.w_pr.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.17.w_pr.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.17.w_pr.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.18.w_pr.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.18.w_pr.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.18.w_pr.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.18.w_pr.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.19.w_pr.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.20.w_pr.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.21.w_pr.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.22.w_pr.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.23.w_pr.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.23.w_pr.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.23.w_pr.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.23.w_pr.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.24.w_pr.inf.RC_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.24.w_pr.inf.RC_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.24.w_pr.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.24.w_pr.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.25.r_w.inf.RC_RC : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.25.r_w.inf.RC_SR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.25.r_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.25.r_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.26.r_w.inf.RC_RC : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.26.r_w.inf.RC_SR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.26.r_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.26.r_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.27.r_w.inf.RC_RC : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.27.r_w.inf.RC_SR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.27.r_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.27.r_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.28.pr_w.inf.RC_RC : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.28.pr_w.inf.RC_SR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.28.pr_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.28.pr_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.29.pr_w.inf.RC_RC : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.29.pr_w.inf.RC_SR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.29.pr_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.29.pr_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.30.pr_w.inf.RC_RC : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.30.pr_w.inf.RC_SR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.30.pr_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.30.pr_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.31.pr_w.inf.RC_RC : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.31.pr_w.inf.RC_SR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.31.pr_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.31.pr_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.32.pr_w.inf.RC_RC : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.32.pr_w.inf.RC_SR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.32.pr_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.32.pr_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.33.pr_w.inf.RC_RC : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.33.pr_w.inf.RC_SR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.33.pr_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.33.pr_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.34.pr_w.inf.RC_RC : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.34.pr_w.inf.RC_SR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.34.pr_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.34.pr_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.35.pr_w.inf.RC_RC : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.35.pr_w.inf.RC_SR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.35.pr_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.35.pr_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.36.pr_w.inf.RC_RC : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.36.pr_w.inf.RC_SR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.36.pr_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.36.pr_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.37.pr_w.inf.RC_RC : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.37.pr_w.inf.RC_SR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.37.pr_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.37.pr_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.38.pr_w.inf.RC_RC : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.38.pr_w.inf.RC_SR : TIMEOUT+ : TIMEOUT+ : TIMEOUT+ : TIMEOUT+
h.38.pr_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.38.pr_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.39.pr_w.inf.RC_RC : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.39.pr_w.inf.RC_SR : EXECUTED : EXECUTED : EXECUTED : EXECUTED
h.39.pr_w.inf.SR_RC : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
h.39.pr_w.inf.SR_SR : TIMEOUT : TIMEOUT : TIMEOUT : TIMEOUT
```

REFERENCES

[ALO00] A. Adya, B. Liskov, P. O'Neil. Generalized Isolation Level Definitions. In Proceedings of IEEE International Conference on Data Engineering, March 2000.

[BBGMOO95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. In Proceedings of the SIGMOD International Conference on Management of Data, May 1995.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency Control and Recovery in Database Systems. Addison Wesley, 1987. (This text is now out of print but can be downloaded from http://research.microsoft.com/pubs/ccontrol/default.htm)

[CHETAL81] D. Chamberlain et al. A History and Evaluation of System R. CACM V24.10, pp. 632-646, Oct. 1981. (This paper is also in: Michael Stonebraker and Joseph Hellerstein, Readings in Database Systems, Third Edition, Morgan Kaufmann 1998.)

[DB2.1] DB2 Universal Database SQL Reference Manual, Version 5.0. Available at http://www.ibm.com/db2

[DB2.2] DB2 Universal Database SQL Reference Manual, Version 6. IBM, 1999. Available at http://www.ibm.com/db2

[EGLT76] K. P. Eswaran, J. Gray, R. Lorie, I. Traiger. The Notions of Consistency and Predicate Locks in a Database System. CACM V19.11, pp. 624- 633, Nov. 1976.

[FLOOS00] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha, "Making Snapshot Isolation Serializable," http://www.cs.umb.edu/~isotest/snaptest/snaptest.pdf

[GLPT77] J. Gray, R. Lorie, G. Putzolu and, I. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base," in Readings in Database Systems, Second Edition, Chapter 3, Michael Stonebraker, Ed., Morgan Kaufmann 1994 (originally published in 1977).

[GR93] J. N. Gray and A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers Inc., 1993.

[INF9.2] INFORMIX Guide to SQL Syntax, Version 9.2. Informix Press, 1999, Menlo Park, CA.

[ISO97] P. O'Neil and E. O'Neil. Isolation Testing. Proposal for NSF Grant IRI 97-11374, see <http://www.cs.umb.edu/~isotest> for Postscript download.

[JAC95] K. Jacobs, with contributors: R. Bamford, G. Doherty, K. Haas, M. Holt, F. Putzolu, B. Quigley. Concurrency Control: Transaction Isolation and Serializability in SQL92 and Oracle7. Oracle White Paper, Part No. A33745, July, 1995.

[JBB81] J. R. Jordan, J. Banerjee, and R. B. Batman. Precision Locks. Proceedings of the ACM SIGMOD International Conference on Management of Data, 1981, pp. 143-147.

[L98a] D. Liarokapis. A preliminary requirements specification for an Isolation Tester, www.cs.umb.edu/~dimitris/thesis.

[L98] D. Liarokapis. First Results by Using HISTEX, www.cs.umb.edu/~dimitris/thesis

[L00] D. Liarokapis. HISTEX ON LINE: Proposal for a Software Engineering Project. www.cs.umb.edu/~dimitris/thesis

[MOHAN90] C. Mohan. Aries/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. Proceedings of the 16th VLDB Conference, 1990, pp. 392-405.

[MOHAN92] C. Mohan. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. Proceedings of the ACM SIGMOD International Conference on Management of Data, June 1992, pp. 371-380.

[MOHAN96] C. Mohan. Concurrency Control and Recovery Methods for B+-Tree Indexes: ARIES/KVL and ARIES/IM. Performance of Concurrency Control Mechanisms in Centralized Database Systems, Vijay Kumar, Ed., Prentice-Hall 1996, pp. 248-306.

[O98] P. O'Neil. History Exerciser Utility: Specification. June 17, 1998. Draft. www.cs.umb.edu/~dimitris/thesis.

[OFOL00] P. O'Neil, A. Fekete, E. O'Neil, and D. Liarokapis. A New Hybrid Predicate-Assertion-Index-Locking (PAX) Algorithm to Prevent Transactional Phantoms (Preprint). http://www.cs.umb.edu/~poneil/paxext.pdf

[OO00] P. O'Neil, E O'Neil. Databases: Principles, Programming, Performance 2nd Edition. Morgan Kaufmann, May 2000.

[PAPA79] C. Papadimitriou. The Serializability of Concurrent Database Updates. Journal of the Association of Computing Machinery. Vol 26. No 4. October 1979, pp631-653.

[PAPA86] C. Papadimitriou. The Theory of Database Concurrency Control. Computer Science Press, 1986.

[SLSV95] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez, "Transaction Chopping: Algorithms and Performance Studies," ACM TODS, Vol. 20, No. 3, Sept. 1995, pp. 325-363.

[SLUTZ98] Don Slutz, "Massive Stochastic Testing of SQL,: Proceedings of the 24th VLDB Conference, 1998, pp. 618-622.

[SQL92] Information technology - Database languages – SQL ISO/IEC 9075:1992

[SQL99] Information technology – Database languages -- SQL -- Part 2: Foundation (SQL/Foundation) ISO/IEC 9075-2:1999