Tabling for Transaction Logic *

Paul Fodor Michael Kifer

State University of New York at Stony Brook,Stony Brook, NY 11794, USA

pfodor,kifer@cs.stonybrook.edu

Abstract

Transaction Logic is a logic for representing declarative and procedural knowledge in logic programming, databases, and AI. It has been successful in areas as diverse as workflows and Web services, security policies, AI planning, reasoning about actions, and more. Although a number of implementations of Transaction Logic exist, none is logically complete due to the inherent difficulty and time/space complexity of such implementations. In this paper we attack this problem by first introducing a logically complete tabling evaluation strategy for Transaction Logic and then describing a series of optimizations, which make this algorithm practical. In support of our arguments, we present a performance evaluation study of six different implementations of this algorithm, each successively adopting our optimizations. The study suggest that the tabling algorithm can scale well both in time and space. We also discuss ideas that could improve the performance further.

Categories and Subject Descriptors I.2.3 [*Artificial Intelligence*]: Deduction and Theorem Proving and Knowledge Processing; D.3.3 [*[Programming Languages*]: Language Constructs and Features frameworks; D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Algorithms

Keywords Transaction Logic, Tabling

1. Introduction

Transaction logic ($\mathcal{TR}$) ([5, 7, 9]) is a general logic of state change, which extends classical first-order logic with new connectives that make it suitable for representing both procedural and declarative knowledge. $\mathcal{TR}$ has been shown to be a powerful tool for reasoning about actions ([4, 10]), knowledge representation ([6]), event processing ([1]), workflow management and Semantic Web services ([15, 16, 27, 28]), AI planning ([3, 18]), security policy frameworks [2], and general knowledge base programming ([11]). In logic programming, $\mathcal{TR}$ provides a clean, logical alternative to the *assert* and *retract* operators of Prolog. In databases, $\mathcal{TR}$ is a declarative language for programming transactions, for updating database views, and for specifying active rules. In AI, $\mathcal{TR}$ can be used for representing procedural knowledge, planning, hypothetical reasoning, subjunctive queries and counterfactuals.

A number of implementations of $\mathcal{TR}$ exist [19, 22–24, 30, 35] but, unfortunately, all are logically incomplete. The major barrier to completeness for these implementations is similar to the reasons for Prolog incompleteness: the computation is based on an SLD-like resolution procedure with a depth-first goal selection strategy. This problem has been studied extensively in the logic programming literature [12, 32], and this led to the development of *tabling* (or *memoing*)—an efficient algorithm for logically complete implementation of logic programs based on SLD resolution [31, 34]. The best known implementation of tabling is XSB,[1] but there are others, such as Yap,[2] B-Prolog,[3] and Mercury.[4]

The success of the tabling solution in Prolog makes it a natural candidate for solving the analogous problems in Transaction Logic. The major difference in $\mathcal{TR}$ is that the latter deals with the phenomenon of *changing states*, which is not an issue in XSB and similar systems, where state changes are viewed as a non-logical feature that is best left outside of the scope of the tabling mechanism. In contrast, state updates have both *model-theoretic* and *procedural* semantics in Transaction Logic, and their correct treatment is essential.

In this paper, we extend tabling to Transaction Logic, but this is not the main result. The issue is that in $\mathcal{TR}$ tabling requires memoing of the underlying database state and not just memoing of the previously called subgoals. Clearly, this is a major problem both in terms of space and time. Of course, a powerful formalism such as Transaction Logic does not come without a price, but our contribution is in showing that there is ample room for optimization. After describing the extended tabling algorithm, we discuss the major trade-offs in its implementation and propose several time/space optimizations. We implemented a dozen of algorithms, which combine our optimizations in various ways. Here we discuss six of those that illustrate the most salient points. We discuss the rationale behind each of them, and then present our experimental results. These results show that a proper integration of our techniques results in a system with the best overall performance and scalability characteristics.

We are not aware of any work that directly deals with problems similar to ours. However, we are building on a host of results, which became ingredients in our optimization techniques or could be used for further optimization. These include the already mentioned works on tabling, the various indexing data structures, such as B^+ trees and other balanced trees (like AVL, Red-Black, and 23-trees), tries, sets, and others [14, 17, 21, 25, 26, 29].

* This work is part of the SILK (Semantic Inference on Large Knowledge) effort within Project Halo, sponsored by Vulcan Inc. The authors were also partially supported by the NSF grant 0964196.

PPDP'10, July 26–28, 2010, Hagenberg, Austria.

[1] `http://xsb.sourceforge.net`

[2] `http://www.dcc.fc.up.pt/~vsc/Yap`

[3] `http://www.probp.com`

[4] `http://www.cs.mu.oz.au/mercury`

This paper is organized as follows. In Section 2, we introduce the basics of Transaction Logic. Section 3 defines our tabling evaluation strategy for $\mathcal{TR}$. In Section 4, we discuss the main issues with practical realization of the tabling algorithm and describe our solutions to these problems. Section 5 describes our performance evaluation study. Section 6 summarizes this paper and outlines future work.

2. Transaction Logic

In this section we will briefly introduce the syntax and the proof theory of Transaction Logic as necessary for understanding the results of this paper. Although knowing the model-theoretic semantics of the logic is not strictly necessary for the purpose of this paper, we will still introduce it informally to give the reader a feel of what this logic is about. For further details on Transaction Logic and its extension, the reader is referred to [3, 8, 9].

The syntax of $\mathcal{TR}$ is very similar to the standard logic. The *atomic formulas* have the form: $p(t_1, ..., t_n)$, where p is a predicate symbol, and the t_i are terms (variables, constants, function terms). The only difference is that this logic introduces two *additional* logical connectives:

- $\otimes$ - the sequential conjunction
- $\diamond$ - the modal operator of hypothetical execution

and extends the other standard connectives, such as conjunction, disjunction, the quantifiers, etc. Informally, the formula $\phi \otimes \psi$ is an action composed of an execution of ϕ followed by an execution of ψ. The formula $\diamond\phi$ is an action of *hypothetically* testing whether ϕ can be executed at the current state, but no actual state change takes place. When ϕ and ψ are regular first-order formulas, $\phi \otimes \psi$ reduces to the usual first-order conjunction, $\phi \wedge \psi$, and $\diamond\phi$ to ϕ.

In this paper we are dealing with a small, but very powerful subset of Transaction Logic, which consists of *serial-Horn rules* and queries. This subset is interesting because it subsumes traditional logic programming and it suffices for many applications of $\mathcal{TR}$ in other areas (Semantic Web, trust, etc.). We will also drop the hypothetical operator $\diamond$, as it does not introduce new issues, and assume that all predicate symbols are partitioned into *fluents* (i.e., regular, static first-order predicates) and *actions*.[5]

A *serial-Horn rule* is a statement of the form

$$h \ :- \ b_1 \otimes b_2 \otimes \ldots \otimes b_n. \tag{1}$$

where h is an atomic formula and b_1, ..., b_n are *literals*. A *literal* is either an atomic formula or a *negated fluent* of the form $\texttt{not}\ f$ (where f is a fluent). Here not is some form of *default negation*, such as Clark's negation as failure [13], the well-founded [33], or stable-model [20] negation. The exact semantics of not does not matter here, because the semantics of the underlying database states is one of the parameters to Transaction Logic; the logic works with all of the above semantics and our results do not depend on a particular choice. It is easy to see that since, as noted above, $\otimes$ reduces to $\wedge$ for fluents, serial-Horn rules subsume normal logic programs with negation. The following simple example illustrates the above concepts. All our examples will be using the usual logic programming convention whereby lowercase symbols represent constants, function, and predicate symbols, and the uppercase symbols represent variables universally quantified outside of the rules. Also as usual, the universal quantifiers are omitted.

$$\begin{array}{lll} move(X,Y) & :- & on(X,Z) \otimes clear(X) \otimes clear(Y) \\ & & \otimes\, \texttt{not}\, tooHeavy(X) \\ & & \otimes delete(on(X,Z)) \\ & & \otimes insert(on(X,Y)) \\ & & \otimes delete(clear(Y)). \\ tooHeavy(X) & :- & weight(X,W) \otimes limit(L) \otimes W < L. \\ ?- & move(blk1, blk15) \otimes move(SomeBlk, blk1). & \end{array}$$

Assuming that *move*, *delete*, and *insert* represent actions and *on*, *clear*, *tooHeavy*, *weight* are fluents, this example illustrates several features of the serial-Horn subset of Transaction Logic. The first rule is a definition of a complex action of moving a block from one place to another. The second rule defines the fluent *tooHeavy*, which is used in the definition of *move* (under the scope of default negation). The second rule consists exclusively of fluents and thus is a regular logic programming rule.

The last statement above is an invocation of a *transaction*, which corresponds to a query in logic programming. It requests to move block *blk1* from its current position to the top of *blk15* and then find some other block and move it on top of *blk1*. What happens if, after placing *blk1* on top of *blk15* the second move fails? This can happen, for example, if all the available clear blocks are too heavy for our robot. Traditional logic programming does not assign a logical semantics to its update operators and in such a case the aforesaid action will remain partially executed: the effects of the first move will remain in the underlying database even though the second move failed. This may, for example, mean that the overall state of the system will become incoherent with the designer's intent. For instance, if the robot is supposed to build a pyramid of some sort, the programmer would have to go into trouble of *explicitly* undoing partially executed actions and trying another route. The lack of the logical semantics for update operators makes logic programs that use them in an essential way the hardest to write, debug, and understand.

In contrast, Transaction Logic gives such operators logical semantics of a transaction. In particular, action invocations are *atomic*. In the above case, if any part of a transaction fails, the effect is as if nothing was done at all. This, if all the remaining blocks are too heavy, the above transaction is not executed—*fails*—all actions are "backtracked over" and the underlying database state is not changed.

Informally, the truth of a transaction is evaluated over sequences of states—*execution paths*—which makes it possible to think of truth assignments in $\mathcal{TR}$'s models as executions. If a transaction, ψ, defined by a set of serial-Horn rules, $\mathbf{P}$, evaluates to true over a sequence of states $\mathbf{D_0}, \ldots \mathbf{D_n}$, we say that it can *execute* at state $\mathbf{D_0}$ by passing through the states $\mathbf{D_1}$, ..., $\mathbf{D_{n-1}}$, ending in the final state $\mathbf{D_n}$. Formally, this is captured by the notion of *executional entailment*, which is written as follows:

$$\mathbf{P}, \mathbf{D_0}, \ldots \mathbf{D_n} \models \psi$$

The following examples play dual role here. First, they give the reader a glimpse into the power of Transaction Logic. Second, they provide a good test-bed for performance evaluation, and we will utilize these examples in our performance study. All these examples are fairly involved, even if they may look simple.

EXAMPLE 1 (Consuming paths). *Suppose $edge$ is a binary fluent and $delete(edge(N, M))$ denotes the action of deleting the edge that goes from node N to node M. The following rules compute reachability in the graph by traversing edges and then swallowing*

[5] This partitioning is not required by $\mathcal{TR}$, but it simplifies explanations here.

them:

$$\begin{array}{l} reach(X,Y) \ :- \\ \qquad reach(X,Z) \otimes edge(Z,Y) \otimes delete(edge(Z,Y)). \\ reach(X,X). \end{array} \tag{2}$$

Note that the first rule defines the action reach *recursively.* □

EXAMPLE 2 (Hamiltonian cycle). *A Hamiltonian cycle is a cycle in a directed graph that visits each vertex exactly once. Similarly to consuming paths, Hamiltonian cycles are detected here by swallowing the already traversed vertexes.*

$$\begin{array}{l} hCycle(Start,Start) \ :- \ \texttt{not}\ vertex(X). \\ hCycle(Start,X) \ :- \\ \qquad edge(X,Y) \otimes vertex(Y) \\ \qquad \otimes delete(vertex(Y)) \otimes insert(mark(X,Y)) \\ \qquad \otimes hCycle(Start,Y) \otimes insert(vertex(Y)). \end{array} \tag{3}$$

This solution to Hamiltonian paths relies on the transactional semantics of $\mathcal{TR}$. The second rule does the search and there are many possible ways for it to fail. Due to the transactional semantics of the logic, changes to the database state made while expanding these failing derivation paths are "forgotten" and new derivations are then tried. □

Note that so far we have been describing our examples procedurally, in terms of search. The actual model-theoretic semantics has none of that. It simply says that (in Example 2) the transaction $hCycle(Start,Start)$ can execute, i.e., that there is an executional entailment of the form

$$\mathbf{P}, \mathbf{D}_0, \ldots \mathbf{D}_n \ \models \ hCycle(Start,Start)$$

where $\mathbf{D_0}$ is the original graph, if and only if there is a Hamiltonian cycle in the graph. The actual cycle can be extracted from the above sequence of states. While the model theory is completely declarative, the aforesaid search does take place: it is performed by the sound and complete proof theory of $\mathcal{TR}$, which appears later.

EXAMPLE 3 (Planning). *The following rules define a STRIPS-like planner for building pyramids of blocks. It uses the already familiar fluents, like* on *and* clear*, and the actions* move, insert *and* delete. *In addition, it defines the actions* pickup, putdown*, and the recursive action $stack$, which represents the pyramid building transaction.*

$$\begin{array}{l} stack(0,Block). \\ stack(N,X) \ :- \ N>0 \otimes move(Y,X) \otimes stack(N-1,Y) \\ \qquad \otimes on(Y,X). \\ stack(N,X) \ :- \ N>0 \otimes on(Y,X) \otimes unstack(Y) \\ \qquad \otimes stack(N,X). \\ unstack(X) \ :- \ on(Y,X) \otimes unstack(Y) \otimes unstack(X). \\ unstack(X) \ :- \ isclear(X) \wedge on(X,table). \\ unstack(X) \ :- \ (isclear(X) \wedge on(X,Y) \wedge Y \neq table) \\ \qquad \otimes move(X,table). \\ unstack(X) \ :- \ on(Y,X) \otimes unstack(Y) \otimes unstack(X). \\ move(X,Y) \ :- \ X \neq Y \otimes pickup(X) \otimes putdown(X,Y). \\ pickup(X) \ :- \ clear(X) \otimes on(X,Y) \\ \qquad \otimes delete(on(X,Y)) \otimes insert(clear(Y)). \\ putdown(X,Y) \ :- \ clear(Y) \otimes \texttt{not}\ on(X,Z1) \\ \qquad \otimes \texttt{not}\ on(Z2,X) \otimes delete(clear(Y)) \\ \qquad \otimes insert(on(X,Y)). \end{array} \tag{4}$$

The above rules represent a straightforward algorithm for building a pyramid. The first rule says that stacking zero blocks on top of X is a no-op. The second rule says that, for bigger pyramids, stacking N blocks on top of X involves moving some other block, Y, on X and then stacking $N-1$ blocks on Y. To make sure that the planner did not remove Y from X while building the pyramid on Y, we are verifying that $on(Y,X)$ continues to hold at the end. Looking down at the definition of move *we may notice that this action will not be performed if X is not clear. What then? The third rule for* stack *says that in that case the robot should* unstack *whatever is no X and make X clear. The* unstack *action is also recursively defined and is, in a sense, the opposite of stacking. Definition of the other actions is straightforward.* □

A proof theory for serial-Horn transaction logic. The proof theory for serial-Horn $\mathcal{TR}$, described in [3, 9], resembles the well-known SLD resolution proof strategy for Horn clauses, but it has additional inference rules and axioms. The theory aims to prove statements of the form $\mathbf{P}, \mathbf{D}_0$ --- $\vdash \psi$, which are called *sequents*. Here $\mathbf{P}$ is a set of serial-Horn rules and ϕ is a *serial-Horn goal*, i.e., a formula that has the form of a body of a serial-Horn rule, such as (1).

An inference succeeds if and only if it finds an execution for the transaction ψ—a sequence of database states $\mathbf{D}_1, \ldots, \mathbf{D}_n$—such that $\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models \psi$.

Axioms: $\mathbf{P}, \mathbf{D}$ --- $\vdash ()$

Inference Rules: In Rules 1–3 below, σ is a substitution, a and b are atomic formulas, and ϕ and $rest$ are serial goals.

1. *Applying transaction definitions*:
Suppose $a \leftarrow \phi$ is a rule in $\mathbf{P}$ whose variables have been renamed apart so that the rule shares no variables with $b \otimes rest$. If a and b unify with a most general unifier σ, then

$$\frac{\mathbf{P}, \mathbf{D} \text{ --- } \vdash (\exists)\, (\phi \otimes rest)\sigma}{\mathbf{P}, \mathbf{D} \text{ --- } \vdash (\exists)\, (b \otimes rest)}$$

2. *Querying the database*:
If b is a fluent literal, $b\sigma$ and $rest\,\sigma$ share no variables, and $b\sigma$ is true in the database state $\mathbf{D}$ then

$$\frac{\mathbf{P}, \mathbf{D} \text{ --- } \vdash (\exists)\, rest\ \sigma}{\mathbf{P}, \mathbf{D} \text{ --- } \vdash (\exists)\, (b \otimes rest)}$$

3. *Performing elementary updates*:
If $b\sigma$ and $rest\,\sigma$ share no variables, and $b\,\sigma$ is an elementary action that changes state $\mathbf{D}_1$ to state $\mathbf{D}_2$ then

$$\frac{\mathbf{P}, \mathbf{D}_2 \text{ --- } \vdash (\exists)\, rest\ \sigma}{\mathbf{P}, \mathbf{D}_1 \text{ --- } \vdash (\exists)\, (b \otimes rest)}$$

Given an inference system, an *executional deduction* (or *proof*) of a sequent, seq_n, is a series of sequents, $seq_1, seq_2, \ldots, seq_{n-1}, seq_n$, where each seq_i is either an axiom-sequent or is derived from earlier sequents by one of the above inference rules. If $\mathbf{D}_0$, $\mathbf{D}_1$, ..., $\mathbf{D}_{n-1}$, $\mathbf{D}_n$ are the database states of these sequents, respectively, then $\mathbf{D}_n, \mathbf{D}_{n-1}, \ldots, \mathbf{D}_1, \mathbf{D}_0$ is called the *execution path* of the deduction.

THEOREM 1 (Soundness and Completeness [3]). *If ϕ is a serial-Horn goal, the executional entailment*
$\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models (\exists)\, \phi$ *holds if and only if there is an executional deduction of $(\exists)\, \phi$ whose execution path is $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$.*

It is important to keep in mind that this completeness result does not prescribe any particular way of applying the inference rules. If these rules are applied in the forward direction, then all execution paths will be enumerated and completeness will be realized. However, such proofs are undirected, exhaustive, and impractical. In contrast, if we apply the rules backwards, then we obtain a strategy that generalizes the usual SLD resolution with left-to-right literal selection—exactly the strategy used in Prolog. This strategy provides an efficient, goal-directed search strategy for proofs, but it

is, unfortunately, incomplete. In many cases, recursive (especially left-recursive) rules cause SLD resolution with left-to-right literal selection to get stuck in an infinite depth-first search of the proof tree.

Coming back to our earlier examples, recall that all of them have recursive definitions of various transactions. Just as in Prolog, it is not hard to see that the SLD strategy for the above proof theory will get stuck in infinite derivation paths. This is especially clear in case of Examples 1 and 3. Just as in ordinary logic programming, to make the above proof theory complete for an SLD-style strategy, it is necessary for the first rule (the one that most resembles SLD resolution) to be applied in a breadth-first manner, but this is hard to implement efficiently. As mentioned in the introduction, the most widely accepted solution to this problem in logic programming is *tabling* [31, 34], which essentially achieves an effect similar to breadth-first search. The next section introduces a tabling strategy for the $\mathcal{TR}$ proof theory shown above.

3. Tabling for Transaction Logic

Tabling for $\mathcal{TR}$ is analogous to tabling for Datalog, but with one major difference: not only the goals that are yet to be proved need to be memoized, but also the database states in which the calls to those goals were made. Likewise, not only the answers to these goals must be memoized, but also the states that get created by execution of those goals. We first describe the main principles of the algorithm and then incorporate it into the proof theory of Section 2 by modifying the first inference rule.

The main idea in tabled logic programming is to re-use answers that were computed for previous calls to the same goal. First, predicates of the program are partitioned into the *tabled* ones and those that are *not* tabled. In principle, all predicates could be tabled and query execution would still be correct. However, in some cases, knowing that some predicates do not have to be tabled (while still preserving completeness) can lead to significant savings (Sections 4, 5). One tabled goal is said to *dominate* another in tabled resolution if the two goals are *variants* of each other (variant tabling), i.e., are identical up to variable renaming, or if the first goal *subsumes* the second (subsumptive tabling). When a subgoal to a tabled predicate starting in a particular state is encountered, a check is made to see whether this is the first occurrence of this subgoal in that state (i.e., no dominating goal call was made before in the same state).

- If the call is new, the pair $(goal, state)$ is saved in a global data structure called the *table space*, and evaluation uses normal clause resolution to compute answers and generate new database states for the subgoal. The computed $(\textit{answer-unification}, \textit{new-state})$ pairs are recorded in the *aswer table* created for the aforesaid $(goal, state)$ pair each time they are computed.
- If the call is *not* new, i.e., a pair $(goal, state)$ exists in the table space for a dominating goal, the answers to the call are returned directly from the answer table for $(goal, state)$ and no clause resolution is used.

The evaluation goes on by returning new answers to subgoals until all answers for all goals generated during this process are computed.

Modified Inference Rule 1: As in the previous section, σ denotes substitutions, a and b atomic formulas, and ϕ, $rest$ are serial-Horn goals.

1a. *Applying transaction definitions for tabled predicates:*
Suppose $b \otimes rest$ is a goal for a program $\mathbf{P}$ where b is a call to a tabled predicate encountered for the first time at state $\mathbf{D}$ (i.e., no dominating pair $(c, \mathbf{D})$ is in the table space). Let $a \leftarrow \phi$ be a rule in $\mathbf{P}$ whose variables have been renamed apart from $b \otimes rest$ (i.e., the rule shares no variables with the goal) and suppose that a and b unify with the mgu σ. Then:

$$\frac{\mathbf{P}, \mathbf{D} \text{ --- } \vdash (\exists)\, (\phi \otimes rest)\sigma}{\mathbf{P}, \mathbf{D} \text{ --- } \vdash (\exists)\, (b \otimes rest)}$$

and the pair $(b, \mathbf{D})$ is added to the table space. When the sequent $\mathbf{P}, \mathbf{D}' \text{ --- } \vdash (\exists)\, (rest)\gamma$, for some substitution γ, is derived, the answer $(b\gamma, \mathbf{D}')$ is added to the answer table associated with the table entry $(b, \mathbf{D})$.

1b. *Returning answers from answer tables:*
Suppose $b \otimes rest$ is a goal call to program $\mathbf{P}$ at state $\mathbf{D}$, b's predicate symbol is declared as tabled, and let there be a dominating pair $(c, \mathbf{D})$ in the table space. Suppose that the answer table for $(c, \mathbf{D})$ has an entry $(a, \mathbf{D}')$ and a and b unify with mgu σ. Then:

$$\frac{\mathbf{P}, \mathbf{D}' \text{ --- } \vdash (\exists)\, (rest)\sigma}{\mathbf{P}, \mathbf{D} \text{ --- } \vdash (\exists)\, (b \otimes rest)}$$

1c. *Applying transaction definitions for non-tabled predicates:*
This rule is identical to Rule 1 in the proof theory of Section 2: Let $a \leftarrow \phi$ be a rule in $\mathbf{P}$ and a's predicate symbol is *not* tabled. Assume that this rule's variables have been renamed apart from $b \otimes rest$ and that a and b unify with mgu σ. Then

$$\frac{\mathbf{P}, \mathbf{D} \text{ --- } \vdash (\exists)\, (\phi \otimes rest)\sigma}{\mathbf{P}, \mathbf{D} \text{ --- } \vdash (\exists)\, (b \otimes rest)}$$

The rest of the tabling proof theory for Transaction Logic (Rules 2 and 3) is identical to the original theory of Section 2.

The rules 1a–1c modify the original proof theory for $\mathcal{TR}$ by capturing the effects of tabling. Rule 1a creates new entries in the table space and their associated answer tables. When a call to a subgoal is complete, the corresponding answer (both the substitution and the resulting database state) are added to an appropriate answer table. Rule 1b deals with calls for which dominating table entries already exist. In those cases, no clause resolution is used and answers are returned directly from the appropriate answer tables. Rule 1c is identical to Rule 1 of the original proof theory for Transaction Logic, but here it is applied only to non-tabled predicates. It simply does clause resolution SLD-style. Notice that Rule 1b might change the current database state after returning an answer for b, since the returned answer might have been obtained as a result of execution of state-changing actions.

We modify the definition of deduction in Section 2 to accommodate tabling: A *tabled* deduction for $\mathbf{P}, \mathbf{D} \text{ ---} \vdash (\exists)\, \phi$, is a series of sequents, where each sequent is an axiom or is derived from earlier sequents by an inference rule of the above tabling inference system.

THEOREM 2 (Soundness and Completeness). *Suppose ϕ is a serial-Horn goal.*
Soundness*: if there is a tabled deduction of the sequent $\mathbf{P}, \mathbf{D}_0 \text{ ---} \vdash (\exists)\, \phi$ with the execution path $\mathbf{D}_0, \mathbf{D}_1, ..., \mathbf{D}_n$ then the executional entailment $\mathbf{P}, \mathbf{D_0}, \mathbf{D_1}, \text{ ---}, \mathbf{D_n} \models (\exists)\, \phi$ holds.*
Completeness*: If the executional entailment $\mathbf{P}, \mathbf{D_0}, \mathbf{D_1}, \text{ ---}, \mathbf{D_n} \models (\exists)\, \phi$ holds then there is a tabled deduction of the sequent $\mathbf{P}, \mathbf{D}_0 \text{ ---} \vdash (\exists)\, \phi$ with the execution path that starts in state $\mathbf{D}_0$ and ends in $\mathbf{D}_n$.*

This theorem is different from the completeness of the proof theory in Section 2 (Theorem 1) in that it does not guarantee that all execution paths will be found: it only guarantees that all final states will be found. This is a very essential difference because the number of all execution paths can be infinite (even in simple cases

where function symbols are not involved), while the number of final states is often finite. The user typically is interested in finding out whether a particular transaction can execute starting at a particular state and finish in a particular final state (or a group of states). The fact that there are additional executions where the same subsequence sequence of states repeats itself (which is the main cause of infiniteness) is usually of no interest to the user. This leads to the following important termination result.

THEOREM 3 (Termination). *Suppose that ϕ is a serial-Horn goal and assume that all proofs of $\mathbf{P}, \mathbf{D}$ --- $\models (\exists)\,\phi$ using the proof theory of Section 2 mention only a finite number of database states and a finite number of goals. Let us further assume that all recursive predicates in $\mathbf{P}$ are marked as tabled. Then the* tabling *proof theory finds one or more proofs of $\mathbf{P}, \mathbf{D}$ --- $\models (\exists)\,\phi$ and* terminates. *Moreover, if there exists a proof of the executional entailment $\mathbf{P}, \mathbf{D}, ..., \mathbf{D}' \models (\exists)\,\phi$ using the proof theory of Section 2 then there is a tabled deduction that finds an execution of $(\exists)\,\phi$ that starts at $\mathbf{D}$ and ends at $\mathbf{D}'$.*

Note that the above theorem does not guarantee that all executions found by the original proof theory of [3] will also be found by the tabling proof theory, and this is a good thing! In this way, the new proof theory will find all the executions *that matter*, and will be able to terminate.

4. Problems and Solutions

In this section we discuss the major hurdles on the way to implementing the algorithm of Section 3 and propose a number of solutions. Then we describe six different implementations that progressively adopt these solutions. A performance evaluation of these implementations is described in Section 5.

4.1 Main Difficulties with Tabling Transaction Logic

The first obvious problem in implementing Transaction Logic is the transactional semantics of its actions, which requires atomicity. As it turned out, this is the easiest problem to address, and all existing implementations support atomicity.

The hard implementational issues have to do with tabling. These issues stem from the same major difficulty, which is easy to spot after a quick review of the tabling algorithm: unlike normal logic programming, tabling in $\mathcal{TR}$ implies saving the underlying database states as part of the answer tables. This raises the following problems:

1. *Space*. Saving database states in answer tables potentially leads to huge duplication of information. This is particularly troublesome for large database states (e.g., tens of thousands or even millions of facts).
2. *Time*. Tabling database states implies the following time-consuming operations:
   (a) *Copying of states*. Since states are modified in the course of transaction execution, tabled states must be copied, since once tabled the contents of that state must stay immutable.
   (b) *Comparison of states*. When a transactional subgoal is invoked at a state, we must check whether that particular goal/state pair is already tabled. This involves comparison of states as sets. In the worst case, comparing two states might take $O(n \cdot log(n))$ time, where n is the size of the states. Worse, newly created states might have to be compared with other tabled states to determine if the newly created set of facts is a genuinely new state or has been seen before.
   (c) *Querying of states*. The states created during the execution of transactions must be efficiently queryable. We will soon see, however, that there is a tension between the efficiency of querying and the various solutions to the aforementioned problems with time and space.

Each of the above problems has a number of solutions, but the different solutions involve various trade-offs, so it is not obvious how the different solutions fare when combined. The next section discusses ideas that lead to substantial savings in various situations.

4.2 The Space of Possible Solutions

We will now map the space of possible approaches to the problems listed in the previous section and discuss the various trade-offs in adopting the different space- and time-saving solutions. In Section 4.3, we discuss the most interesting combinations of these solutions, and their performance is evaluated in Section 5.

Space issues. Our first observation is that although the initial state of a transaction might be huge, a typical transaction changes only a few dozens of facts. Transactions that originate in AI or graph algorithms, as in our Examples 1, 2, and 3, might modify hundreds or even thousands of facts, but this is still far cry from millions or even billions of facts that an initial state might contain. This suggests an obvious idea: *differential logs*. That is, instead of tabling an entire state, we can represent a state as a pair of the form $(initial\_state, changelog)$. This representation not only saves space, but also reduces the amount of time required for copying states. A differential log is normally represented as a pair of logs $(InsertLog, DeleteLog)$. The former contains the records of inserted facts and the latter of deleted ones. Differential logs introduce a trade-off between the decreasing cost of storing and copying states and the increasing time for querying database states. Depending on the data structures used for changelogs, this overhead could be a constant factor of 2 or more.

The next possibility is to employ the various forms of *compression*, such as:

- *Sharing*. Logs can be stored using data structures, like *tries*, which enable high degree of sharing, so the total space requirement would be less than the sum of the sizes of all the logs.
- *Factoring*. Database facts can be stored on a heap and shared among states. The states themselves can refer to these facts using pointers or a numbering scheme. Thus, duplication of facts that are common to many different states is much less costly (only one word per duplicate fact).
- *Table skipping*. It might be possible to reduce the number of states that need to be tabled by carefully analyzing the rules and determining that only the states associated with certain subgoals have to be tabled. Other states can be modified directly without the need for storing or copying them. The theoretical basis for skipping is Theorem 3. All that is needed is to ensure that enough predicates are tabled to affect termination. The theorem states that it suffices to table just the recursive predicates, but in some cases even that much might be unnecessary.
- *Double-differential logs*. When table-skipping is used, the changes made by the transaction *with respect to the previous tabled state* can be kept in a log and not merged to that state until the next tabled state is reached. In this case, the current state is represented as a pair $(tabled\_state, changelog\_relative\_to\_tabled\_state)$. In turn, the tabled state is represented as a pair $(initial\_state, main\_changelog)$, so the entire state is represented using the initial state and two relative change logs. The first of these logs is called the *main* change log and the sec-

ond is the *residual* change log. We call this state representation strategy *double-differential logging*.

Time issues. Two of the main time-related issues are copying and comparing of states. The third issue, which stems from the suggestion to use differential logs, has to do with the increased cost of querying the underlying database states.

- *State comparison.* Our first observation is that, in most cases, the newly created states are different from most of the already seen tabled stated. So, we need a fast method to rule out most of the equalities. One such method is based on *incremental hash functions*. Simple incremental hash functions are cardinality, the total number of arities of the facts in the states, or, for example, any function of the form $\bar{h}(State) = \Sigma_{e \in State}\, h(e)$ **mod** N. For better results, one can employ several such functions. (These hash functions must be incremental so they could be computed quickly over large sets.)

  If the hash functions fail to differentiate among the new state and some of the tabled states, the sets must be compared directly. This can be made faster if *replicas* of tabled states are kept sorted, since comparing two sorted lists is linear in the size of the lists.

  Still, this is not completely satisfactory if, for example, a newly created state has to be directly compared with multiple tabled states, which the hash functions failed to tell apart. It turns out, however, that the problem can be avoided by the use of data structures, such as *tries*. For instance, we can store the already seen tabled states as sorted lists in a trie. Then, comparison of any new state with *all* the stored states can be done in time linear *in the size of the new state* and will not depend on the number of the already seen states. We can further combine hashing with trie comparison by representing each tabled state as a list whose prefix (say, the first three elements, if we choose to use three hash functions) are the hash values of our hash functions and the rest is a sorted list of the actual facts that belong to the state. Thus, in searching the trie the first few comparisons will be made based on the hash values and then states will be discriminated based on the actual facts they contain.

- *Separate state repository*. Tabled states and goal calls are typically kept in tries, because this data structure enables fast (linear time) checks to find out whether a pair $(call, state)$ had been seen before. The question is then whether these pairs are stored in one trie (say, as a term $pair(call, state)$) or the calls are stored in one trie and states in another (the latter is called a *state repository*). Storing states and calls in the same trie typically requires more space (because calls and states tend to share less structure in such tries) and time (since call-state comparisons tend to fail later than in the case of separate state repositories).

- *Querying of states.* The data structures used for differential logs make a big difference for the querying time of the states, since in order to find out whether a particular fact is in a state one has to query both the initial state and the log. For double-differential logs, the overhead is even higher, since two logs must be queried in addition to the initial state. Since the initial state is static, it can be designed in the most advantageous way as far as querying is concerned. For the logs, we have to balance the insertion time against the query time.

  Unordered lists are best as far as the update time goes, but they are some of the worst for querying. Also, for state comparison we need *sorted* list, which makes unsorted lists less attractive. Tries are good for querying and updating, but they are poor at maintaining the sorted order among the facts. For state comparison, tries must be converted to lists at the cost of $n \cdot log(n) \cdot |S|$, where $|S|$ is the number of facts in the trie.

  Nevertheless, in our experiments, we stored some logs as tries, since they are the most optimized data structure in our underlying platform, XSB. To compensate for the tries' inefficiency in keeping the logs sorted, we sometimes maintained sorted lists as auxiliary data structures. A much better choice would have been B^+ trees, as they can be made shallow (thus improving the search) and they naturally keep data sorted.

- *Copying of states*. First, note that the table skipping and factoring methods that were introduced as space-saving techniques are also important time-saving techniques, because the fewer states are tabled—the fewer state-comparisons and copying are needed. Double-differential logs can also reduce the number of times states have to be copied. This happens because in double-differential logging, new tabled states are created by merging the previous tabled state with the residual log. This is done just before entering the *next* tabled state. In contrast, in single-differential logging, states that might get tabled at the next opportunity are initially created by copying the state that was tabled just now. The copy is then modified directly and it gets saved in the state repository when the next tabled call is made. If no new tabled call is reached, the copy has been made in vain. Double-differential logging delays copying of states and thus is less prone to wasteful copying.

  Beyond that, the fastest data structure to copy would be a list. In fact, if logs are represented as *unsorted* lists then no copying would be needed whatsoever. Logs can simply be passed as arguments to the predicates that represent actions. For instance, the log at state k could be $(InsertLog_k, DeleteLog_k)$ and the next state (say, after inserting p) it would be $([p|InsertLog_k], DeleteLog_k)$, which shares the lists $InsertLog_k$ and $DeleteLog_k$ with the previous state.

  Unfortunately, as discussed earlier, lists are not efficient for querying, and we need them sorted. In our performance evaluation, we compared list-based implementations with others to validate the trade-off between copying and querying of states. With an eye on querying, balanced trees are reasonably efficient to copy, since their space overhead is a constant factor (compared to lists). In our comparisons, however, red-black trees and AVL trees did worse than tries because tries are highly optimized in XSB. However, an optimized implementation of B^+ trees would be far superior than tries. The space overhead factor for B^+ trees is only $1 + 1/(k-1)$, where k is the degree of the tree, and they can be copied very efficiently, if implemented in a low-level language like C.

  Thus, a trade-off exists between the costs of querying and copying, which we evaluate in our performance study.

4.3 Implementations

Overall, we implemented more than a dozen of different algorithms, which realize various combination of the above ideas. In this section, we discuss six of the most interesting such implementations.

Common features. All implementations discussed here share the following common features, which were introduced earlier in this section:

- Data compression via factoring.
- Differential logs.
- State comparison:
  - via incremental hash functions—to quickly rule out most false matches

- state repositories that use tries to store replicas of the main differential logs—to ensure at most linear-time match of newly created states against all previously seen states

These basic optimizations speed up our tests by up to three orders of magnitude and use about two orders of magnitude less memory, but their effects are even greater on larger problems. Beyond that, the different implementations employ other optimizations as follows[6].

Implementation 1. This implementation uses the above common features in which differential logs are single, since table-skipping is *not* used. The logs are maintained as *ordered* lists stored in the state repository. New states are constructed via insertion-sort operations. As noted earlier, lists are a poor choice for querying states, but they are near-optimal for copying. Recall that a differential log has the form $(InsertLog, DeleteLog)$. Moving to the next state is accomplished by inserting a record in the insertion or deletion logs. In the worst case, this is linear in the log size, but the average is under 3/4 of the log size. Since successive states often share their list tails, this can also result in space savings.

Implementation 2. This implementation is similar to #1, but logs are stored both as ordered lists and tries. The ordered lists reside in the state repository, as before, and tries are used to speed up querying. When moving from state to state, the tries are modified directly, without copying, so the only significant overhead here is the need to maintain a query trie. To improve performance, creation of the query trie can be delayed until the first query or update.

Implementations 3a and 3b. These implementations use table skipping to reduce the number of tabled states. Table-skipping avoids state comparison and copying when executing non-tabled (usually non-recursive) actions. State copying is still required at tabled states. Furthermore, since states produced by non-tabled transactions are not saved in tables, there is no need to check if we have seen such states before. Both implementations use sorted lists to represent logs. However, 3a uses single differential log and 3b uses double logs.

Implementations 4a and 4b. Like 3a and 3b, these implementations use table skipping, where 4a uses single differential logs and 4b uses double logs. The difference is that 4a represents its single log as a trie and 4b does the same for its main differential log. The residual differential log in 4b is still maintained as a sorted list. (In our tests, the residual differential logs were generally short, which did not justify the overhead of using tries for them.) Similar to the implementation 2, the creation of the main differential log (e.g., for the implementations 4b) is delayed until such data structure is needed.

5. Performance evaluation

The above implementations were tested on a workstation with Pentium dual-core 2.4GHz CPU and 3GB memory running on Ubuntu Linux and XSB Prolog version 3.2.

In describing the results of our tests, we use tables that show time (in seconds) and space (in Kb) costs for the different implementations using the problems from Section 2 of gradually increasing size. To increase accuracy, we make the tests run for considerable amounts of time and avoid the possibility where different algorithms might pick up solutions that incur different costs. To this end, our tests compute *all* possible solutions for every problem in our suite and the numbers of solutions for each case are listed in the tables.

One of the important goals of this performance study is to demonstrate the benefits of table-skipping and double-differential logging. To show this, we include tables that display the numbers of tabled (saved) states, the numbers of times states were copied, and the numbers of times new states were compared with the contents of the state repositories (table-skipping implementations should do fewer of these operations). These tables also help us explain the reported times and assess the various trade-offs.

The overall conclusion from the study is that table-skipping and double-differential logging incur relatively small overheads for small problems, but bring substantial savings for larger problems and make them scale better. Likewise, maintaining data structures, like tries, that speed up querying of states brings significant speedups. The main overhead of those of our implementations that rely on tries (implementations 2, 4a, and 4b) is that copying tries is slow (7 times slower than copying lists in XSB). Since XSB's tries do not preserve the order on their contents, we had to also keep states as sorted lists—both time and space overhead. The use of B^+ trees in lieu of tries would have solved both of these problems, if an efficiently integrated version existed for XSB.

The suite of the different implementations of $\mathcal{TR}$ and of the test cases used in this comparison is provided at `http://flora.sourceforge.net/tr-interpreter-suite.tar.gz`.

Consuming paths

Table 1 shows execution times and memory consumption for the consuming paths problem for graphs with 100, 250, and 350 vertices. The row *# of Solutions* also shows the total number of solutions found.

Graph size	100		250		350	
# of Solutions	5050		31375		61425	
	CPU	Mem.	CPU	Mem.	CPU	Mem.
1	0.128	806	1.544	4843	3.940	9473
2	0.212	5538	2.292	66413	5.996	173389
3a	0.136	807	1.540	4843	3.924	9473
3b	0.152	806	1.672	4843	4.608	9473
4a	0.224	10325	2.796	128434	7.880	337070
4b	0.204	5538	2.128	66413	5.680	172976

Table 1. Times for finding consuming paths in graphs

It might seem surprising that Implementation 1, which incorporates only the basic optimizations, is one of the two best performers. Implementation 3, which adds table skipping, does only infinitesimally better. The explanation for this behavior is provided by Table 2: The nature of the consuming paths problem is such that all states must be tabled, so there is no advantage to table-skipping. Indeed, Table 2 shows that the number of tabled states and state comparisons is exactly the same for all implementations and depends only on the problem size. Using efficient data structures for logs, such as tries, does not help either. Only a relatively small number of queries is issued, and the benefits of faster querying using tries are negated by the overhead of copying tries compared to lists (earlier we mentioned that copying a trie takes 7 times longer). Tries also take more space than lists and, since the number of tabled states is the same for all implementations, the ones that maintain the logs using tries require significantly more space.

[6] http://flora.sourceforge.net/tr-interpreter-suite.tar.gz

Graph size	100		250		350	
	States	Comp.	States	Comp.	States	Comp.
1	5051	5050	31376	31375	61426	61425
2	5051	5050	31376	31375	61426	61425
3a	5051	5050	31376	31375	61426	61425
3b	5051	5050	31376	31375	61426	61425
4a	5051	5050	31376	31375	61426	61425
4b	5051	5050	31376	31375	61426	61425

Table 2. Numbers of tabled states and state comparisons for finding consuming paths in graphs

Nevertheless, it is easy to demonstrate that even for the consuming paths problem the use of table-skipping, tries, and double-differential logging is greatly beneficial. To see this, we can use the consuming paths method to find ten paths simultaneously in ten disjoint graphs. Our solution to this problem was obtained from the original consuming paths problem by simply repeating the "consuming" part of the rules in (2) ten times on different *edge* predicates.

$$\begin{aligned} reach(X,Y) \ :- \\ & reach(X,Z) \\ & \otimes edge1(Z,Y) \otimes delete(edge1(Z,Y)) \\ & \otimes edge2(Z,Y) \otimes delete(edge2(Z,Y)) \\ & ... \\ & \otimes edge10(Z,Y) \otimes delete(edge10(Z,Y)). \\ reach(X,X). \end{aligned} \tag{5}$$

Table 3 clearly shows that table-skipping, tries, and differential logging bring substantial time benefits, as Implementation 4a, which incorporates all of these optimizations is by far the best.

Graph size	100		200		250	
	CPU	Mem.	CPU	Mem.	CPU	Mem.
1	6.236	4580	47.642	18219	92.425	28881
2	8.568	371762	M.Err.	M.Err.	M.Err.	M.Err.
3a	4.796	2533	37.182	10066	71.840	15620
3b	4.024	2533	30.073	10065	58.083	15620
4a	1.780	77873	13.536	596734	25.929	1155434
4b	1.292	39744	8.564	301429	16.325	582398

Table 3. Time and space for building 10 consuming paths in 10 graphs

Similarly to the ordinary consuming paths example, the explanation is provided by Table 4: the number of tabled states and state comparisons performed by the table-skipping implementations 3 – 4b is ten times less than the corresponding numbers for implementations 1 and 2. We also see that table-skipping is better memory-wise, since implementations 3a and 3b consume half of the memory used by Implementation 1. Implementations 4a and 4b are much more memory hungry compared to implementations 3a and 3b because of the use of query tries, which consume much more memory than lists.

Graph size	100		200		250	
	States	Comp.	States	Comp.	States	Comp.
1	50501	50500	201001	201000	313751	313750
2	50501	50500	201001	201000	313751	313750
3a	5051	5050	20101	20100	31376	31375
3b	5051	5050	20101	20100	31376	31375
4a	5051	5050	20101	20100	31376	31375
4b	5051	5050	20101	20100	31376	31375

Table 4. Numbers of tabled states and state comparisons for building 10 consuming paths in 10 graphs

Hamiltonian cycles

Our next experiment computes all Hamiltonian cycles in graphs of sizes 50, 150, and 200 nodes. The results are shown in Table 5. This table provides several interesting observations:

- In constructing Hamiltonian cycles, many more queries are issued than in the case of consuming paths, so efficient data structures for querying are important. Thus, Implementation 2 is much faster than Implementation 1.
- Table 6 shows that using table-skipping reduces the number of tabled states and state comparisons by about 1/3. This is not high enough to offset the benefits of fast querying, so Implementation 1 is still slightly better that Implementations 3a and 3b.
- The querying overhead of double-differential logging is quite noticeable in this case, so the times for implementations 3b and 4b are higher than for implementations 2 and 3a. Nevertheless, Implementation 4b beats 3b (both use double differential logs) because it uses query tries rather than lists.
- The number of state comparisons performed by versions 3a and 4a is higher than in case of 3b and 4b. This validates our earlier observation that, since double-differential logging defers state copying and comparison (unlike single-differential logs), this might lead to fewer of such comparisons and copies being done overall. This problem is partially responsible for the higher runtime of Implementation 4a, which makes a larger number of expensive trie copies and comparisons. The other reason is that the query tries need to be transformed into sorted lists at state comparison.

Graph size	50		150		200	
# of Solutions	50		150		200	
	CPU	Mem.	CPU	Mem.	CPU	Mem.
1	0.252	2412	8.392	51543	23.405	118248
2	0.244	6111	4.144	132082	9.148	303932
3a	0.164	2362	3.956	51091	10.100	118566
3b	0.236	7337	5.644	187927	13.968	442537
4a	0.300	15284	6.852	330211	16.105	755352
4b	0.300	15446	5.696	379042	12.584	885453

Table 5. Times for finding Hamiltonian cycles in graphs

Graph size	50		150	
	States	Comp.	States	Comp.
1	7403	7500	67203	67500
2	7403	7500	67203	67500
3a	4903	5051	44703	45151
3b	4903	5000	44703	45000
4a	4903	5051	44703	45151
4b	4903	5000	44703	45000

Table 6. Tabled states and state comparisons for finding Hamiltonian cycles

As with consuming paths, it is easy to demonstrate that, for larger examples, the combination of table-skipping, query tries, and double-differential logging, i.e., Implementation 4b, scales better and is the overall winning combination. To this end, we can use the problem of constructing ten Hamiltonian cycles in ten disjoint graphs analogously to the way the ten simultaneous consuming paths problem was constructed in (5). Table 7 shows the results, which do not require further elaboration.

Graph size	50		150	
	CPU	Mem.	CPU	Mem.
1	4.912	164777	M.Err.	M.Err.
2	6.052	424113	M.Err.	M.Err.
3a	3.076	9878	86.505	255174
3b	4.340	14854	105.814	391963
4a	1.656	58959	M.Err.	M.Err.
4b	1.356	46072	27.4210	1228925

Table 7. Times for finding 10 Hamiltonian cycles in 10 graphs

Blocks World

We conclude our performance study with the blocks world planning example for pyramids of 5, 6, and 7 blocks. Since the number of plans grows exponentially, we could not evaluate larger problems on our test machine. Our results are shown in Table 8.

Since our largest problem uses only seven active blocks, the main differential logs and, especially, the residual logs tend to be quite small. As a result, there is no significant benefit in using query tries. Similarly, although Table 9 indicates that table skipping reduces the number of comparisons by the factor of 3, the overhead of creating and comparing all those extra states in implementations 1 and 2 is not high. On the other hand, implementations 3b and 4b suffer slightly due to the higher querying overhead associated with double-differential logging.

Interestingly, Table 9 again shows the higher number of state comparisons (and therefore state copies) performed by single-differential logging implementations 3a and 4a. In case of 4a, this leads to a significant overhead because copying and comparing tries takes 7 times more time than in case of lists. Since Implementation 3a uses lists (and these lists are short) this implementation is not seriously affected by all the extra copying.

Blocks	5		6		7	
# of Pyramids	120		720		5050	
	CPU	Mem.	CPU	Mem.	CPU	Mem.
1	0.212	576	2.392	5586	29.265	63207
2	0.196	656	2.100	6197	26.265	68636
3a	0.196	546	2.192	5286	27.905	60105
3b	0.228	544	2.528	5284	31.661	60102
4a	0.288	3296	3.268	46269	41.958	1005012
4b	0.204	608	2.268	5793	28.117	64915

Table 8. Time and space requirements for building pyramids of N blocks in blocks worlds

Blocks	5		6		7	
	States	Comp.	States	Comp.	States	Comp.
1	1546	4210	13327	42792	130922	480326
2	1546	4210	13327	42792	130922	480326
3a	501	9767	4051	107882	37633	1364911
3b	501	1300	4051	13020	37633	144354
4a	501	9767	4051	107882	37633	1364911
4b	501	1300	4051	13020	37633	144354

Table 9. Numbers of tabled states and state comparisons for building pyramids in blocks worlds

Once again, transforming our planning problem into one in which ten separate pyramids are being built in ten parallel worlds clearly shows the advantage of our optimizations. The performance figures in Table 10 point to Implementation 4b as a clear winner.

Blocks	5		6		7	
	CPU	Mem.	CPU	Mem.	CPU	Mem.
1	1.800	9696	21.457	72741	286.413	128150
2	1.780	29289	19.441	233285	M.Err.	M.Err.
3a	1.140	889	13.208	7346	172.838	55984
3b	1.808	892	21.433	7349	287.413	75930
4a	1.312	30988	15.588	409155	M.Err.	M.Err.
4b	1.096	1614	11.984	12854	148.109	128150

Table 10. Time and space requirements for building pyramids of N blocks in 10 parallel blocks worlds

6. Conclusions and Future Work

In this paper we adapted the commonly used tabling technique [31, 34] from ordinary logic programs to Transaction Logic, a purely declarative extension of classical logic for defining state-changing transactions. We have shown that the proof theory of Transaction Logic modified with tabling is sound and, for all practical purposes, complete. We discussed a host of difficulties in implementing tabling for Transaction Logic and proposed a number of important optimizations for dealing both with time and space explosion. We implemented an interpreter that can combine several different optimizations as plugins, which enabled us to compare the different optimizations. In particular, we have shown the results of a performance study for six different implementations, which validate our intuitions about the value of the different optimizations.

For future work, we plan to further validate our results by incorporating an efficient implementation of B^+ trees. We are also planning to extend our work to Concurrent Transaction Logic [8, 30], a logic that extends $\mathcal{TR}$ with concurrent, interleaved actions, which presents additional challenges.

References

[1] Darko Anicic, Paul Fodor, Roland Stuhmer, and Nenad Stojanovic. An approach for data-driven logic-based complex event processing. In *The 3rd ACM International Conference on Distributed Event-Based Systems (DEBS)*, 2009.

[2] Moritz Y. Becker and Sebastian Nanz. A logic for state-modifying authorization policies. In *ESORICS*, 2007.

[3] A.J. Bonner and M. Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical Report CSRI-323, University of Toronto, November 1995. `http://www.cs.toronto.edu/~bonner/transaction-logic.html`.

[4] Anthony J. Bonner. Modular composition of transaction programs with deductive databases. In *DBPL*, pages 373–395, 1997.

[5] Anthony J. Bonner and Michael Kifer. Transaction logic programming. In *ICLP*, pages 257–279, 1993.

[6] Anthony J. Bonner and Michael Kifer. Applications of transaction logic to knowledge representation. In *ICTL*, pages 67–81, 1994.

[7] Anthony J. Bonner and Michael Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133 (2):205–265, 1994.

[8] Anthony J. Bonner and Michael Kifer. Concurrency and communication in transaction logic. In *JICSLP*, pages 142–156, 1996.

[9] Anthony J. Bonner and Michael Kifer. A logic for programming database transactions. In *Logics for Databases and Information Systems*, pages 117–166, 1998.

[10] Anthony J. Bonner and Michael Kifer. Results on reasoning about updates in transaction logic. In *Transactions and Change in Logic Databases*, pages 166–196, 1998.

[11] Anthony J. Bonner and Michael Kifer. The state of change: A survey. In *Transactions and Change in Logic Databases*, pages 1–36, 1998.

[12] Weidong Chen and David Scott Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43 (1):20–74, 1996.

[13] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 292–322. Plenum Press, 1978.

[14] Douglas Comer. Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[15] Hasan Davulcu, Michael Kifer, C. R. Ramakrishnan, and I. V. Ramakrishnan. Logic based modeling and analysis of workflows. In *PODS*, pages 25–33, 1998.

[16] Hasan Davulcu, Michael Kifer, and I. V. Ramakrishnan. Ctr-s: a logic for specifying contracts in semantic web services. In *WWW*, pages 144–153, 2004.

[17] A. Dovier, A. Policriti, , and G. Rossi. Integrating lists, multisets, and sets in a logic programming framework. *Applied Logic*, 3:213–229, 1996.

[18] Paul Fodor. Initial results on justification for the tabled transaction logic. In *AAAI Spring Symposium*, 2009.

[19] Amalia F.Sleghel. An optimizing interpreter for conccurent transaction logic. Master's thesis, University of Toronto, 2000.

[20] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Logic Programming: Proceedings of the Fifth Conference and Symposium*, pages 1070–1080, 1988.

[21] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *SFCS '78: Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, Washington, DC, 1978. IEEE Computer Society.

[22] Samuel Hung. Transaction logic prototype, 1996.

[23] Samuel Y.K. Hung. Implementation and performance of transaction logic in prolog. Master's thesis, University of Toronto, 1996.

[24] M. Kifer. FLORA-2: An object-oriented knowledge base language. The FLORA-2 Web Site. http://flora.sourceforge.net.

[25] M. Liu. Relationlog: a typed extension to datalog with sets and tuples. *Journal of Logic Programming*, 36, 1998.

[26] E. Pontelli. Logic programming with sets: Theory and implementation. Master's thesis, University of Houston, 1992.

[27] Dumitru Roman and Michael Kifer. Reasoning about the behavior of semantic web services with concurrent transaction logic. In *VLDB*, pages 627–638, 2007.

[28] Dumitru Roman and Michael Kifer. Semantic web service choreography: Contracting and enactment. In *International Semantic Web Conference*, pages 550–566, 2008.

[29] R. Sekar, I. V. Ramakrishnan, and A. Voronkov. Term indexing. In *Handbook of automated reasoning*, pages 1853–1964. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001.

[30] Amalia F. Sleghel. Concurrent transaction logic prototype, 2000.

[31] T. Swift and D.S. Warren. An abstract machine for SLG resolution: Definite programs. In *Int'l Logic Programming Symposium*, Cambridge, MA, November 1994. MIT Press.

[32] H. Tamaki and T. Sato. OLD resolution with tabulation. In *Int'l Conference on Logic Programming*, pages 84–98, Cambridge, MA, 1986. MIT Press.

[33] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[34] David Scott Warren. Memoing for logic programs. *Communications of the ACM*, 35 (3):93–111, 1992.

[35] G. Yang, M. Kifer, and C. Zhao. FLORA-2: A rule-based knowledge representation and inference infrastructure for the Semantic Web. In *International Conference on Ontologies, Databases and Applications of Semantics (ODBASE-2003)*, volume 2888 of *Lecture Notes in Computer Science*, pages 671–688. Springer, November 2003.