Evaluating Triggers Using Decision Trees

Lance Obermeyer
Applied Research Laboratories
The University of Texas at Austin
lanceo@arlut.utexas.edu

Daniel P. Miranker
Department of Computer Sciences
The University of Texas at Austin
miranker@cs.utexas.edu

Abstract

This paper presents an algorithm for implementing rule filtering in active and trigger enabled databases. The algorithm generates one or more decision trees that determine what rules or triggers might be enabled by an individual database element, reducing the number of rules or triggers that must be evaluated. The algorithm operates by symbolically representing the space of database elements and subdividing the space based on rule predicates. Regions of the state space represent particular combinations of enabled rules. Decision trees are then generated based on the subdivided state space. The trees have the important property that no individual test is repeated. The ordered binary decision diagram (BDD) data structure is used to represent and manipulate the state space.

1. Introduction

Modern database systems increasingly support active behavior through rules. This support ranges from simple database triggers to complete active database functionality. Rules typically follow the event – condition – action (ECA) model originally proposed in Hipac [13]. This rule model specifies the particular event or set of events a rule will respond to, the condition that must be true for the rule to proceed, and the action to execute if the condition is satisfied.

Many current commercial relational databases restrict their trigger subsystems to support only a single rule (trigger) per table per event per situation. For example, Oracle version 7 allows one trigger per table per event (insert, delete, update) per situation (before row, before statement, after row, after statement) for a total of 12 triggers per table [17]. However, no more than one trigger is active at any one time.

One area where active databases have an advantage over simple trigger systems is in supporting multiple rules per table. Active database rarely restrict the number of rules per table. This provides developers with the freedom to create multiple rules per table if they wish. The cost of this added freedom is in execution time. Depending upon the coupling mode [13] of the rule, rule evaluation happens within the critical path of either the triggering DML command or the enclosing transaction commit. It is therefore imperative that rule processing be as efficient as possible.

One obvious technique for speeding up rule processing is to avoid evaluating rules that cannot execute. This can be accomplished in two levels by examining the triggering tuple (in a rule with an `insert`, `update`, or `delete` event). The first level is data independent, and consists of comparing the relation name of the updated tuple to the relation name(s) mentioned in the event portion of the rule. If a rule is not sensitive to events on the updated relation, then evaluation of the condition of the rule can be skipped. An update that can be ignored based on purely schematic information has been termed a *readily ignorable update* [6].

The second level is data dependent, and consists of comparing the triggering tuple to the constant test predicates in the rule's condition. For example, figure 1 shows a schema and three rules in SQL3 style syntax from a hypothetical finance application. All three rules are sensitive to the `CreditCard` relation, so none are readily ignorable. The when clause of the rules contains other conditions that are omitted for brevity. Similarly, the actions of the rules are also omitted.

If the triggering tuple does not satisfy the constant tests of a rule, the rule cannot fire. Evaluation of these rules can stop with an examination of the triggering tuple, and no other actions need be performed. We term the process of comparing a triggering tuple to the constant test predicates of a rule condition for the purpose of eliminating some rules from consideration *trigger filtering*. Researchers in the expert system community have long used trigger filtering to speed up system execution. For example, the RETE [9] and TREAT [14] match algorithms use trigger filtering techniques to generate the elements in alpha memories (a form of specialized predicate index).

Formally, we define a program as consisting of a set of rules. Each rule contains a possibly empty condition consisting of a set of predicates connected by the standard Boolean connectives. Predicates can be of two types, *filtering predicates* and non-filtering predicates. We define a filtering predicate as any predicate that can be evaluated with respect to a triggering tuple. Filtering predicates come in three types. The most common type is a constant test, which compares a tuple or attribute of a tuple to a constant value. In the above example `n.limit = 5000` is a constant test. The second type is an intra-tuple test, which compares an attribute of a tuple with another attribute of the same tuple. In the above example, `n.balance > n.limit` is an

intra-tuple test. The final type is a function call[1], which combines constant tests and intra-tuple tests behind a function. Non-filtering predicates include join predicates and aggregation predicates. For any rule, we term the set of filtering predicates joined by their appropriate connectives the *filtering expression* for the rule. The filtering predicates for the rules in figure 1 are shown in figure 2.

A straightforward way of implementing trigger filtering for multiple rules is simply to maintain a list of rules and their respective filtering expressions. This may be wasteful because individual predicates may be repeated, as is the case in the example with the predicate `n.type = "gold"`.

This paper describes a technique for efficiently solving the trigger filtering problem. The technique generates one or more decision trees. Added, modified, or removed elements are evaluated by the trees and the set of rules whose filtering expression is satisfied is identified. The decision trees have the advantage that no tests are duplicated. Thus any database `insert`, `update`, or `delete` into a monitored table induces a bounded length descent of the tree.

Many other researchers have studied trigger implementation. Buneman and Clemons report on a technique for trigger filtering based on a limited evaluation of the complete rule, not just the filtering expression. Their technique first removes readily ignorable updates, then, for remaining updates, evaluates complete trigger expressions over restricted size data sets. Join predicates are evaluated, but relative to small size relations [5]. Cohen [8] describes a method of compiling trigger conditions into a match network similar to RETE [9]. This work is focuses on evaluating the complete trigger expression, not filtering out unsatisfiable triggers. Similarly, the large bodies of work on production system match algorithms and database integrity constraints primarily focus on improving join performance. The only published work known to the authors on the trigger filtering problem is by Hanson and Johnson, who report on a technique for solving the trigger filtering problem [11]. Their methodology is discussed in section 2.2. Blakeley, Coburn, and Larson report on the somewhat related problem of detecting updates to base relations that are irrelevant to materialized views [1]. Their work is based on determining the relationship between many elements and one rule (view definition), whereas the trigger filtering problem determines the relationship between one element and many rules. In the degenerate case of one element and one rule, the two are essentially the same.

The remainder of the paper is organized as follows. Section 2 presents three methods for solving the trigger filtering problem. The third method is using decision trees, which is the focus of this paper. Section 3 presents an algorithm to generate decision trees using an algorithm based on binary decision diagrams (BDDs) [3]. Section 4 presents an empirical evaluation of the decision tree technique. Section 5 lists three areas of future work. Section 6 provides concluding remarks.

```
CreditCard(name,balance,payment,
           limit,type,state);

create trigger T1
after insert on CreditCard
referencing new as n
when (n.limit = 5000 and
      n.state = "TX" and
      n.type = "gold" and ...)

create trigger T2
after insert on CreditCard
referencing new as n
when (n.type = "gold" and
      n.balance > n.limit and ...)

create trigger T3
after insert on CreditCard
referencing new as n
when (n.balance > n.limit and
      n.payment > 1000 and ...)
```

Figure 1 Example Rules

The remainder of the paper uses relational terminology. However, the technique presented is applicable to filtering insert, update and remove events in object oriented databases as well. Events that do not compare a specific object against a set of rules, such as temporal events, will not fit well within this framework.

2. Possible Solutions

This section presents an overview of three methods to implement a trigger filtering solution. As stated previously, the purpose is to determine the individual rules that pass the trigger filtering stage for a given database object. In the following examples, calling the function `signal` with the rule name as a parameter indicates passing filtering. The database object being evaluated is indicated by the variable n.

The methods presented in this section are applicable to both interpreted and compiled environments.

2.1 Naïve

The straightforward solution is termed *naïve*, and is a single `if` statement for each filtering expression. Thus for *n* filtering

```
FP1: n.limit = 5000
FP2: n.state = "TX"
FP3: n.type = "gold"
FP4: n.balance > n.limit
FP5: n.payment > 1000
```

Figure 3 filtering predicates

```
if(FP1 ∧ FP2 ∧ FP3) signal(T1)
if(FP3 ∧ FP4)   signal(T2)
if(FP4 ∧ FP5)   signal(T3)
```

Figure 2 Naive

[1] Many active databases, especially those adopting a C based rule syntax, allow calling functions in the condition portion of a rule. These functions must be side effect free, and be strict functions on the input tuple. Example systems include VenusDB [15], SAMOS [10], and Reach [4].

expressions, there are n individual `if` statements. Figure 3 shows naively generated code for the rules in figure 1. The statements have the problem that filtering predicates FP3 and FP4 are evaluated twice. The size of the statements and the maximum number of tests through the statements are equal to the summation of the filtering predicates in each filtering expression. The minimum number of tests is the number of filtering expressions.

2.2 Interval Skip List

An alternative method is proposed by Hanson and Johnson [11]. They propose a technique relying on intervals. For each filtering expression, an interval over an attribute is selected. Intervals can be bounded ($0 < n.value < 1$) or unbounded ($n.value < 1$). An equality test is treated as an interval of width 0. For each attribute in the structure, an interval list is maintained. Each filtering expression is entered into the interval list for its selected interval's attribute. Thus each expression is entered into one and only one interval list. They recommend the interval skip list as the implementation data structure.

The filtering expressions satisfied by a particular database tuple are determined by the following algorithm. For each attribute of the element, a stabbing query logarithmic in the number of entered expressions is executed on the attribute's interval list with the element's attribute value as the key. This determines a possibly empty set of filtering expressions that are partially satisfied by the tuple. For each partially satisfied expression, the remaining filtering predicates are evaluated relative to the tuple in a linear fashion.

This method has three limitations, however. First, it only works with conjunctive filtering expressions. Any filtering expression that include disjunction is transformed into its equivalent disjunctive normal form expression, and each resulting conjunctive clause is separately managed. Since the number of clauses in a statement converted to DNF is in the worst case exponential in the size of the original expression, a potentially explosive number of intervals may need to be maintained. Also, correct execution of a system may dictate only a single notification message per rule for a given database tuple. If the disjunctive expression is broken up into multiple clauses, and more than one clause is satisfied by a given tuple, then multiple notifications may result. To correct this, either the generated clauses must be mutually exclusive, requiring the complete set of tests per clause, or a separate post processing mechanism must be used to filter out possible multiple calls.

Second, the method relies on every conjunctive clause having one attribute that is both selective and over an ordered domain. Conjunctive clauses not possessing these two qualities are inserted into a separate list managed via a naïve method. In essence, these non-indexed filtering expressions are treated as having an infinite interval.

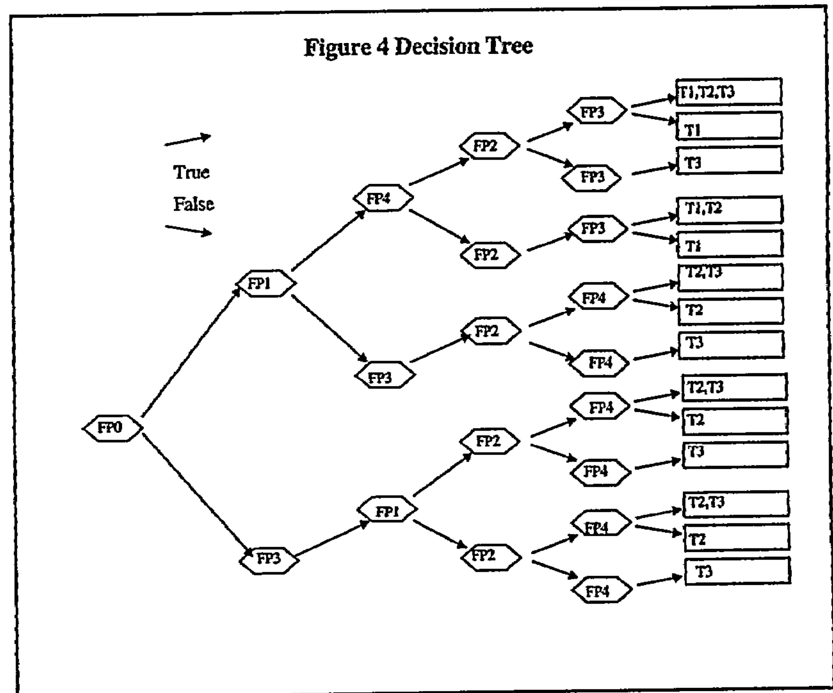

Figure 4 Decision Tree

Third, once the partially satisfied filtering expressions are determined by indexing into an interval list, the remaining filtering predicates in each expression are sequentially evaluated. Thus individual filtering predicates may be tested more than once. For example, consider the following two filtering expressions.

```
n.a1 = true and n.a2 = true and n.a3 = true
n.a1 = true and n.a2 = true and n.a4 = true
```

Assume the first predicate of the above two expressions is entered into the interval list (interval width 0), and the database tuple has

Label	State(s)
T1	FP1 ∧ FP2 ∧ FP3 ∧ ¬FP4 ∧ FP5 ∨ FP1 ∧ FP2 ∧ FP3 ∧ ¬FP4 ∧ ¬FP5
T2	FP1 ∧ ¬FP2 ∧ FP3 ∧ FP4 ∧ ¬FP5 ∨ ¬FP1 ∧ FP2 ∧ FP3 ∧ FP4 ∧ ¬FP5 ∨ ¬FP1 ∧ ¬FP2 ∧ FP3 ∧ FP4 ∧ ¬FP5
T3	FP1 ∧ FP2 ∧ ¬FP3 ∧ FP4 ∧ FP5 ∨ FP1 ∧ ¬FP2 ∧ ¬FP3 ∧ FP4 ∧ FP5 ∨ ¬FP1 ∧ FP2 ∧ ¬FP3 ∧ FP4 ∧ FP5 ∨ ¬FP1 ∧ ¬FP2 ∧ ¬FP3 ∧ FP4 ∧ FP5
T1,T2	FP1 ∧ FP2 ∧ FP3 ∧ FP4 ∧ ¬ FP5
T2,T3	FP1 ∧ ¬FP2 ∧ FP3 ∧ FP4 ∧ FP5 ∨ ¬FP1 ∧ FP2 ∧ FP3 ∧ FP4 ∧ FP5 ∨ ¬FP1 ∧ ¬FP2 ∧ FP3 ∧ ¬FP4 ∧ FP5
T1, T2,T3	FP1 ∧ FP2 ∧ FP3 ∧ FP4 ∧ FP5

Figure 5 List Representation

an `a1` value of `true`. Thus stabbing into the interval list will retrieve both the above two expressions. Evaluating the two expressions will involve testing the `n.a2 = true` predicate twice.

2.3 Decision Tree

This paper proposes using decision trees with the important feature that no test is repeated. By virtue of this property, the maximum number of tests during a descent of the tree is bounded by the total number of unique filtering predicates. Additionally, a well constructed tree can have a minimum and average path less than that of a naïve version. The decision tree for the rules in figure 1 are shown in figure 4. In this tree, the maximum number of tests is five, two tests less than the amount in figure 3. Similarly, the minimum number of tests is two ($\neg$FP0 $\wedge$ $\neg$FP3), whereas the minimum in figure 3 is three.

Generating optimal decision trees is a known NP-complete problem [12]. Therefore, a heuristic method must be used to generate the tree. This method is presented in the next section.

3. Algorithm

Assume the set of filtering expressions is E, with each element e containing some subset of the set P of filtering predicates. Each filtering predicate can be either true or false. With respect to the filtering predicates, the state space can be viewed as n-cube with $|P|$ dimensions. Individual cells within the cube can be labeled with the maximal set of filtering expressions satisfied at that state. In other words, each cell contains an element from the powerset of the set of filtering expressions.

An alternative method of representing the above information is a list of [state, label] pairs, where a state is a conjunction of the variables, some possibly negated. When multiple cells have the same label, then a particular label is defined by a set of states. The special case of the label $\varnothing$ can be ignored. This is shown in figure 5.

3.1 List Construction

The purpose of this phase is to generate a list of labels and their corresponding set of states. We formally define the following terms.

- a state space S is the set of all possible database objects for a given relation.
- a filtering expression $f \in F$ is a Boolean valued function on $s \in S$.
- a label L is an element from the powerset of F. Thus L defines a possibly empty set of filtering expressions.
- a characteristic function for a label L, denoted $cf(L)$, is a Boolean valued function on a state $s \in S$ such that $cf(L)(s) = \text{true} \Leftrightarrow \forall f \in L[f(s) = \text{true}]$ [7]. Intuitively, $cf(L)$ defines a subset of the state space S where every element s satisfies all of the filtering expressions in L.

```
Procedure GenerateClassifications(in F, out C)
    Initialize classification list to empty
    For each filtering expression f in F
        Initialize temporary variable t to false
        For each existing classification c
            If c.cf implies f.cf
                Add f.id into c.l
                Set t to t ∨ c.cf
            else if c.cf intersects f.cf
                Remove c from classification list
                Copy c into two separate classifications,
                    c1 and c2
                Add f.id into c1.l
                Set c1.cf to c.cf ∧ f.cf
                Set c2.cf to c.cf ∧ ¬f.cf
                If c1.cf is satisfiable
                    Insert c1 into C
                If c2.cf is satisfiable
                    Insert c2 into C
                Set t to t ∨ c1.cf
        If (t != f.cf)
            Insert [f.id, f.cf – t] into C
    Return classification list C
```

Figure 6 Classification Algorithm

- a *classification* is a [label, characteristic function] pair.

Rather than enumerating the individual states for a label, we instead provide the characteristic function. Thus the goal is to generate a list of classifications.

The algorithm is shown in figure 6. The notation c.l is an abbreviation for the label of a classification c, and c.cf is an abbreviation for the c's characteristic function. It works through an inductive style process of comparing a filtering expression f to each existing classification c. With each comparison, there are two interesting outcomes. The first is that the c's set of states is a subset of the f's set of states. In this instance, the filtering expression is added to the classification's label. The second interesting outcome is if there exist states within the c's set of states that are within the f's set of states and there exist other states within the c's set of states that are not within the f's set of states. In this instance, the classification c is split into two classifications, $c1$ and $c2$. The first classification corresponds to states within both c and f. The label for this classification is set to the label of c plus f. The second corresponds to states within c but not within f. This classification receives the label of c. Note that the union of the states within $c1$ and $c2$ is exactly c. For both $c1$ and $c2$, a satisfiability check is made, and satisfiable classifications are inserted into the classification list C. The satisfiability check prunes classifications that are semantically impossible, such as `n.value = 1` $\wedge$ `n.value = 2`. If all states for f are not accounted for by the above two items, a new classification is created containing only those remaining states.

The above algorithm can be implemented quite easily by employing the Ordered Binary Decision Diagram (BDD) data structure [3]. BDDs are useful for representing functions over binary variables. For this application, filtering predicates are viewed as atomic variables, and filtering expressions are BDD functions over the variables. In the above algorithm, the characteristic functions for classifications are calculated by joining filtering expressions, expressed in BDD form, using the Boolean operators $\wedge$, $\vee$, and $\neg$, which are directly supported by BDDs. It must be noted that for certain classes of functions BDDs behave very badly, with a worst case exponential size bound with respect to the number of variables [3]. While we have not experienced any BDD blowup in practice, further work is required to accurately determine the likelihood of BDD blowup. Similarly, the Boolean satisfiability issues are handled with excellent average case performance but without polynomial guarantee.

3.2 Tree Generation

Given a list of classifications, the next task is to generate a binary decision tree such that any leaf of the tree corresponds to one and only classification. Given that optimal binary tree generation is a known NP-complete problem [12], the method will necessarily be heuristic.

This phase is modeled after decision tree generation in ID3, a machine learning algorithm for classifying items based on features [18]. Like ID3, this algorithm depends upon a heuristic to select the feature to test at each level. The heuristic used here is similar to the ID3 information gain heuristic.

The algorithm, shown in figure 7, works by greedily choosing a filtering predicate p to test, outputting the test to the decision tree, and recursing on the then and else sides. The then side is parameterized by the classifications restricted by p = true. The else side is parameterized by the classifications restricted by p = false. Recursion continues until C is empty. Classifications are removed from C in one of two ways. A classification equal to true implies the classification's characteristic function is satisfied along the current path. The classification's label is output and the entry removed. Conversely, a classification equal to false implies the classification cannot be satisfied along the current path, so the classification is removed. Note that this all occurs at compile time, not runtime.

A heuristic is involved with choosing the filtering predicate to test at a point. The heuristic is modeled after the ID3 information gain heuristic [18]. If a particular filtering predicate is required by every characteristic function in a set of classifications, then that predicate is chosen. If no required predicate exists, then the predicate that splits the fewest number of classifications is chosen. Here, splitting means separating adjacent cells in the n-cube that have the same label.

As with the classification generation step, set and Boolean operations are implemented using BDDs.

3.3 Space Enhancement

The worst case space complexity of the decision tree method is linear in the number of classifications, which itself is exponential in the number of filtering predicates. This occurs when each new filtering expression intersects with every previous classification in the classification generation algorithm, doubling the number of classifications at that point.

```
Procedure GenerateTree(in C)
    For each c in C
        If c.cf = true
            Output c.l
            Delete c from C
        If c.cf = false
            Delete c from C
    If C is empty
        Return
    p = ChoosePredicate(C)
    Output p
    Generate Tree(C | p)
    GenerateTree(C | ¬p)
```

Figure 8 Tree Generation

One way to partially mitigate this explosion is to segregate the filtering expressions into different groups such that no member of a group completely intersects with every other member. This can be accomplished using the algorithm in figure 8. This enhancement works by organizing the filtering expressions into connected components, and applying the decision tree generation technique to each component individually.

Note that this enhancement preserves the important feature that no filtering predicate is tested more than once is still preserved.

4. Empirical Evaluation

The authors believe the decision tree method of trigger filtering is superior to the typical naïve method because it results in, on average, fewer tests at runtime. This is demonstrated in section 4.2. The cost of the method is in tree generation time, which is only done once, and in tree size at runtime. Tree generation time, while not reported here, generally takes less than one second, with the largest trees taking tens of seconds.

The decision tree generation method was evaluated using two criteria, the total number of nodes in the trees and the average path length through the trees. Average path length is the most important measure, since that is the runtime speed measure.

Generate an undirected graph. The filtering predicates represent nodes. Two nodes are connected if they appear together in a filtering expression.

Split the graph into connected components

For each component, determine the set of filtering predicates and create a set of filtering expressions .

For each set of filtering expressions, apply the decision tree generation procedure.

Figure 7 Enhancement

Two workloads were analyzed, synthetic and actual. The synthetic workload consists of randomly generated filtering expressions over a single table. There were three degrees of freedom in the generator, the total number of filtering predicates, the maximum number of predicates per expression, and the total number of expressions generated. Two reasonable values were chosen for each parameter, for a total of eight separate trials. The actual workload consisted of classifications from four expert system programs from the Texas Benchmark Suite, a suite of rule programs from the expert system community [2].

4.1 Test count

The total number of tests in a tree is a size measure. The desired result is a test count that approaches that of naive. Decision trees were generated using the naïve and decision tree approaches for different compositions of rules.

The test counts for the actual and synthetic workloads are shown in tables 1 and 2. In all instances, the decision tree method generated more tests than the naïve method. This is expected. In three instances (10-3-10, 10-6-10, and mom), the decision tree was extremely large.

Table 4. Test Counts Synthetic Workload

	Naive	*Tree*
10-3-5	10	30
10-3-10	19	511
10-6-5	13	103
10-6-10	24	999
30-3-5	10	16
30-3-10	20	71
30-6-5	13	98
30-6-10	25	111

Table 3. Test Counts Actual Workload

	Naive	*Tree*
Mab	37	56
Mom	75	1086
Waltz	27	23
Waltzdb	25	27

Table 1. Path Length Synthetic Workload

	Naive	*Tree*
10-3-5	12.3	8.3
10-3-10	22.3	10
10-6-5	12.8	10.0
10-6-10	25.5	11.0
30-3-5	12.3	9.7
30-3-10	24.3	16.8
30-6-5	12.8	11.0
30-6-10	25.8	20.8

Table 2. Path Length Actual Workload

	Naive	*Tree*
Mab	18.4	10.8
Mom	7.2	5.9
Waltz	21.7	9.4
Waltzdb	10.0	7.8

4.2 Path Length

The path length through the decision tree is a speed measure, and is therefore the primary figure of merit. Average path length was measured. This number represents the average number of tests performed during the trigger filtering stage in response to an individual database element. A lower number represents less work performed at runtime.

The system was tested under two workloads, a synthetic workload and a set of benchmark programs. The synthetic workload consisted of 10,000 randomly constructed elements. The exact same instance stream was used for each test. The instances were fed through each of the trigger filters generated during test count testing.

The average path lengths for the actual and synthetic workloads are show in tables 3 and 4. In all cases, the decision tree method outperformed naïve by having a smaller average path length.

5. Future Work

Three areas of future work remain. The first is to develop a heuristic that further divides a connected component if the component is going to explode in space. The second is to integrate attribute testing costs into the tree generation heuristic. This will allow generation of efficient trees for environments where there are non uniform predicate test costs. An example would be testing over complex data types such as images. A final area of future work is to support dynamic addition and deletion of filtering expressions. The current scheme completely regenerates the classification list and decision trees when the set of filtering expressions is modified.

6. Conclusion

This paper presented an algorithm for implementing rule filtering in active and trigger enabled databases. The core algorithm generates a decision tree that determines what rules or triggers might be enabled by an individual database element. This is accomplished by descending a database element down a decision tree. When the element reaches a tree leaf, the leaf contains a label listing the rule or trigger identifiers that might be enabled. The paper also presents an extension based on grouping rules into connected components that helps alleviate occasional exponential space explosion. In a sense, the decision tree represents a function whose domain is the set of all elements in a particular relation and whose range is the powerset of rules. It is important to note that, at runtime, the function is only evaluated with respect to individual elements. Boolean satisfiability calculations are not attempted at runtime.

Decision trees generated by the technique outlined in this paper have the important property that the maximum path from the root to a leaf is bounded by the total number of filtering predicates, which is necessarily less than or equal to the maximum path for the straightforward naïve

technique. Empirical evaluation shows that the average number of tests through the decision tree is less than the average number of tests using the naïve method for all tests data sets.

The algorithm is based on a symbolic representation of the space of database elements. The state space is iteratively subdivided into regions that represent particular combinations of enabled rules. The ordered binary decision diagram (BDD) data structure is used to quickly and easily represent and manipulate the state space.

7. Acknowledgements

The idea of subdividing the state space into classifications is due to an unpublished paper by Satoshi Nishiyama, Keith Goolsbey, and Daniel P. Miranker. A partial implementation is describe in [16].

8. References

1. J. Blakeley, N. Coburn, and P.-A. Larson, "Updating derived relations: detecting irrelevant and autonomously computable updates," *ACM Transactions on Database Systems*, vol. 14, no. 3, September, pp. 369-400, 1989.

2. D. Brant, T. Grose, B. Lofaso, and D. P. Miranker, "Effects of database size on rule system performance: Five case studies," in *Proceedings of the 17th International Conference on Very Large Data Bases*. Barcelona, Spain, September, 1991, pp. 287-296.

3. R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys*, vol. 24, no. 3, September, pp. 293-318, 1992.

4. A. Buchmann, J. Zimmermann, J. Blakeley, and D. Wells, "Building an integrated active OODBMS: Requirements, architecture, and design decisions," in *Proceedings of the 11th International Conference on Data Engineering*. Taipeh, Taiwan, March, 1995, pp. 117-128.

5. O. P. Buneman and E. K. Clemons, "Efficiently monitoring relational databases," *ACM Transactions on Database Systems*, vol. 4, no. 3, September, pp. 368-382, 199.

6. Buneman. O. P. and E. K. clemons, "Efficiently monitoring relational databases," *ACM Transactions on Database Systems*, vol. 4, no. 3, September, pp. 368-382, 1979.

7. E. Cerny and M. A. Marin, "An approach to unified methodology of combinational switching circuits," *IEEE Transactions on Computers*, vol. 8, August, pp. 45-756, 1977.

8. D. Cohen, "Compiling complex database transition triggers," in *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*. Portland, OR, May, 1989, pp. 225-234.

9. C. Forgy, "RETE: A fast match algorithm for the many pattern/many object pattern match problem," *Artificial Intelligence*, vol. 19, pp. 17-37, 1982.

10. A. Geppert, S. Gatziu, K. R. Dittrich, H. Fritschi, and A. Vaduva, *Architecture and implementation of the active object-oriented database management system SAMOS*, TR 95.29. Institut fur Informatik, Universitat Zurich, Switzerland, 1995.

11. E. Hanson and T. Johnson, "Selection predicate indexing for active databases using inteval skip lists," *Information Systems*, vol. 21, no. 3, pp. 269-298, 1996.

12. L. Hyafil and R. L. Rivest, "Constructing optimal binary decision trees is NP-complete," *Information Processing Letters*, vol. 5, no. 1, 1976.

13. D. R. McCarthy and U. Dayal, "The architecture of an active database management system," in *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*. Portland, OR, May, 1989, pp. 215-224.

14. D. P. Miranker, *TREAT: A new and efficient match algorithm for AI production systems*. Los Altos, CA: Pittman/Morgan-Kaufman Publishers, 1989.

15. D. P. Miranker and L. Obermeyer, "An overview of the VenusDB active multidatabase system," International Symposium on Cooperative Database Systems for Advanced Applications. Kyoto, Japan, December, 1996.

16. S. Nishiyama, "Optimizing compilation of select phase of production systems," Master's thesis. Department of Computer Sciences, The University of Texas at Austin, 1991.

17. Oracle Corporation, *Oracle 7 users guide*1992.

18. J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, pp. 81-106.

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee.

CIKM *97 LasVegas Nevada USA*

Copyright 1997 ACM 0-89791-970-x/97/11..$3.50