Integrated Alerting for Structured and Free-Text Data in TriggerMan[1]

Eric N. Hanson and Himanshu Raj
301 CSE, CISE Department
University of Florida
Gainesville, FL 32611-6120
hanson@cise.ufl.edu, hr0@cise.ufl.edu
TR-01-009
July 17, 2001

Abstract

TriggerMan is a highly scalable, asynchronous database triggering and alerting system. In this paper, we present strategies to enhance TriggerMan to support triggers involving free-text predicates—predicates defined on unstructured text fields. These techniques include a new free-text predicate indexing method called the trie filter, and optimization techniques for choosing the best predicates and keywords for indexing. We introduce a new text matching function called tm_contains() that allows users to specify text conditions using a Boolean, weighted Boolean, or vector space model. This function is analogous to the text matching user-defined routines available for current object-relational database products. Our approach can choose the best predicate to index on, whether it is a structured predicate, such as an equality condition, or a free-text predicate.

Introduction

Modern databases contain a mixture of both structured data and free-form text. Recently, text-search extension modules have become available for object-relational databases, e.g., Verity [Veri01], Excalibur [Exca01] etc. These modules allow users to retrieve data using information-retrieval operations combined with traditional relational operators such as <, >, <>, =, ≤, ≥, and the SQL LIKE operator. Database rule systems, including built-in trigger systems and outboard trigger engines such as TriggerMan [Hans99], should also support free-text predicates in their conditions when the DBMS supports free-text querying. Similarly, systems that support alerting on streams of structured text (XML) messages, such as NiagaraCQ, could also be enhanced with free-text alerting capability [Chen00, Liu99]. Otherwise, there will be a mismatch between the type of predicates users can specify in queries and rule conditions.

Building a predicate index that allows efficient and scalable testing of a large set of rule conditions that contain a combination of traditional in free-text predicates is a challenge. This paper describes the design of an outboard database alerting system that supports a combination of free-text and traditional predicates. The design is an extension of the TriggerMan architecture.

We extend the TriggerMan condition language with a new function,

[1] This research was supported by the National Science Foundation under grant IIS-9820839.

tm_contains(document, query [, threshold])

For example, consider this table schema:

video(vid, title, date, category, description)

You can create a TriggerMan trigger to alert a user when a new video is defined that has category “sports” and contains the words “David Lee” using this notation:

```
create trigger lee
from video
when category = “sports” and tm_contains(description, “David Lee”)
do …;
```

This trigger fires if a new video record has category “sports” and a description that contains “David Lee” as a phrase.

The original version of TriggerMan indexed on equality predicates only. Indexing only on equality misses opportunities for efficient indexing in some cases. For example, suppose that 50% of all videos had category “sports” and only 0.1% contained the phrase ”David Lee” respectively in their descriptions. The tm_contains clause is thus more selective than the equality clause. This suggests we should index on it and not on the equality.

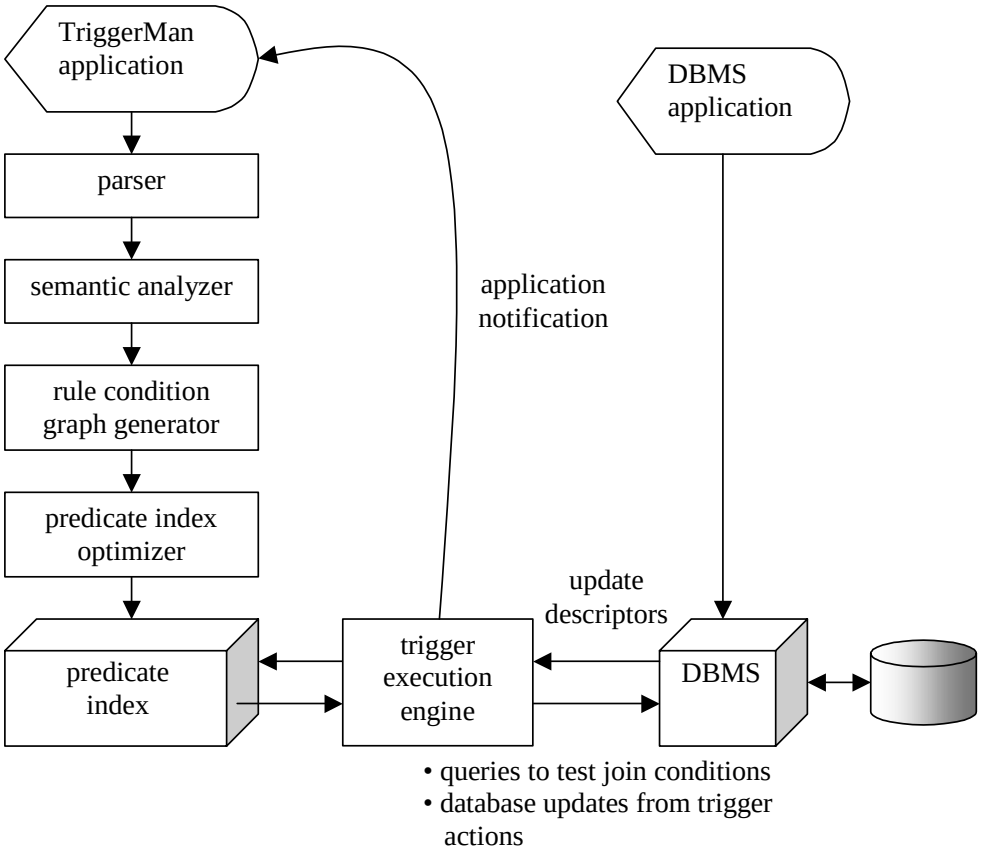

Figure 1. Architecture of TriggerMan enhanced with free-text alerting.

By including tm_contains() as an operator that can be used in a TrigerMan rule condition, we extend the system with Selective Dissemination of Information (SDI) capabilities, which have previously been studied in the information retrieval field [Yan94]. In addition, SDI capabilities for structured text documents have been discussed by Altinel et al. [Alti00]. We do not incorporate structured text alerting capabilities here. Rather, we support a combination of alerting based on structured data fields and unstructured text fields.

In the rest of this paper, we describe a method to efficiently index rule predicates on either equality predicates or tm_contains. Our solution includes a new type of predicate index tailored to the tm_contains function, and an optimization strategy that determines which predicate clauses to index.

Architecture

The architecture of the version of TriggerMan that integrates trigger condition testing for both structured data and free-text is illustrated in Figure 1.

This architecture is a natural extension of the original TriggerMan architecture. The architecture now includes a separate *predicate index optimizer* that analyzes selection predicates and determines which clauses to index.

TriggerMan supports the notion of an *expression signature*. A specific type of expression signature may be associated with a type of predicate index. TriggerMan uses a skip list [Pugh90] to index equality predicates of the form ATTRIBUTE = CONSTANT. The extended TriggerMan system can index tm_contains() predicates as well. We introduce a new predicate index component called the *trie filter* to support tm_contains(). The trie filter is described later in the paper.

The tm_contains() Function for Free-Text Matching

The general syntax of the tm_contains() function is given as following:

tm_contains(DocumentAttribute, QueryString [, Additional Parameters]).

The DocumentAttribute is the column name of the table that holds the document. Here we assume only unstructured text (a string of characters) as the data type of this column. An example type for such a column might be 'Varchar(255)' or 'CLOB' [Info01].

The QueryString is composed of *query words* and *operators*, and is enclosed within a pair of quotes (' '). A query word can be either a *word*, or a *phrase*. A word is a sequence of characters excluding white space, optionally enclosed within a pair of double quotes (" "). It can optionally contain a wild card character ('*')[2] at its right end. For example, "david", david, dav*, "davi*" are all valid instances of a word. A phrase is simply two or more *words* separated by white space and enclosed within a pair of double quotes (" "). Some valid examples for phrases are "David Lee", "Davi* Le*" etc. Stemming of query words is not implemented in the preliminary version of tm_contains, however it can be simulated to an extent by using wildcards.

[2] '*' matches any number of characters

The following *operators* are supported by tm_contains() to facilitate free-text matching:
ACCRUE (‘,’) – This is perhaps the most commonly used operator while performing free-text queries using the vector space model [Baez99] (the actual symbolic operator might be different, e.g. in some search engines, a white space performs the same job). The document is searched for all the query words separated by this operator, and a combined score is calculated (specific details of the scoring function for all the operators are presented later in this section). In case this score value is at least as much as the required score value (*threshold*), the match succeeds.
AND – the logical *AND* operator. A match *partially succeeds*[3] only when all the query words separated by this operator exist in the document. Exclusion of even one word disqualifies the document against match, hence explicitly zeroing out the score of the match. We say “partially succeeds” rather than “succeeds” here because even partial success does not guarantee that the score rises above a specified threshold.
OR – the logical *OR* operator. A match partially succeeds when any of the words separated by the operator exist in the document.
NOT – the logical NOT operator. This operator has the usual Boolean semantics when used with AND/OR. However, using it with ACCRUE implies a different semantics. This issue is addressed later in the section. The NOT operator cannot exist alone, and requires the presence of at least one other operator. A restriction on the use of this operator is that it can only precede a query word, and not any general expression. Additional restrictions on the use of NOT appear later in the paper.

These operators may co-exist in a query string, and precedence rules apply wherever applicable. Precedence of *AND* is higher than both *OR* and *ACCRUE*, which have the same precedence. Precedence rules can be overridden by appropriately parenthesizing the query words. For example, X AND Y OR Z is implicitly interpreted as (X AND Y) OR Z. It can also be stated as X AND (Y OR Z). The interpretation of the query string by tm_contains() will be different for both the cases.

Additional parameters include either a numeric *threshold* between 0 and 1.0, or the keyword *BOOLEAN*. The additional parameters are optional, and the system uses a default value of threshold in case it is not stated explicitly. The keyword BOOLEAN forces any match with nonzero score value to succeed.

As it can be observed from the operators, the tm_contains operator uses an extended vector space model for specifying the query strings. There are several semantic issues associated with the model. Firstly, the use of Boolean operators (AND/OR/NOT) is meant to provide the matching as implied by the Boolean model, along with some insight on the relevance of the document based on the score. This score is calculated using a scoring strategy, discussed later in the next section. For example, merely matching the document (in the Boolean sense) with the QueryString does not qualify the tm_contains operator to succeed, unless the score of the document is greater than or equal to a threshold value. For using strict Boolean semantics, one should use the keyword BOOLEAN, as specified above.

Secondly, though one may use AND and/or OR operators with the ACCRUE operator, the semantics is not well defined in that case, albeit the system will return a score value. However

[3] For a complete match against the query string, the score value of the match must be at least as much as the required score value (specified explicitly or otherwise).

the user might use NOT with ACCRUE, the semantics of which is as follows. The system will match the non-negated terms first, and will come up with a *partial score*. In case the partial score is greater than the threshold, the document will be searched for the negated terms. Lest any of the negated terms should appear, the score will be poisoned to zero, and the search will terminate unsuccessfully. Only when none of the negated terms appears will the match be declared as successful. Conjunctive terms consist of two or more words separated by AND or ACCRUE. We require the existence of at least one non-negated query word in each conjunctive term in the QueryString. This is so that there is at least one non-negated query word to index on, which is required for correct operation of our trie filter.

Setting Thresholds Using a Test-Run Facility

In order to set thresholds effectively for tm_contains() operators in trigger conditions, a test run facility is essential. This facility would allow users to select thresholds by looking at example data ranked by score. Without a test run facility, users will find it difficult or impossible to choose threshold values.

We recommend that application developers allow users to select thresholds for the tm_contains operators that appear in the conditions of triggers created by their applications in the following way:

- Allow the user to enter their trigger condition, *Q*, using a convenient user interface, such as a form.
- Run a query corresponding to this trigger condition, and rank the output in descending order of the weight produced by the free-text matching function used (e.g. vtx_contains()).
- Allow the user to select the lowest-ranked tuple that meets their information requirement. Call this tuple *i*.
- Extract the document field of tuple *i*. Call the document value *D*.
- Compute the score for tm_contains(*D, Q)* using an API routine tm_score() provided with TriggerMan. Call this score *S*.
- Use *S* as the threshold when creating the user's trigger.

The drawback of this approach is that the value computed by tm_score() will not be exactly the same as that determined by the free-text query routine (vtx_contains() in this case). Nevertheless, there is a close correspondence between the ranking produced by the free-text query routine, and by tm_contains(). The correspondence is accurate enough to allow the user to set a meaningful threshold.

As an example, consider again the "video" table, plus an additional table with the following schema:

viewed_by(vid, name, timestamp, comments)

This table tells who has seen a video and when. It also gives that person's comments on the video in text format. Suppose a user wants to be notified when Bob views a video in category "Sports" and includes the words "Michael," "Jordan" and "Golf" in his description. Furthermore, the application developer has chosen to use weighted matching to allow users to set their thresholds. The application needs to create the following trigger:

```
create trigger JordanGolf
from video as v, viewed_by as b
after insert to b
when b.vid = v.vid
and v.category = "Sports"
and b.name = "Bob"
and tm_contains(b.comments, "Michael, Jordan, golf", THRESH)
do …
```

To let the user choose a threshold THRESH to use in the tm_contains operator in this trigger, the application constructs and runs the following query, in which XYZ_contains represents some free-text query function provided with an object-relational DBMS:

```
select *, tm_score(b.comments, "Michael, Jordan, golf") as s
from video as v, viewed_by as b
when b.vid = v.vid
and v.category = "Sports"
and b.name = "Bob"
and XYZ_contains(b.comments, "Michael, Jordan, golf")
order by tm_score(b.comments, "Michael, Jordan, golf") desc
```

The output of this query is presented to the user, who draws a line at tuple t to select a threshold. The application then computes THRESH as follows:

```
THRESH = tm_score(t.comments, "Michael, Jordan, golf")
```

The application then creates the trigger JordanGolf above, using the specific value of THRESH computed by tm_score().

The preceding example assumes that TriggerMan is being used with an object-relational DBMS, and that it provides a user-defined routine tm_score() for use in queries. In an application of TriggerMan with a relational DBMS without object-relational capabilities, application developers will be required to retrieve data out of the DBMS, and then apply the tm_score() API routine to the appropriate text field to help generate ordered output to allow the user to select a threshold.

We place the following restrictions on where tm_contains() can be used, for the reasons stated:

- You may not use more than one tm_contains() on a single tuple variable in a trigger condition because then it is impossible for applications to do a test run to let the user choose a threshold for all of them effectively. For example, the following is illegal:

  ```
  create trigger t from r when tm_contains(r.x, …) and tm_contains(r.y,…) do…
  ```

- You can only put a tm_contains() on the event tuple variable. The event tuple-variable is the one that appears in the AFTER clause, or the sole tuple variable in the case of a single-table trigger. We impose this restriction because otherwise we would have to use a text extension module routine like vtx_contains() when generating join callback queries to process join trigger conditions [Hans99]. However, a routine such as vtx_contains() will not use the same scoring function as tm_contains. The cost and complexity of putting

tm_contains() on other than the event tuple variable, as well as the possibly inconsistent semantics, make it not worthwhile.

Calculation of score of the document against a query

Before we state the scoring function, *sim(q,d)*, specifying the similarity between a query *q* and a document *d*, we need to define the following:

Definition Let N be the total number of documents in the system and n_i be the number of documents in which the word k_i appears. Let $freq_i$ be the raw frequency of k_i in document d. Then, the normalized frequency f_i of k_i in document d is given by

$$f_i = \frac{freq_i}{\max_l freq_l}$$

where the maximum is computed over all the words, l, mentioned in document d. If k_i does not appear in d, then $f_i = 0$. Further, let idf_i, the inverse document frequency for k_i, be given by

$$idf_i = \log \frac{N}{n_i}$$

The weight of k_i, associated with the document d is given by

$$w_i = f_i * idf_i$$

As we assume that a query word is not repeated in the query string, there is no weight associated between the query word and the query string.

In case the query word is a phrase, the weight of the query word is assigned to the weight of the word existing in the phrase having the least non-zero weight. This strategy is same as when dealing with a query having query words separated by ANDs, as described below. All *stop words* have a zero weight, and are not used in the scoring.

To determine the weight of a word, *w*, ending in a wildcard, the system assumes the presence of OR amongst all *qualified words*, of which *w* (excluding the wildcard character) is a prefix. The weight of *w* is taken to be the highest among the weights of all *qualified words*. This is in conformity with the rule applied when dealing with a query having all words separated by ORs.

To facilitate the calculation of score, we require idfs for query words present in the query string. For this purpose, we intend to use a *dynamic* dictionary as a module of the system, capable of providing the idf for any specified word. This module will modify the idfs of words dynamically as more and more documents come in. As we expect the idf value of a word to change very slowly, the impact of this information on the performance of index structure (discussed later in the paper) is relatively small. However, one could consider restructuring the index structure occasionally, so that it could be in accordance with the current state of the system. In addition, to facilitate wildcard matching, the dictionary module will be able to find all the words having a common given prefix, and optionally having some specified length. New words can be added to the dictionary along with a pre-specified idf value.

Having defined the basics, we define the scoring function, *sim*(*q*,*d*) for different kinds of *query strings*. For query string, *q*, consisting of query words separated by the ACCRUE operator, e.g. $q = k_1, k_2, \cdots k_n$

$$sim(q,d) = \frac{\sum_{i=1}^{n} w_i}{\sqrt{\sum_{i=1}^{n} w_i^2}}$$

Note that all $k_i, 1 \le i \le n$ are non-negated. The issue of *q* containing a word preceded by the NOT operator has been discussed before in the previous section. If a negated query word does not appear in the document, it does not have any effect on the final score. Negated terms are just used to make the query more selective.

For query string, *q*, consisting of query words separated by Boolean operators, e.g. $q = k_1\ OR\ (k_2\ \ AND\ NOT\ k_3)$. Before we discuss the scoring function for this kind of query string, here are some observations about the syntactic structure of *q*:

- A conjunction of one or more negated query words cannot appear as a disjunct, e.g. $q = k_1\ OR\ NOT\ k_2$ is not allowed to appear as a sub-expression in a query string, nor is $q' = k_1\ OR\ (NOT\ k_2 AND\ NOT\ k_3)$
- The weight of a negated query word is Boolean in nature, and is not a function of its idf. It is defined as 0, if the query word appears in the document, 1 otherwise.

Following the *Extended Boolean Model* and using the infinity norm [Baez99], the scoring function for a generalized disjunctive query string $q = k_1\ OR\ k_2 \cdots OR\ k_n$ is defined as,

$$sim(q_{or},d) = \frac{\max_{i=1}^{n}(w_i)}{\sqrt{\sum_{i=1}^{n} w_i^2}}$$

Similarly, for a generalized conjunctive query string $q = k_1\ AND\ k_2 \cdots AND\ k_n$, the scoring function is defined as,

$$sim(q_{and},d) = \frac{\min_{i=1}^{n}(w_i)}{\sqrt{\sum_{i=1}^{n} w_i^2}}$$

The processing of more generalized queries is done by grouping the operators in a predefined order (specified by precedence rules and/or parenthesization). We can apply a recursive method to calculate the score of a general query. For instance, consider

$q = k_1\ OR\ (k_2\ AND\ NOT\ k_3)\ OR\ (k_4, k_5, k_6)$. The score of this query string against a document d can be calculated as

$$sim(q,d) = \frac{\max(w_1, \min(w_2, w_3), (w_4 + w_5 + w_6))}{\sqrt{w_1^2 + w_2^2 + w_3^2 + w_4^2 + w_5^2 + w_6^2}}$$

In the above formula, the weight of k_3, w_3, is *Boolean* in nature as described previously.

Building the Predicate Index

In this section, we describe the steps in building the predicate index given a trigger definition. Figure 2 depicts the phase-wise transition from the trigger definition statement to the predicate index used for rule condition testing. The rule condition graph is an internal representation of the trigger definition as a graph.

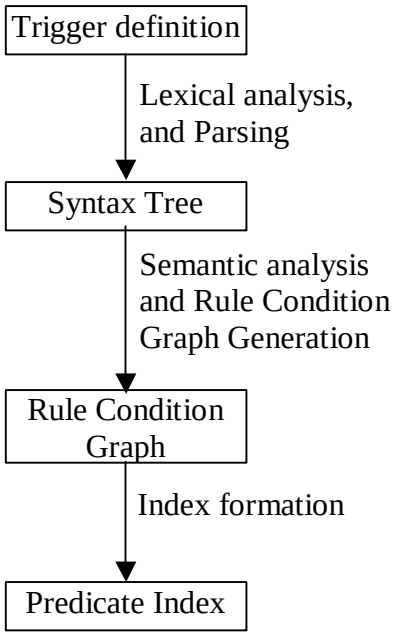

Figure 2. Flow diagram describing the compilation of a trigger definition.

As an example, consider again the following trigger definition:

```
create trigger JordanGolf
from video as v, viewed_by as b
after insert to b
when b.vid = v.vid
and v.category = “Sports”
and b.name = “Bob”
and tm_contains(b.comments, “Michael, Jordan, golf”, THRESH)
do …
```

The rule condition graph for the above trigger definition is given in Figure 3.

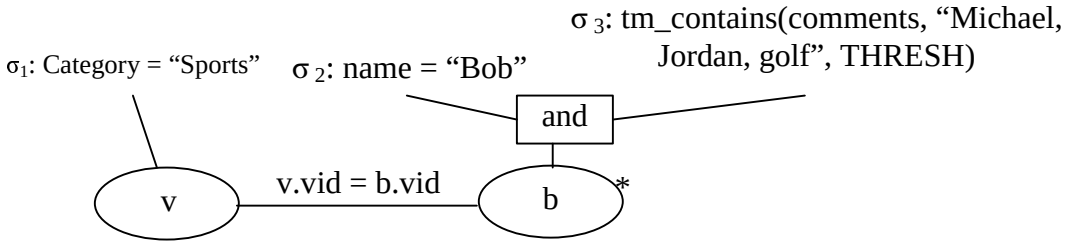

Figure 3. Rule Condition Graph.

Here v is the tuple variable representing relation video; b is the tuple variable representing relation viewed_by; the asterisk (*) on b denotes that b is the event tuple, and $\sigma_i, 1 \le i \le 3$ denotes a selection.

The selection predicates present in the rule condition graph are used in constituting the expression signature for the trigger. A detailed discussion on building the expression signature and indexing of predicates is described in an earlier paper [Hans99]. As said earlier, the free-text predicate (i.e. tm_contains operator) is also indexable. Next, we describe the extension of the Triggerman predicate index that allows the indexing of free-text predicates. This is made possible by introducing a new component, named the trie filter, to the predicate index.

Trie Filter–the Free-Text Predicate Index

In the course of this discussion, we will refer to a predicate containing a tm_contains operator as a free-text predicate. Every free-text predicate contains a *DocumentAttribute* field, specifying the column of the relation containing data in free-text format. This field can be used to build an expression signature for the predicate, e.g. ‘$tm\_contains(DocumentAttribute, CONSTANT)$’. All trigger definitions (or *profiles*) containing a free-text predicate having the same expression signature can be put under the corresponding node in the index structure [Hans99]. However, in case the profile contains more than one indexable predicate, choosing the one to use for indexing that results in the optimum performance is a challenging task. We describe more about this in the next section.

Earlier work on free-text predicate indexing has been reported in the work by Nelson [Nels98]. The technique we propose here has significant advantages over the technique described by Nelson. In particular, we avoid certain performance anomalies by indexing low-frequency words at the highest levels of our index structure.

As an example, consider a relation R(a, b, c), where a and b are the attributes containing free-text (e.g. of type VARCHAR, or CLOB) and c is of type integer. The following is a sample trigger definition that is to be indexed. As discussed in Appendix A, the tokens for AND, OR and NOT in a free-text query are & (ampersand), | (pipe), and ! (exclamation mark).

```
create trigger T1
after insert to R
when tm_contains(a, 'w1 & w2 & w3 | w4 & !w5', BOOLEAN)
        and b = 50
do …
```

This profile contains two predicates, having signatures '$tm\_contains(a, CONSTANT)$' and '$b = CONSTANT$' respectively, and can be indexed under either of them.

For efficient predicate matching, we propose a tree-like structure for the free-text predicate index, a variant of structures proposed by Yan et al. [Yan94], [Yan01]. This structure supports both Boolean and Vector Space free-text predicates (or sub-queries) in an integrated fashion. Before we describe this structure and indexing of free-text queries in detail, we need to classify the type of free-text sub-queries.

We define *operator type* to be one of AND, OR, or ACCRUE. A *basic* free-text sub-query (or predicate) is one containing operators of exactly one operator type. The literals in a basic free-text sub-query can be positive or negated query words. A negated query word is preceded by NOT. For example, assume a, b, and c are query words. The following are the examples of some valid basic free-text sub-queries:

$q_1 = a\ AND\ b\ AND\ NOT\ c$

$q_2 = a\ OR\ b\ OR\ c$

$q_3 = a, b, NOT\ c$

$q_4 = a$

A *composite* free-text sub-query (predicate) is composed of one or more basic free-text sub-queries. See appendix A for a context free grammar to generate valid basic and composite free-text sub-queries. The following is the example of a valid composite sub-query using the basic ones defined above:

$q = q_1\ AND\ (q_2\ OR\ q_3)$

The index structure that we propose can be used to index only basic sub-queries. A composite sub-query's result can be calculated using the results of the component basic sub-queries, which can be evaluated efficiently using the index. In the best case, a composite query is composed of a single basic sub-query, and hence is directly indexable. The system decomposes the composite free-text predicate into the *syntax tree* composed of basic sub-queries using the CFG described in Appendix A. These basic sub-queries are inserted into the index structure. The system also stores the syntax tree to facilitate the evaluation of the composite free-text predicate later.

In order to facilitate phrase matching, the free-text predicate is first preprocessed to a *normal form*. In this normal form, for every phrase that appears non-negated in the free-text predicate, we connect its words using an AND operator. The information regarding the identifiers of the words actually constituting a phrase is tagged, and is later used to annotate the basic sub-queries. For example, a free-text predicate, *p*, and its normal form, *p'*, are shown as below.

p = a AND “b c” AND NOT “d e”
p’ = [a AND b AND c AND NOT “d e”]$^{\{\text{phrase list} - (2,3)\}}$

Here 2 and 3 denote the ids for b and c, respectively. The list containing 2 and 3 denotes that b and c constitute a phrase.

The index structure consists of two kinds of nodes, *branch* nodes and *predicate* nodes. A branch node consists of a word, any number of branch nodes as children, and possibly a list of predicate nodes. When searching, if the word in the branch node is found (i.e., a match) in the document, we continue searching for words in the child branch nodes. In addition, a match at a branch node suggests a probable success for a predicate node. Each predicate node corresponds to one basic sub-query. A predicate node contains:

- A list of negated query words,
- A phrase identifier list containing the ids of words that are actually part of a phrase (identified while decomposing the normal form to basic sub-queries), and
- Possibly the most insignificant sub-vector in case the basic sub-query contains one (this is described in more detail in the subsequent section on optimizing the trie filter).

The root node is a branch node containing a *NULL* word and an empty list of predicate nodes. It always matches any document.

Assume *p* is the free-text predicate to be indexed for the profile P. First, we preprocess *p* into its normal form *p’*. Assume *p’* is composed of *m* basic predicates $p_1, p_2, \cdots, p_m$. The following are the different cases to consider for indexing $p_j, 1 \le j \le m$.

Case 1: Assume that $p_j, 1 \le j \le m$ consists of k_j words, $w_1, w_2, \ldots, w_{k_j}$ connected by ANDs All of these words are non-negated. Furthermore, assume the words are arranged in the non-increasing order of their idfs. We organize these words into a tree as follows. By convention, we say that the root node of the index structure is at level 0. We create (if not already existent) a branch node at each level $i, 1 \le i \le k_j$ corresponding to w_i. The branch node at level $i, 1 \le i \le k_j$ is made a child of the branch node at level $i-1$. The branch node at level *k* contains a predicate node designating p_j in its list of profile nodes. In case p_j contains any negated words, they are kept in the predicate node separately in the list of negated terms. Figure 4 shows the structure for some sample basic predicates given below. Only relevant information is shown in nodes.

Sample predicates:

p1 = a AND b AND NOT c
p2 = a AND d
p3* = a AND d AND e
p4 = b AND f AND NOT g
p5 = a AND b

* words d and e are actually part of a phrase.

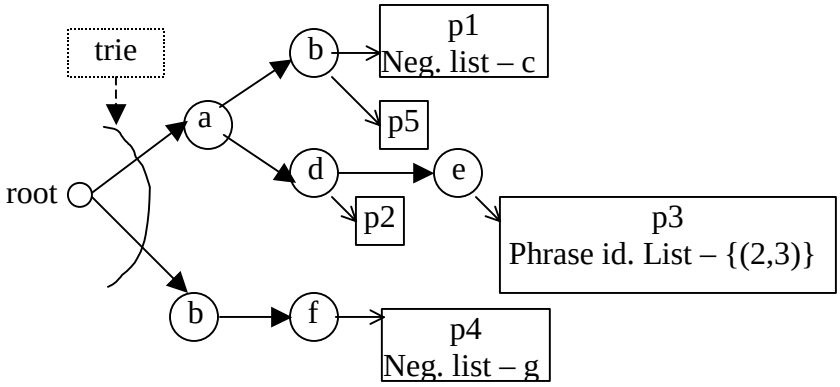

Figure 4. A trie filter structure.

Here the numbers 2 and 3 in the phrase id list identify d and e respectively in the predicate p3.

To make searches more efficient, we can build a trie based on characters at each level of the structure corresponding to the children of a branch node. Keeping in mind that the cost in terms of memory can significantly outweigh the performance gain, we use lists of words rather than a trie structure when the number of children of a branch node is small. We switch back and forth dynamically between these two schemes as profiles are added or removed. When the number of child branch nodes goes above a certain threshold *Thresh1*, we convert the list to a trie; when the number does below *Thresh2*, we convert the tree to a list, where $Thresh2 << Thresh1$.

Case 2: In case $p_j, 1 \le j \le m$ consists of *k* query words, $w_1, w_2, \ldots, w_k$ connected by ORs, we create (if not already existent) a branch node at level 1 corresponding to every $w_i, 1 \le i \le k$ and put a predicate node designated p_j under all these branch nodes. In this case, p_j cannot contain any negated term. Figure 5 shows the structure for some sample basic predicates given below.

Sample Predicates:

p1 = a OR b

p2 = a OR c

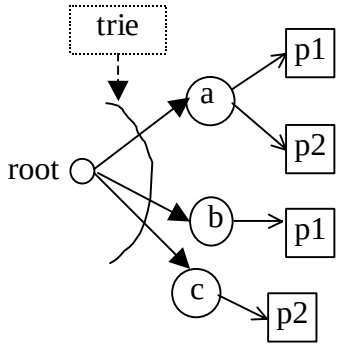

Figure 5. A trie filter structure for sample OR predicates.

Case 3: In case $p_j, 1 \le j \le m$ consists of k query words, $w_1, w_2, \ldots, w_k$ connected by ACCRUEs, the method to index is similar to that of case 2. Note that in this case, p_j can contain negated terms. These terms are kept in the predicate nodes.

Algorithm MatchDocument {

1 – For the document attribute (hereafter referred to as document) of the event tuple, search the trie filter to find the set of basic sub-queries that match against the document. Assume the set $R = b_1', b_2', \cdots, b_k'$, where $b_i', 1 \le i \le k$ are the sub-queries that match. See Function FindMatchingBasicPreds below for a detailed discussion of this step.

2 - Let $P(b)$ be the free-text predicate containing b as a basic sub-query. Form the set

$$PS = \bigcup_{i=1}^{k} \{P(b_i')\}.$$

3 - for each $p \in PS$ {

Evaluate *p*, which is a composite free-text predicate, based on the values determined for all the basic predicates it contains which can be found in R. If *p* matches the document, then continue to evaluate restOfPredicate (if any) [Hans99] for the profile indexed under *p*.

}

}

Figure 6. Algorithm MatchDocument.

Optimizing the trie filter – Selective query indexing

In case the composite free-text predicate consists of just one basic sub-query, and belongs to the Vector Space model (i.e. contains a positive threshold value), we can optimize the trie filter in cases 2 and 3 stated above. The idea is to selectively index the basic sub-query under a particular subset of all the query words present in it, called the *significant subset*.

For case 2 (OR), the significant subset is composed of the query words that have idf at least equal to the threshold value. All other words can be neglected as their presence or absence does not affect the decision whether or not the sub-query is successful. The sub-query is indexed under all the words in significant subset, similar to case 2, except the insignificant words are excluded.

For case 3 (ACCRUE), we find the *most insignificant sub-vector* of the basic sub-query using the method specified by Yan et al. [Yan01]. The significant subset consists of all the query words

except those present in most insignificant sub-vector. The sub-query is indexed under all and only the words in the significant subset. The most insignificant sub-vector is stored in the predicate node for later use.

Free-text Matching Algorithm

Having defined the free-text index structure, we now describe the algorithm to match the document against existing profiles using the trie filter. The algorithm *MatchDocument* is presented in Figure 6.

There is a point to ponder before we describe the algorithm to find matching basic sub-queries. In case there are more children of a branch node than words in the document itself (which is a common case for a system containing a lot of profiles having an indexed free-text predicate for this document), it would be efficient if we take the words in the document and search them against the trie filter. In the other case, when the number of branch nodes to be searched is small, the elements of these nodes should be searched in the document. To facilitate this search operation, we propose building a high order trie structure [Hor95] over the document. Let this be called the *document trie*. The element node corresponding to a word in the document trie is annotated with the length information of the word and a list holding the start and end locations for that word in the document. The start and end locations of words facilitate phrase matching. Note that a phrase match occurs when the words in the phrase appear in the same order in the document as they appear in the phrase. These words may be separated by zero or more words in the document. We also need to be able to enumerate all the words with a given prefix to gather the location information for a word containing a wild card. This preprocessing to build the document trie is only done once when we have to match the document against a trie filter.

In Figure 7, we describe the algorithm FindMatchingBasicPreds, presented in pseudo-code. This algorithm makes use of a function FindMatchingBasicPredsHelper shown in Figure 8. FindMatchingBasicPredsHelper takes a pointer to a branch node and a list of positions as arguments. The list of positions contains information about the position of words appearing in the branch node and its ancestor branch nodes.

```
Algorithm FindMatchingBasicPreds (BranchNode trieFilterRoot,
     Document Doc,:INPUT, Set MatchingBasicPreds:OUTPUT)
{
     Set MatchingBasicPreds to NULL
     FindMatchingBasicPredsHelper (trieFilterRoot, NULL, MatchingBasicPreds)
     for each p in MatchingBasicPreds {
          if Doc contains a word in p's negated word list
               remove p from MatchingBasicPreds
               continue
          if p contains an insignificant sub-vector
               modify score of p
     }
}
```

Figure 7. Algorithm FindMatchingBasicPreds.

```
Function FindMatchingBasicPredsHelper (BranchNode theBranchNode,
        List ListOfWordPositions:INPUT, Set MatchingBasicPreds:INOUT)
{
    for each p in theBranchNode's list that contains predicate nodes
    {
        if (p's phrase identifier list is not NULL)
            check for phrase matches using p's phrase identifier list and ListOfWordPositions. If
            successful, add p to MatchingBasicPreds.
        else
            add p to MatchingBasicPreds.
        Calculate p's score.
    }
    Find all child branch nodes of theBranchNode that appear in the document. Mark these
    nodes successful.
    for each child branch node bChild of theBranchNode marked successful
    {
        List' = ListOfWordPositions appended with the location information of the word in bChild
        FindMatchingBasicPredsHelper (bChild, List', MatchingBasicPreds)
    }
}
```

Figure 8. Function FindMatchingBasicPredsHelper.

Selecting an Optimal Predicate Clause for Indexing

In this section, we describe the approach to select the appropriate predicate for indexing, in case there exist more than one indexable predicate. The approach is based on finding the predicate with optimal selectivity factor and search cost. While these factors have been studied in detail for conventional predicates involving relational operators like $\leq, =, >$ etc. [Sel79]; little prior work is available addressing the issue for free-text predicates such as tm_contains().

To analyze the predicates, first we convert the complete selection predicate (the when clause in the trigger definition) into CNF. Before we present the cost-selectivity model to pick a proper conjunct on which to index, we describe how to estimate the selectivity and cost metrics associated with a free-text predicate in the next section.

Selectivity Estimation of Free-Text Predicates

In an information retrieval (IR) scenario, selectivity for a free-text predicate is defined as the fraction of the total number of documents that will be retrieved when searched for the query containing just this predicate. In an alerting scenario, it is better described as the probability of a document successfully matching a profile containing just this predicate. This probability can be estimated using the definition for an IR system, by querying the data stored to find out the fraction of documents that match. A simple solution to this problem would be to store the words along with a list of *document ids* of all the documents in which it appeared. These lists of document ids associated with words can then be used to find out the selectivity of the whole

predicate. This solution is clearly inefficient in terms of storage space. In addition, finding conjunction and disjunction of long lists containing ids is a costly operation. In [Chen00a], some approximation techniques are suggested to estimate the selectivity of Boolean queries. The idea is to use a group of hash functions to compute the set resemblance among the sets corresponding to the search terms. For a sufficiently large number of independent hash functions, the approximation turns out to be good enough in practice. As the decision whether to use one predicate to index the profile or the other is not crucial from a correctness point of view, we can use this scheme to estimate the selectivity of a free-text predicate when it is Boolean in nature (contains just the Boolean operators). We do not take into account the threshold value while deciding the selectivity. The reason is that we have to match the document and compute a score anyway before it can be checked against the threshold.

Although the technique described in [Chen00a] is only applicable to queries containing the Boolean operators, an estimation of the selectivity of a query containing the vector space operator (ACCRUE) can be done as follows. We translate all ACCRUE operators into OR operators and eliminate words appearing in the insignificant sub-vector. Selectivity of this query can be estimated as stated above.

Cost Estimation of Free-Text Predicates

We associate two types of costs with a free-text predicate. In case there exists a trie filter structure on free-text predicates, the indexed search has a cost Cost_of_index_search associated with it. This cost is amortized over the search cost for a number of predicates using the trie structure. A brute force search without using the trie filter structure has a cost Cost_of_brute_force_search. In both the cases, the search for query words in the document is performed with the help of a trie structure built over the document. As this trie structure is prepared in the preprocessing step on the arrival of the event tuple, the cost estimation formulas do not account for the cost associated with building the trie over the document. The preprocessing time for an event tuple increases by an amount of $O(\sum_{i=1}^{l}|D_i|)$, where $|D_i|$ is the size of *i*th text field in the event tuple, and *l* is the total number of text fields in the event tuple. Given below are the formulas associated with indexed and brute force search cost for a predicate containing *k* words and a trie filter containing *m* words and *n* predicates. In the worst case, all the words in the predicate or all the words in the trie filter need to be searched.

$$Cost\_of\_index\_search = \frac{c_1 \sum_{i=1}^{m} |w_i|}{n}$$

$$Cost\_of\_brute\_force\_search = c_2 \sum_{i=1}^{k} |w_i|$$

Algorithm FindPredicateToIndex
{
    if there does not exist any indexable term
        return NULL.
        Let P_1', …, P_m' be the indexable terms in Pred.
        if m = 1
            return P_1'
        else
        {
            Let sel(P) be the selectivity of P, Cost_of_index_search(P) be the cost of doing an index search through the index built for predicates in the equivalence class of P, and Cost_of_brute_force_search(P) be the cost of doing a brute force search for P without using any index structure. For an indexable term P_j', let P_1'', …, P_k'' be the remaining terms in Pred.

            For every indexable term P_j', $1 \leq j \leq m$, compute the rank of P_j' given by the following formula, a variation of the formula proposed by Hellerstein [Hel94]:

            // Cost_brute_force_predArray is defined later

$$rank(P_j') = \frac{sel(P_j') - 1}{\left(\begin{array}{l} Cost\_of\_index\_search(P_j') \\ \quad + sel(P_j') * Cost\_brute\_force\_predArray(P_1'', \cdots, P_k'') \end{array} \right)}$$

            return P_j' with the least rank value.
        }
}

Function Cost_brute_force_predArray (Array predArray[1…k]:INPUT)
{
    for i = 1 to k compute

$$rank(predArray[i]) = \frac{sel(predArray[i]) - 1}{Cost\_of\_brute\_force\_search(predArray[i])}$$

        sort predArray in non-decreasing order of rank.
    return Cost_brute_force_helper(predArray,k)
}

Function Cost_brute_force_helper (Array predArray[1…k], integer k:INPUT)
{
    Cost = 0
    ProbabilityTestIsRequired = 1
    for i = 1 to k begin
        Cost = Cost + ProbabilityTestIsRequired * Cost_brute_force_search(predArray[i])
        ProbabilityTestIsRequired = ProbabilityTestIsRequired * sel(predArray[i])
    end
    return Cost
}

Figure 9 – Algorithm FindPredicateToIndex and associated routines.

Here $|D|$ is the average size of a document, $|w_i|$ is the length of i'th word in the trie filter, and c_1, c_2, c_3 are constants. These constants are determined empirically.

Index Predicate Selection Based on Per-Predicate Cost and Predicate Selectivity

In this section, we describe a heuristic to choose an appropriate predicate for indexing the profile. If the profile involves a single relation, the profile will be indexed (if there is any indexable predicate in it) under the node corresponding to that relation in the TriggerMan predicate index. In case it involves more than one relation, it may be indexed under various nodes corresponding to different relations, depending on whether the WHEN clause contains any indexable predicate involving that relation. This rule is overruled if there is an AFTER clause present in the trigger definition (declaring the presence of an event tuple variable). In that case, the profile is only indexed under the relation associated with the event tuple variable. In the following discussion, we will focus on only a part of the WHEN clause involving the relation under which we want to index the profile. This strategy can be applied one by one to all the relations that appear in a profile.

For example, let us consider a part of the WHEN clause, Pred, appearing in a profile that involves the relation R. As allowed by the TriggerMan command syntax, Pred can be written in a conjunctive form as follows:

$$\text{Pred} = P_1 \text{ and } P_2 \text{ and } \ldots P_N$$

Here, a term P_i is either a free-text predicate (or term) or an ordinary database predicate. As discussed above, a free-text predicate is indexable in the TriggerMan system. Similarly, a predicate of the form R.column = CONSTANT is also indexable. However, the approach described next can accommodate any indexable predicate in general.

The heuristic to find a suitable predicate on which to index the profile is given in Figure 9 as the algorithm FindPredicateToIndex in pseudo-code. The figure also contains associated functions that help FindPredicateToIndex do its job.

A profile will always be indexed if it contains at least one indexable predicate. In case there does not exist even one indexable predicate, an expression signature for Pred taken as a whole is generated and stored in the system. For more on creating signature expressions and storing constant placeholder values, see the work by Hanson et al. [Hans99].

The predicate that is returned by the algorithm FindPredicateToIndex (if not null) is then used to index the profile. We have already described how to index using free-text predicate in the section Trie Filter–the Free-Text Predicate Index. For other types of indexable predicates, we use the strategies previously described by Hanson et al. [Hans99].

Summary

In this paper, we present an extension to TriggerMan to enhance its capabilities to alert on free-text data. The design of a new operator, tm_contains(), to deal with free-text attributes is presented. The scoring function is designed to rank-score the incoming document against the free-text predicate to allow the filtering of irrelevant documents for a particular user. The

predicate index structure in TriggerMan is extended to index on free-text predicates along with other predicates supported for indexing. We also present heuristic strategies to choose an optimal predicate for indexing the profile. As future work, we propose the introduction of new operators for free-text predicates and extension of TriggerMan's predicate index to handle different types of indexable predicates.

References

[Alti00] Altinel, M. and M. J. Franklin, Efficient Filtering of XML Documents for Selective Dissemination of Information, *The VLDB Journal*, 2000, pp 53-64.

[Baez99] Baeza-Yates, R. et al, *Modern Information Retrieval*, Addison Wiley, 1999.

[Chen00] Chen, J., D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for Internet databases, *In Proceedings of the ACM SIGMOD Conference on Management of Data*, 2000.

[Chen00a] Chen, Z. et al., Selectivity Estimation for Boolean Queries, *In Proceedings of the ACM Symposium on Principles of Database Systems*, 2000.

[Exca01] User's Guide, Excalibur Text Search DataBlade Module, Informix Coorporation, Available at www.informix.com/answers/english/docs/datablade/5401.pdf.

[Info01] *Informix Guide to SQL: Reference*, Informix Software, 2001.

[Hans99] Hanson, E., Scalable Trigger Processing, *Proceedings of the IEEE Data Engineering Conference*, 1999.

[Hel94] Hellerstein, J. M. Practical Predicate Placement. *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp. 1994, 325-335.

[Hor95] Horowitz, E., Sahni, S., Mehta, D., *Fundamentals of Data Structures in C++*, Text, Computer Science Press, 1995.

[Liu99] Liu, L., Pu, C., and Tang, W., Continual Queries for Internet-Scale Event-Driven Information Delivery, *IEEE Transactions on Knowledge and Data Engineering*, 1999.

[Nels98] Nelson, P. C., Evaluation of Content of a Data Set using Multiple And/Or Complex Queries, *United States Patent* Number 5,778,364, July 7, 1998.

[Pugh90] Pugh, W., Skip Lists: A Probabilistic Alternative to Balanced Trees, *Communications of the ACM, Vol. 33, Issue 6, 1990, pp. 668-676*.

[Sel79] Selinger, P. G., Astrahan, M., Chamberlin, D., Lorie, R., and Price, T., Access Path Selection in a Relational Database Management System, *Proceedings of the ACM SIGMOD International Conference on the Management of Data, 1979, pp. 22-34.*

[Veri01] User's Guide, Verity Text Search DataBlade Module, Informix Coorporation, Available at www.informix.com/answers/english/docs/datablade/8245.pdf.

[Yan94] Yan, T. W. and H. Garcia-Molina, Index Structures for Selective Dissemination of Information under the Boolean Model, *ACM Transactions on Database Systems, Vol. 19, Issue 2, 1994, pp. 332-364.*

[Yan01] Yan, T. W. and H. Garcia-Molina, The SIFT Information Dissemination System, *ACM Transactions on Database Systems*, to appear.

Appendix A—Free-text Query Syntax

A syntactically legal query expression E that may appear as the second argument of tm_contains() is defined by the following grammar.

Grammar:

E → E OR T | T
E → E ACCRUE T | T
T → T AND F | F
F → (E)
F → QueryWord
F → NOT QueryWord
QueryWord → Word | Phrase

Tokens:

AND → '&'
OR → '|'
NOT → '!'
ACCRUE → ',' | whitespace between two words

We define an alphabet *S* that contains numeric digits, letters, and underscore. A Word is any sequence of characters from *S*. A word may optionally be enclosed in double quotes. A word may end in the wildcard *, which matches zero or more characters.

A Phrase is any sequence of words separated by white space and enclosed in double quotes.