INTEGRATED ALERTING FOR STRUCTURED AND FREE-TEXT DATA IN TRIGGERMAN

By

HIMANSHU RAJ

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2001

Dedicated to my parents, who taught me a valuable quality, perseverance

ACKNOWLEDGMENTS

I am thankful to Dr. Eric N. Hanson, my mentor at UF, without the guidance of whom this work would not have been possible. Many thanks go to the database center students and especially to Mrs. Sharon Grant, who always helped me at any time. Last, but not least, I am thankful to my friends, who made my stay at Gainesville blissful and helped me achieve my goals.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

INTEGRATED ALERTING FOR STRUCTURED AND FREE-TEXT DATA IN TRIGGERMAN

By

Himanshu Raj

December 2001

Chairman: Dr. Eric N. Hanson
Major Department: Computer and Information Science and Engineering

TriggerMan is a highly scalable, asynchronous database triggering and alerting system. In this work, 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 user-defined routines for text matching 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.

CHAPTER 1
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 [Inf01d] and Excalibur [Inf01c]. 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 [Han99], 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 [Che00a, 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 and free-text predicates is a challenge. This document 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,

```
tm_contains(document, query [, additional parameters])
```

For example, consider this table schema:

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

We 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.

By including tm_contains() as an operator that can be used in a TriggerMan 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 and Franklin [Alt00]. 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 document, 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. We also present implementation details and performance results for a prototype implementation of TriggerMan equipped with free text alerting along with alerting on structured data.

CHAPTER 2
ARCHITECTURE

TriggerMan [Han99] is a highly scalable asynchronous trigger processing system. It uses main memory index structures for fast and efficient processing of numerous triggers defined in a single database. The current version of TriggerMan is implemented as an extension module (Datablade) for Informix Dynamic Server 2000 with Universal Data Option, an object relational DBMS [Inf01b]. This chapter describes the architectural enhancements made to the TriggerMan system by including trigger condition testing for free-text data.

Architecture of the enhanced TriggerMan System

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 [Han99]. 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 [Pug90] 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 document.

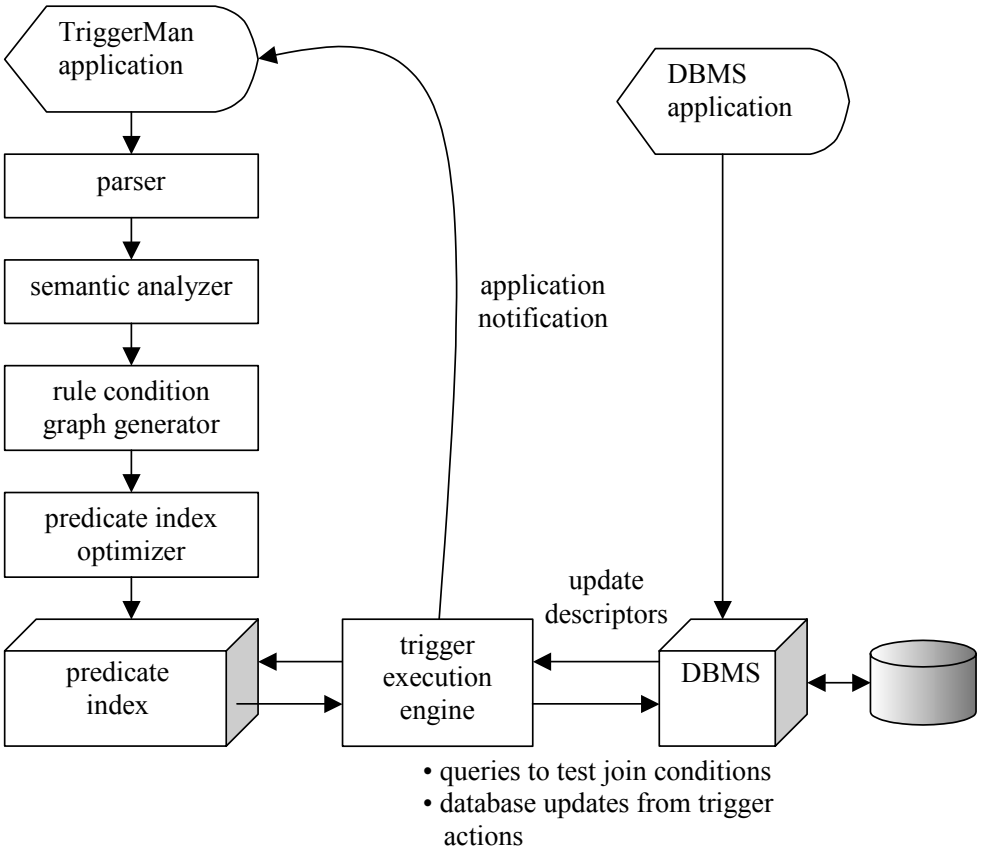

Figure 1: Architecture of TriggerMan enhanced with free-text alerting.

The tm_contains() Function for Free-Text Matching

General Syntax

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

```
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' [Inf01a].

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 ('*') at its right end. For example, "david," david, dav*, and "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 [Por80] is not implemented in the preliminary version of tm_contains; however it can be simulated to an extent by using wildcards.

Supported Free-Text Operators

The following operators are supported by tm_contains() to facilitate free-text matching (character in brackets denote the most commonly used character symbols for these operators)

ACCRUE (',')

This is perhaps the most commonly used operator while performing free-text queries using the vector space model [Bae99, Kol98] (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 chapter). When 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 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 document.

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 greater than 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.

Semantic Issues with tm_contains() Function

The tm_contains operator uses an extended vector space model for specifying the query strings. There are several semantic issues associated with the model. First, the use of Boolean operators (AND/OR/NOT) is meant to provide the matching as implied by the Boolean model [Bae99], 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 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. If any of the negated terms appear, the score will be poisoned to zero, and the search will terminate unsuccessfully. Only when none of the negated terms appear will the match be declared 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 users 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 …
```

In addition to the scheme described above for choosing the threshold value, we can also use the following scheme, if the TriggerMan system is running as a DataBlade for an object relational DBMS and if it provides a user-defined routine tm_score() for use in queries. 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().

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

1. 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 (the system can produce a combined score for all the predicates [Fag96]; however drawing a conclusive line for more than one threshold values using this combined score is not effective in general) . For example, the following is illegal:

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

2. 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 [Han99]. 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.

CHAPTER 3
CALCULATION OF SCORES

This chapter describes the methodology used for computing the score of a query against a document. This score value is compared against the *threshold* provided in the tm_contains function, which determines the success or failure of a match between a free-text predicate and the document. Before we state the scoring function, *sim(q,d)*, specifying the similarity between a query *q* and a document *d*, we need to define a few terms.

Definitions

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

$$w_{i,d} = f_i * idf_i$$

As we assume that a query word is not repeated in the query string, the weight associated between the query word and the query string, $w_{i,q}$, is the idf of the word, i.e.,

$$w_{i,q} = idf_i$$

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 the same as when dealing with a query having query words separated by ANDs, as described below. All *stop words* (high frequency words like 'the' etc.) 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.

The Scoring Function

Having defined the basics, we define the scoring function, *sim*(*q,d*) for different kinds of *query strings*. First we discuss the scoring formulas for query strings consisting of only one of the operators, ACCRUE, AND or OR. We characterize these kinds of query strings as *basic queries*.

Scoring Function for Basic Queries

For query string, q, consisting of query words separated by the ACCRUE operator, e.g., $q = k_1, k_2, \cdots k_n$

$$sim(q_{accrue}, d) = \frac{\sum_{i=1}^{n} w_{i,d} * w_{i,q}}{\sqrt{\sum_{i=1}^{n} w_{i,d}^2} \sqrt{\sum_{i=1}^{n} w_{i,q}^2}}$$

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 previously. 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.

Consider a 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 [Bae99], the scoring function for a generalized disjunctive query string $q = k_1\ OR\ k_2 \cdots OR\ k_n$ is defined as

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

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

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

These formulas have been modified to maintain the uniformity in a hybrid model such as ours, where both Vector Space and Boolean model can be used inside a generalized query string.

Scoring Function for General Queries

The processing of more generalized queries, characterized as composite queries, is done by grouping the operators in a predefined order (specified by precedence rules and/or parenthesization). First we rewrite the generalized query string as composed of

maximal basic queries (a maximal basic query is a maximal substring of the composite query that can be characterized as a basic query). For instance, consider the composite query string $q = k_1\, OR\, (k_2\ AND\, NOT\, k_3)\, OR\, (k_4, k_5, k_6)$. It can be rewritten as $q = q_1\, OR\, q_2 OR\, q_3$, where each of $q_1, q_2,$ and q_3 is a basic query. Next, we form the syntax tree (a binary tree) over the rewritten composite query and apply a recursive method to calculate the score. Score values for the leaf nodes of this tree can be computed using the scoring strategies discussed above. For any non-leaf node in the tree (which would be one of the operators AND, OR or ACCRUE), the score is defined on the basis of the operator present in that node.

For a node containing AND, the score is defined as

$$Score_{parent} = \min(Score_{child1}, Score_{child2}).$$

For a node containing OR, the score is defined as

$$Score_{parent} = \max(Score_{child1}, Score_{child2}).$$

For a node containing ACCRUE, the score is defined as

$$Score_{parent} = Score_{child1} + Score_{child2}.$$

The score of the composite query string, *q*, against the document *d* is given as $sim(q, d) = Score_{root}$, where root identifies the root of the syntax tree formed above.

CHAPTER 4
BUILDING THE PREDICATE INDEX

In this chapter, 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 condition definition as a graph.

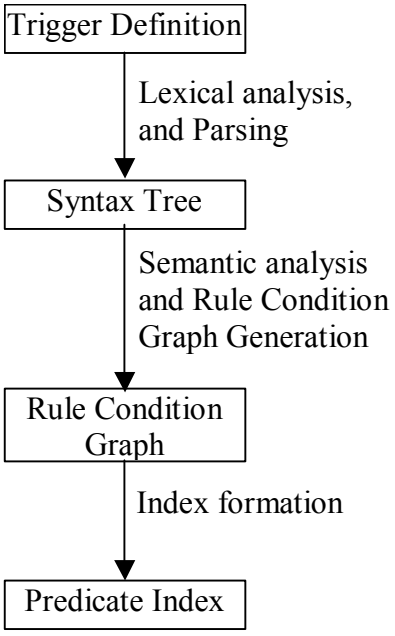

Figure 2: Flow diagram describing the compilation of a trigger definition.

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.

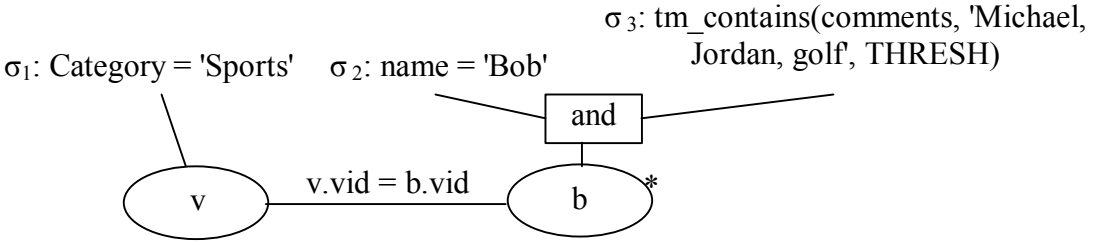

Figure 3: Rule Condition Graph.

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 [Han99]. 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 [Han99]. 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 chapter.

Earlier work on free-text predicate indexing has been reported in the work by Nelson [Nel98]. 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
from R
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 and Garcia-Molina [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 – Indexing the AND type Basic Predicate

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

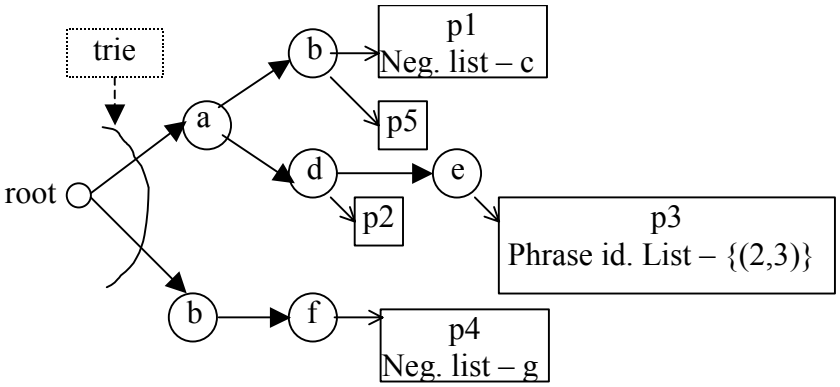

Figure 4: A trie filter structure.

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 build a trie based on characters at each level of the structure corresponding to the children of a branch node. We also build a list of words by chaining the leaf nodes of this trie structure. Building this kind of structure helps choose the matching strategy against the document dynamically. This issue is discussed in detail in the algorithm for document matching described in the next chapter.

Case 2 – Indexing the OR type Basic Predicate

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

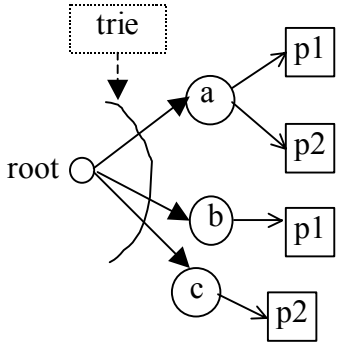

Figure 5: A trie filter structure for sample OR predicates.

Case 3 – Indexing the ACCRUE type Basic Predicate

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.

---

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

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 rather than the keyword BOOLEAN), 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 and Garcia-Molina [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.

CHAPTER 5
FREE-TEXT MATCHING ALGORITHM

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

```
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, ..., b'_k, where b'_i, 1 ≤ i ≤ 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 = ⋃_{i=1}^{k} {P(b'_i)}.

3 - for each p ∈ 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)
     [Han99] for the profile indexed under p.
     }

}
```

Figure 6: Algorithm MatchDocument.

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 [Knu73] 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 the first time it needs to be searched.

In Figure 7, we describe the algorithm FindMatchingBasicPreds, presented in pseudo-code. This algorithm makes use of a function FindMatchingBasicPredsHelper described 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,

     Set MatchingBasicPreds)

{

     1 - Set MatchingBasicPreds to NULL

     2 - FindMatchingBasicPredsHelper (trieFilterRoot,
          NULL, MatchingBasicPreds)

     3 - 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 by using the weights
               of the words in the insignificant sub-
               vector [Yan01]

     }

}
```

Figure 7: Algorithm FindMatchingBasicPreds.

```
Function FindMatchingBasicPredsHelper

   (BranchNode theBranchNode, List ListOfWordPositions,

      Set MatchingBasicPreds)

{

     1 - 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.

               }

          }

     2 - Find all child branch nodes of theBranchNode
          that appear in the document. Mark these nodes
          successful. Probe the children into the
          document trie if number of children is less
          than the number of words in the document;
          probe document words into the trie of
          children otherwise.

     3 - 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.

CHAPTER 6
OPTIMAL PREDICATE CLAUSE SELECTION

A trigger condition may contain more than one predicate in its WHEN clause. The key to faster condition testing is that the trigger condition should be indexed on the predicate that is more selective and incurs less search cost relative to other predicates. In this chapter, 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. We combine these two factors into a single value, rank, which can then be used to decide which predicate should be chosen for indexing the trigger condition. These factors have been studied in detail for conventional predicates involving relational operators like $\leq, =, >$ etc. [Sel79]. However, 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. Then we apply the algorithm (discussed in appendix B) to find the predicate to index on the complete selection predicate. This algorithm analyzes all indexable predicates (if any), and returns the optimal one (returns NULL if there exists no indexable predicate). We can then use this predicate to index the trigger condition. Before we present the cost-selectivity model to pick a proper predicate 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 each word along with a list of document ids of all the documents in which that word 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 [Che00b, Jag99], 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 [Che00b] 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 ith text field in the event tuple, and l is the total number of text fields in the event tuple. Cost formulas associated with indexed and brute force search cost for a predicate containing k words and a trie filter containing n indexed predicates (the total number of query words present in the trie filter is m) are discussed subsequently.

The cost of indexed search is a per predicate cost and is amortized over all the predicates indexed in the trie filter. While matching the document against the trie filter structure, all words in the trie filter need to be probed into the document trie in the worst case (this serves as an upper bound even if we are decide dynamically whether to probe

the document using the words into the trie filter, or vice versa; see details of matching algorithm in previous chapter). The formula for finding the cost of index search for a predicate is given below. Here $|w_i|$ is the length of i'th word in the trie filter, and c_1 is a constant, determined empirically.

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

The above formula provides with a very pessimistic upper bound on the cost of index search for a predicate, since it assumes that all the words for all the predicates present in the predicate index will be searched. In reality, the cost totally depends on the selectivity of predicates present in the predicate index and the document being matched against the predicate index. We propose the improvement upon this formula as an important area for future work.

The cost of brute force search for a predicate is the cost of matching a document against the predicate in the case when the predicate is not indexed. In this case, we need to probe every word in the predicate into the document trie in the worst case. Hence the cost of brute force search for a predicate is bounded above by the formula given next. Here $|w_i|$ is the length of i'th word in the predicate, and c_2 is a constant, determined empirically.

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

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 normal form as follows:

Pred = P_1 and P_2 and …P_N

Here, a term P_i is either a free-text predicate (or term), or an ordinary database predicate, or a conjunct containing these types of predicates (connected by OR(s)). 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. In case P_i is a conjunct that contains one or more OR operators, it is unindexable in the TriggerMan system. However, the approach described next to find the indexable predicates can accommodate any indexable predicate in general.

The heuristic to find a suitable predicate on which to index the profile is given in Appendix B as the algorithm FindPredicateToIndex in pseudo-code. The appendix 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. [Han99].

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 in chapter 4. For other types of indexable predicates, we use the strategies previously described by Hanson et al. [Han99].

CHAPTER 7
FREE-TEXT ALERTING IN TRIGGERMAN - IMPLEMENTATION DETAILS

Introduction

This chapter describes the enhancements made to the TriggerMan system to enable it to do free-text alerting along with alerting on structured data, such as integers, floats, etc. In this work, a *Selective Dissemination of Information* (SDI) system was built, and integrated seamlessly with the existing TriggerMan system, so that users can define alerts on table attributes that contain free-text information in them, such as of type CHAR, VARCHAR, LVARCHAR, TEXT etc. Due to TriggerMan's clean and modular structure, this task, while quite large, did not prove to be nightmarish.

Due to time and resource constraints, the prototype system implemented includes some of the features described in previous sections to a partial extent. In the sections to follow, a brief description of various modules that are implemented afresh or were modified is presented. See Figure 9 for a description of various modules and the interactions among them.

Parser and Semantic Analyzer for Command Language Extensions

As discussed before, the command language syntax of TriggerMan is enhanced with a new operator, *tm_contains* (see chapter 2 for details). This operator features a free-text predicate as its argument, written to conform with the grammar presented in the appendix. The free-text predicate language supported by the system is rich in features, and provides the integration of both Boolean model and vector space model for free-text

queries [Bae99]. A separate module is implemented for lexical analysis, parsing and semantic analysis of free-text predicates. This module is integrated with the other modules, so that a call to *tm_contains* is parsed as a function call, and an appropriate syntax tree is built for the CREATE TRIGGER command statement. The syntax tree is then semantically analyzed, and is passed to the trigger creation module.

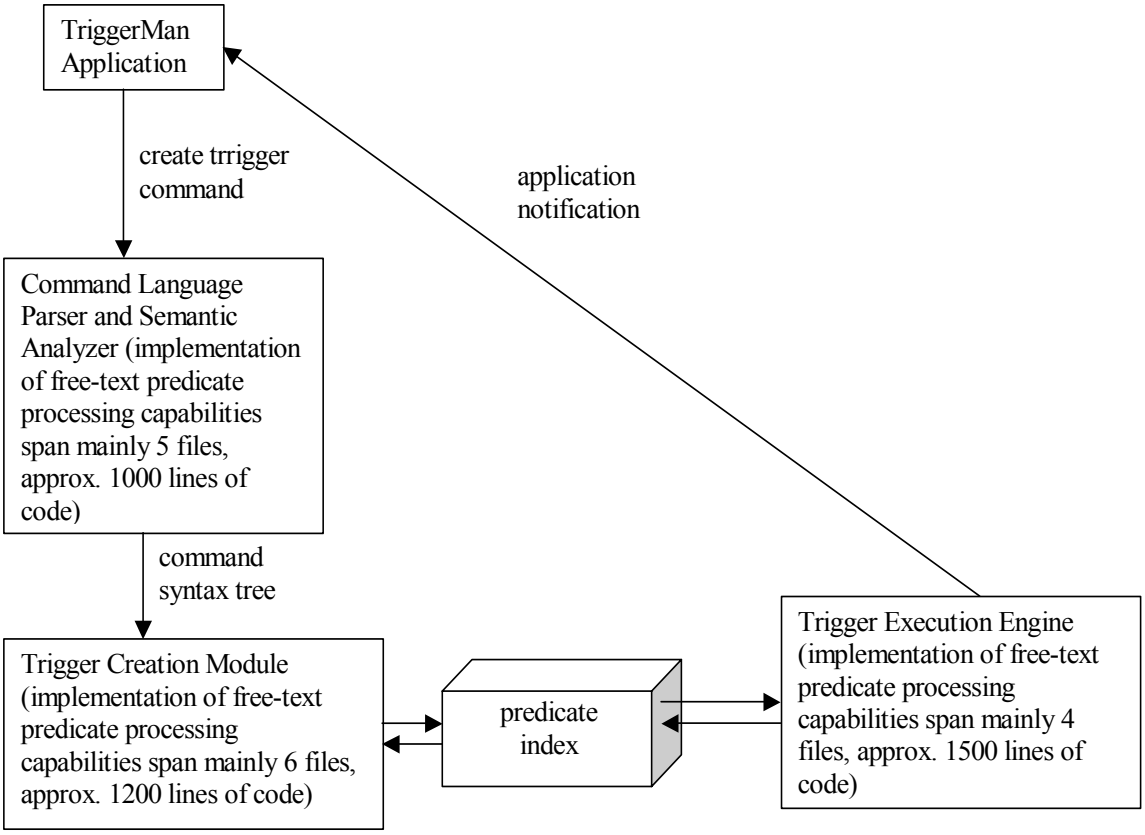

Figure 9: Modules describing extension to the original TriggerMan system.

Trigger Creation Module

The trigger creation module

- Finds indexible predicate(s), if any are present in the trigger definition.

- Creates an expression signature for every tuple variable in the trigger definition (or just the AFTER clause tuple variable if trigger has an AFTER clause), if not already present [Han99].
- Indexes the trigger definition under the signature obtained in the step above.

In the existing system, the only indexible predicate was the *equality* predicate, in presence of which the trigger was indexed using a *skiplist* [Pug90] on the constant set [Han99]. Otherwise, the trigger definitions were organized under a linear list structure.

The enhanced system can index on free-text predicates (the predicate consisting of tm_contains operator) as well, in addition to equality predicates. A new index structure, the *trie filter* (see previous chapters for details), is implemented to index free-text predicates efficiently so as to provide faster matching for these predicates. This structure indexes the predicate using the *query words* present in the predicate

In case the trigger definition contains both types of indexible predicates, we choose to index on the equality predicate rather than free-text one. This design choice is arbitrary as it is based on the belief that the rank of a free-text predicate is more than that of the equality predicate. A topic for future work is to implement an indexing system based on the cost-selectivity criterion of predicates based on the ideas described in the previous chapters.

Trigger Execution Engine

The TriggerMan system receives update descriptors (tokens) describing the events (insert, update, and delete) on the relations for which the system has triggers defined

[Han99]. These tokens are matched against the predicate index, and in case they match a trigger condition, the action associated with the trigger is executed.

In the enhanced system, support for matching the free text data (for which a free-text predicate exists) against the predicate index is added. The system converts the free-text data into a search structure (trie) on first use, and uses this particular index structure subsequently to match against free-text predicates. These free-text predicates can either be indexed inside the trie filter structure, as described previously, or may exist as the *restOfPredicate* [Han99] for some trigger definition. For the algorithm describing free-text matching, see chapter 5.

CHAPTER 8
PERFORMANCE EVALUATION

Introduction

In this chapter, we evaluate the performance of a prototype TriggerMan system implementation, capable of free-text as well as structured data alerting. We present various test results performed on the prototype implementation of the TriggerMan system along with the precise description of the test environment and methodology followed for testing.

The Test Environment and Methodology

The test environment was designed to study the performance of that part of the TriggerMan system that implements indexing for free-text predicates and performs matching between trigger conditions and free-text data using the free-text predicate index. The basic idea was to install triggers containing only a tm_contains function in their conditions. For each trigger, the free-text predicate inside the tm_contains function was randomly generated using a dictionary of words. The dictionary of words itself is a collection of 2000 randomly generated words having length between 3 and 7 characters. These words are associated with certain probability of occurrence inside a document. These probability values are close to each other, and form a smooth decreasing curve. A free-text predicate contains 2 to 4 words, and one randomly chosen type of operator (AND/OR/ACCRUE) connecting them. Hence, all our free-text predicates will be basic predicates. All the triggers are defined on the INSERT event of a single table; the text

attribute used inside tm_contains function is of type LVARCHAR. Also, the action part of every trigger contained the command *skip*, which is equivalent to a no-op. This was desired since we just wanted to test the time it takes to match the predicate index, and not the time it would take to execute the trigger actions. The following is an example of one of the triggers that were installed. *Textdata* is the table on which the triggers were defined; *tdata* is the free-text type attribute of the table used in the tm_contains () function call.

```
CREATE TRIGGER text_test
FROM Textdata
AFTER INSERT to Textdata
WHEN tm_contains (tdata, 'V482T & RTK7JY', BOOLEAN)
DO skip;
```

We tested the performance of the free-text predicate index by probing a random document of a certain size against it. The document is also composed of words from the same word list described above. We keep on adding randomly selected words to the document until it is at least of the required size. We are more interested in the case where the size of document is small (such as in the case of a newspaper article or memo field in a database), probably a couple of hundred bytes to a few kilobytes. We vary the size of document from 250 bytes to 4000 bytes to see the impact of document size on matching time. Each document is inserted as the tdata attribute of the table Textdata, the structure of which is described above. This will cause the trigger conditions to be tested and matched.

We implemented a client program to perform the tests described above. The client can open a connection to the Informix Dynamic Server, and can be used to either create a

fixed number of random triggers (as described above), or to insert a new row in the table on which triggers using tm_contains are defined (since all triggers are defined for the INSERT event). In addition, the client calls the tman_process_upd() function in the TriggerMan DataBlade. The tman_process_upd () function calls appropriate routines to perform matching of the predicate index with the newly arrived token (containing event-specific data), and activates the actions of successful triggers.

The tests were performed on a Dell Precision 420 workstation with following configuration:

Table 1. Workstation Configuration

No. of Processors	2
Processor Model	Intel Pentium III/ 966 MHz
Available Physical Memory	256 MB
Operating System	Windows 2000 Professional

The code is single threaded, and hence runs only on one processor. Next we describe various test results obtained using the above designed methodology.

Test Results and Performance Evaluation

Following were the two types of tests performed on the system:

- Trigger Creation Time Test - How fast can the system create *n* random triggers as described in previous section?

- Update Descriptor Processing Time Test - What is the response time for processing an update descriptor for a particular document size when *n* triggers are installed in the system? How does this time vary with the document size?

Trigger Creation Time Test

Figure 10 shows the results obtained for trigger creation times for different numbers of triggers.

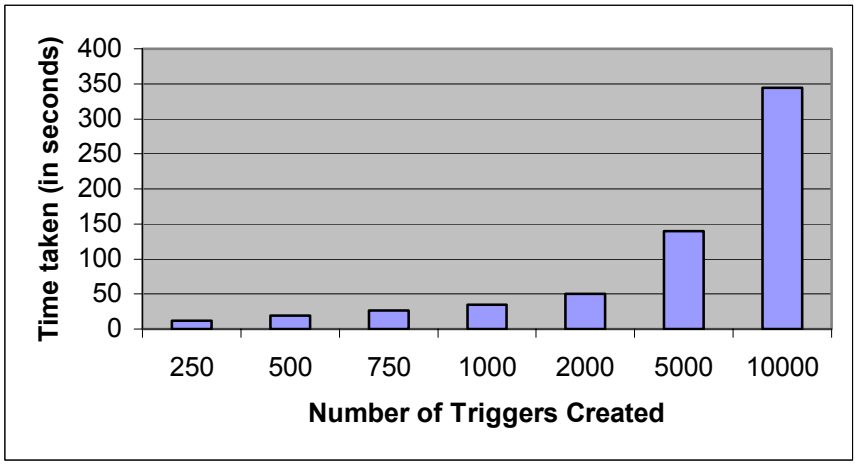

Figure 10: Graph of time taken against the number of triggers created.

Initially we observe a sub-linear relationship as we increment the number of triggers created. However, after a certain point, this relationship no longer seems to hold. The reason for this phenomenon can be explained as follows. As more and more basic predicates are created (all our triggers contain just basic predicates), the trie structure at the root level of the free-text predicate index, the trie filter, becomes deeper and deeper. Hence it becomes more and more time consuming to find whether a word already exists at the root level of the index structure. It could be the situation at other levels of the index structure too, though randomization does ensure to an extent that chances of that would be small (only true if the probabilities of occurrences for words are close, which is the

case in our environment). Again, this phenomenon could have been caused by the increased paging activity.

Average time to create a trigger on the basis of above observations is around 0.03 seconds, which promises a good average turnaround time to the client installing a trigger in the system.

Update Descriptor Processing Time Test

Figure 11 shows the update descriptor processing time versus the number of triggers present in the system for different document sizes.

The time increases as we increase the number of triggers, as can be anticipated, since the size of the predicate index increases, and we have to go deeper in the trie structure. Also, for a particular number of triggers, the time increases with the increase in the document size. The reason behind this is that we also build a trie structure on documents. Either the query words inside the predicate index are probed inside the document or the words inside the document are probed inside the predicate index. In both the cases, the increase in document size will result in increased time for update processing.

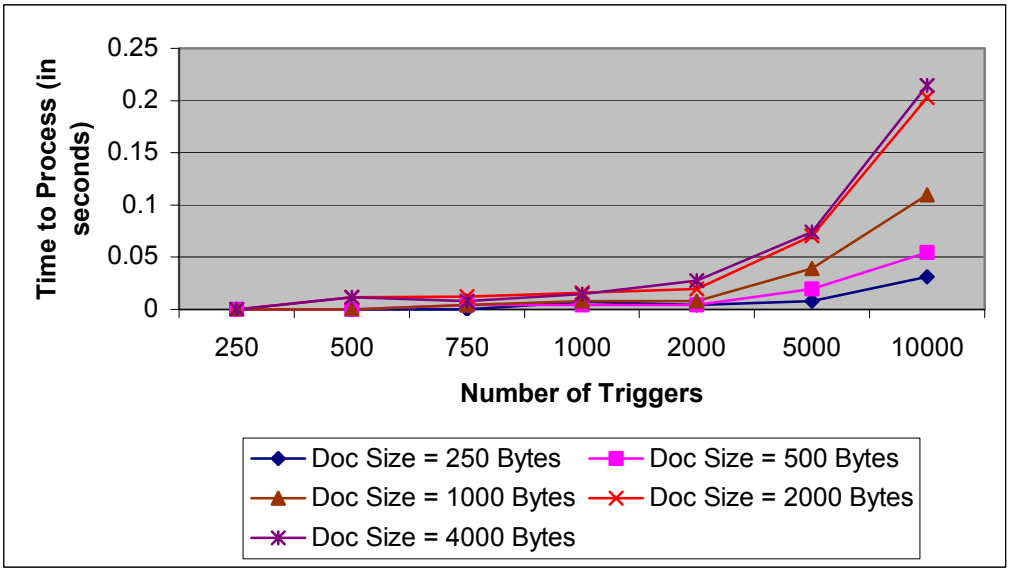

Figure 11: Update descriptor processing time versus number of triggers for various document sizes.

The update processing time will also depend on the actual number of triggers that are fired, and hence will be highly dependent to a particular setting. However, in this test environment where all the processing is based on random triggers and documents, we find a specific pattern on the number of triggers actually fired in different cases. Seemingly, the fraction of the triggers fired versus the total number of triggers installed remain almost the same for a particular document size, and is independent of the total number of triggers installed. These results are summarized in table 2.

These results, though synthetically generated, do show that the number of triggers fired increase with increase in document size, which is intuitive. However, the interesting pattern that appears for a particular document size happens to exist because of the random nature of triggers and documents.

Table 2. Fraction of triggers fired for various document sizes and total number of triggers.

Document Size (in Bytes)	Fraction of triggers fired versus the total number of triggers						
	250	500	750	1000	2000	5000	10000
250	0.04	0.04	0.06	0.05	0.05	0.05	0.05
500	0.10	0.09	0.10	0.11	0.10	0.09	0.09
1000	0.18	0.16	0.17	0.18	0.18	0.17	0.18
2000	0.30	0.30	0.31	0.31	0.31	0.30	0.30
4000	0.32	0.32	0.31	0.32	0.31	0.30	0.31

A realistic setting would portend that the trigger conditions would be more selective, and the number of triggers fired would be less. To simulate this particular setting, we change the free-text predicate in the trigger conditions. We generate free-text predicates containing two random words separated by an AND operator. We also use a uniform distribution of probability for words being used in trigger conditions and the documents. This makes the trigger conditions more selective. For example, let's consider a document containing 100 words. The Probability of a word appearing in this document would be $\frac{100}{2000}$ or 0.05. Hence the probability that both the words inside the trigger condition appear in the document would be $(0.05)^2$ or .0025. Figure 12 illustrates the results obtained for this particular setting. To reduce the impact of document trie creation time (highly variable) in the overall time taken for update descriptor processing, we consider these results for higher numbers of total triggers created, e.g. 1000 onwards. As an example, for 10000 triggers with this level of selectivity and for a document size of

2000 bytes, the system could process around $\frac{1}{0.020} \approx 50$ documents per second. The total number of triggers fired was around 3 % of the total number of triggers installed.

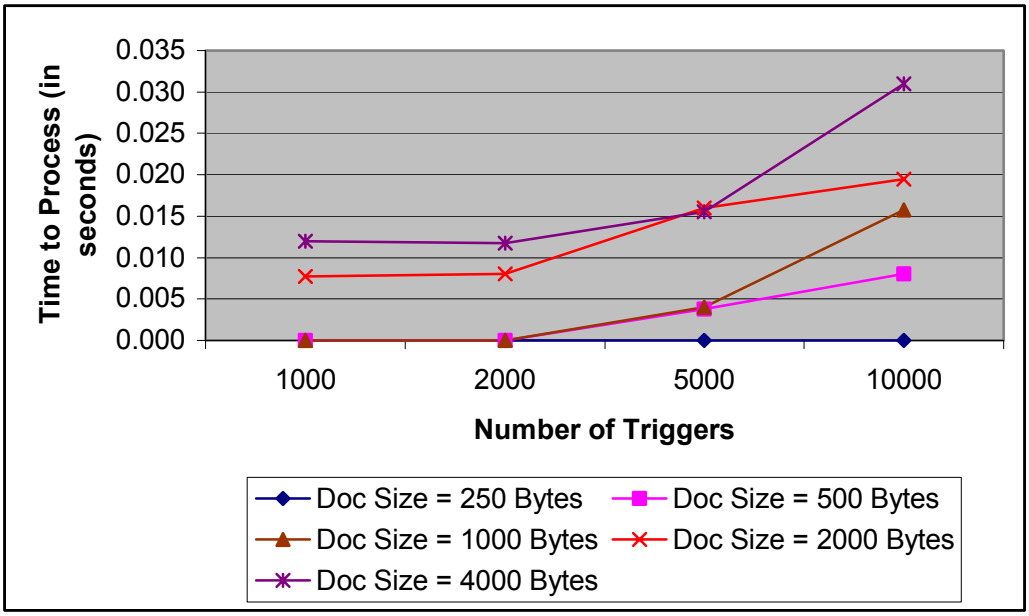

Figure 12: Update descriptor processing time versus number of triggers for various document sizes for more selective trigger conditions

CHAPTER 9
CONCLUSIONS AND FUTURE WORK

In this work, 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 given. 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. We have implemented a prototype version of the TriggerMan system capable of free-text alerting. We conducted various tests on this prototype implementation to measure its performance and efficiency. The results from this performance evaluation are promising. The system can support large number of triggers, and can process incoming tokens containing free-text data at a good speed. For example, processing time of 50 tokens containing 2000 byte documents per seconds with 10000 triggers (containing free-text predicates) installed in the system using predicate index was typical.

High speed alerting opens new horizons for publish/subscribe systems [Fab01], where clients subscribe to a customized service (by expressing their interests as trigger conditions or alerts) from an agent (the publisher). The publisher regularly sends new information to its clients, filtered as per their interests. The main cost incurred by the publisher is to test all the alerting conditions for various users, and send the information to them if found appropriate. In case the total number of alerting conditions is big and the

speed of incoming new information is high, it will become hard for the publisher to keep up its pace of testing all the alerting conditions for every piece of new information. A good example of such a publish/subscribe system where would be a personalized news service where people can get news specific to their interests. The TriggerMan system also enhances the existing triggering capabilities of underlying DBMS. It provides scalable trigger processing for numerous triggers on a single relation in a database. The free-text alerting capabilities provide alerting capabilities on unstructured free-text data, enhancing the set of trigger conditions that can be posed to the DBMS.

As future work, we propose the introduction of new operators for free-text predicates, such as for stemming, for case sensitive matching etc. We also propose inclusion of structured free-text data for alerting, such as XML documents. Another area for future work is to optimize the trigger condition evaluation by restructuring it, since it may contain expensive predicates, and proper ordering of predicates for evaluation will save evaluation time for a non-matching trigger condition. Some such optimization techniques for queries with expensive predicates are discussed by Kemper et al. [Kem92, Kem94]. The idea behind these techniques is to evaluate first the predicates with relatively higher selectivity and lower cost than the other predicates. We propose employing these techniques to restructure the restOfPredicate [Han99], the non-indexable part of trigger condition. This would optimize the trigger condition evaluation time. We have already described an area for future work in chapter 6 as the improvement to the formula for the cost estimation of index search for a predicate. Also, the prototype system implements the design to a limited extent. Hence we propose a complete implementation of the design, e.g. implementation of a sub-system to provide the weight of a word,

optimal predicate selection for indexing using the algorithm presented in Appendix B, and elimination of stop words from documents.

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 quote. A white space is any character that is not in S and is not an operator.

APPENDIX B
HEURISTIC TO FIND APPROPRIATE INDEXABLE PREDICATE

Main Algorithm to find the best predicate to Index

```
Algorithm FindPredicateToIndex (Predicate Pred)
{
     if there does not exist any indexable term
          return NULL.
     Let P1',…,Pm' be the indexable terms in Pred.
     if m = 1
          return P1'
     else
     {
          /* choose one indexable predicate and
          determine its search costs*/
          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 Pj', let P1",…, Pk" be the remaining
          terms in Pred.
```

```
            /* This step determines the fitness metric of
            a particular indexable predicate used for
            indexing the trigger condition. This metric
            is given as the rank of the predicate. The
            predicate having the least rank is the one to
            index on. */
            For every indexable term Pj', 1 ≤ j ≤ m,
            compute the rank of Pj' given by the
            following formula, a variation of the formula
            proposed by Hellerstein [Hel93, Hel94]
            (Cost_brute_force_predArray is defined
            shortly):
```

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

```
            return Pj' with the least rank value.
        } // end else
}// end Algorithm FindPredicateToIndex
```

Helper Functions

```
/* This function finds the cost of searching predicates
in a brute force fashion, aapplying conjuncts in order
from most to least selective and using conditional AND
logic. */
Function Cost_brute_force_predArray
        (Array predArray[1…k]:INPUT)
{
```

```
     for i = 1 to k
```

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

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

/* This function computes the cost of a brute force
search on a predicate array of a given size, using
conditional AND logic. */
Function Cost_brute_force_helper
     (Array predArray[1…k], integer k:INPUT)
{
     Cost = 0
     ProbabilityTestIsRequired = 1
     for i = 1 to k
     {
          Cost = Cost + ProbabilityTestIsRequired *
          Cost_brute_force_search(predArray[i])
          ProbabilityTestIsRequired =
          ProbabilityTestIsRequired * sel(predArray[i])
     }
     return Cost
}
```

REFERENCES

[Alt00] Altinel, M. and Franklin, M. J., Efficient Filtering of XML Documents for Selective Dissemination of Information, In *Proceedings of the 26th VLDB Conference*, 2000, pp. 53-64.

[Bae99] Baeza-Yates, R. and Ribeiro-Neto, B., Modern Information Retrieval, *Reading*, Addison-Wesley, 1999.

[Che00a] Chen, J., DeWitt, D., Tian, F., and Wang, Y., NiagaraCQ: A Scalable Continuous Query System for Internet Databases, In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 2000.

[Che00b] Chen, Z., Korn, F., Koudas, N., and Muthukrishnan, S., Selectivity Estimation for Boolean Queries, In *Proceedings of the ACM Symposium on Principles of Database Systems*, 2000.

[Fab01] Fabret, F., Jacobson, H. A., Llirbat, F., Pereira, J., Ross, K. A., and Shasha, D., Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems, *ACM SIGMOD*, 2001, Vol. 30, No. 2, pp. 115-126.

[Fag96] Fagin, R., Combining Fuzzy Information from Multiple Systems, In *Proceedings of ACM Symposium on Principles of Database Systems*, 1996, pp. 216-226.

[Han99] Hanson, Eric N., Carnes, C., Huang, L., Konyala, M., Noronha, L., Parthasarathy, S., Park, J. B., and Vernon, A., Scalable Trigger Processing, In *Proceedings of the IEEE Data Engineering Conference*, 1999, pp. 266-275.

[Hel93] Hellerstein, J. M., Predicate Migration: Optimizing Queries with Expensive Predicates, *ACM SIGMOD*, Vol. 22, No. 2, 1993, pp. 267-276.

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

[Inf01a] Informix Corporation, Informix Guide to SQL, http://www.informix.com/answers/english/docs/visionary/infoshelf/sqls/sqls.ix.html, 08/15/2001.

[Inf01b] Informix Corporation, Informix Dynamic Server, Universal Data Option, http://www-4.ibm.com/software/data/informix/, 08/15/2001.

[Inf01c] Informix Corporation, User's Guide, Excalibur Text Search DataBlade Module, www.informix.com/answers/english/docs/datablade/5401.pdf, 08/15/2001.

[Inf01d] Informix Corporation, User's Guide, Verity Text Search DataBlade Module, www.informix.com/answers/english/docs/datablade/8245.pdf, 08/15/2001.

[Jag99] Jagadish, H. V., Kapitskaia, O., Ng, R. T., and Srivastava, D., Multi-Dimensional Substring Selectivity Estimation, In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, 1999, pp. 387-398.

[Kem92] Kemper, A., Moerkotte, G., and Steinbrunn, M., Optimizing Boolean expressions in object bases, In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, 1992, pp. 79-90.

[Kem94] Kemper, A., Moerkotte, G., Peithner, K., and Steinbrunn, M., Optimizing disjunctive queries with expensive predicates, In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 1994, pp. 336-347.

[Knu73] Knuth, D., The Art of Computer Programming: Fundamental Algorithms, *Reading*, Addison-Wesley, 1973.

[Kol98] Kolda, T. G. and O'Leary, D. P., A Semidiscrete Matrix Decomposition for Latent Semantic Indexing in Information Retrieval, *ACM Transactions on Information Systems*, Vol. 16, No. 4, October 1998, pp. 322-346.

[Liu99] Liu, L., Pu, C., and Tang, W., Continual Queries for Internet-Scale Event-Driven Information Delivery, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 11, No. 4, 1999, pp. 610-628.

[Nel98] 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.

[Por80] Porter, M. F., An Algorithm for Suffix Stripping, *Program*, Vol. 14, No. 3, 1980, pp. 130-137.

[Pug90] 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, In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 1979, pp. 22-34.

[Yan94] Yan, T. W. and Garcia-Molina, H., 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 Garcia-Molina, H., The SIFT Information Dissemination System, *ACM Transactions on Database Systems*, accepted, to appear.

BIOGRAPHICAL SKETCH

Himanshu Raj received his Bachelor of Technology degree in computer science and engineering from the Indian Institute of Technology, Guwahati. His areas of interest are database systems, mobile computing and security. He can be reached at hr0@cise.ufl.edu or rhim@lycos.com.