Report Number: WUCS-91-06

1991-05-01

Efficient Queries For Quasi-Ordered Databases

Victor Jon Griswold

The problem of selecting database tuples is central to the task of resolving many forms of queries. The report defines a class of selection queries and presents a set of algorithms for their efficient resolution. The class of queries investigated is those queries which impose a quasi order on the tuples in a database.

Efficient Queries For Quasi-Ordered Databases

Victor Jon Griswold

WUCS-91-06

May 1, 1991

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis, Missouri 63130-4899

Abstract

The problem of selecting database tuples is central to the task of resolving many forms of queries. This report defines a class of selection queries and presents a set of algorithms for their efficient resolution. The class of queries investigated is those queries which impose a quasi order on the tuples in a database.

This work has been supported by NSF grant DCI-8600947, Bellcore, BNR, Italtel, NEC, DEC, SynOptics, and NTT. The author may be reached via email address vjg@wucs1.wustl.edu.

Copy to:

Andreas Bovopoulos
Neil Haller - Bellcore
Alan Kirby - DEC
Ken-ichi Yukimatsu - NTT
Guru Parulkar
Ron Schmidt - SynOptics
Jonathan Turner
Akira Arutaki - NEC America

Abstract to:

Milind Buddhikot
Jerome R. Cox, Jr.
Chuck Cranor
Zubin Dittia
Andy Fingerhut
Larry Gong
Rex Hill
Diamantis Kotoulas
Nader Mirfakhraei
James Sterbenz
Einir Valdimarsson
Ellen Witte

TABLE OF CONTENTS

LIST OF FIGURES

No. Page

Efficient Queries For Quasi-Ordered Databases

Victor Jon Griswold

1. Introduction

1.1 Background

The project leading to the work presented in this report involves the monitoring of distributed systems by means of observing "events" generated by the systems being monitored. By comparing functions of their attributes, these events are ordered in logical time and validated against user-specified behavior constraints.[6, 7] Within the monitor, incoming events are most naturally treated as database tuples.* The event attributes correspond directly to tuple components.

The validation and time-ordering subsystems of the monitor must frequently query the event database. By far the most common form of these queries picks a single event V (generally the event just received by the monitor), finds all the previously-recorded events related to V by some criterion (such as "find the **transmit** corresponding to V, a **receive**"), and performs some action on each of those events related to V.

The goal of this report is to find algorithms which respond efficiently to this form of query on the event database. Towards this goal, we will make use of a novel property of our

---

* To be precise, the term database refers not just to a set of tuples, but rather to the higher-level structure which includes the tuple set, indices used to access those tuples, and other auxiliary information. In this report, though, we will abstract somewhat and use database synonymously with "set of tuples" except where ambiguity may result.

application. With most database systems, the database is relatively stable and arbitrary queries may be posed against it. The above monitor, however, knows the exact form of each query before any events are received. Tuples, the received events, are subsequently entered into the database incrementally (starting with an empty database).* This prior knowledge of the queries to be posed allows significant optimization of the structures used to store the database. Given the same events, two monitor executions with different inference rules (which define which queries will be posed) will generally use different structures to store those events.

1.2 General Problem

The monitor's inference rules compare one event V at a time with those events in the database "related to" V. The question now becomes "what relationships can be specified which are both useful and can be resolved efficiently?" In general, these relationships are comparisons between event attributes. Consider, for example, events associated with a communications channel. The packet identifiers for a "transmit" and a "receive" event may be compared to determine if they match. Similarly, the "sequence number" attributes of two events may be compared to decide which should have been processed first.

The first example above checks for equality between two event attributes; the second checks for a "less than" or "greater than" ordering between attributes. The relationships supported for the monitor's database queries are conjunctions (combinations via '**and**') of such checks. If disjunctions ('**or**' combinations) are needed, one may simply pose more than one query and combine the results.**

Many, if not most, of the queries posed by the monitor are done in order to determine the temporal (time) orderings between a newly-recorded event and those events already known to the monitor. Time imposes a transitive ordering between events; if A occurs before B and B occurs before C, A occurs before C. Suppose the monitor has just been notified of C and is attempting to order its already-known events with C. A database query could be posed which resolves to both A and B. The inclusion of A in this result, however, is redundant; the fact that A occurs before C may be inferred through other information and does not need to be known explicitly. Though the inefficiency for this example is small, consider what might happen when monitoring

---

* Note that queries are interleaved with these additions.

** This involves reducing the original query Q to disjunctive normal form and posing separate queries for each of Q's conjunctive terms.

a system which generates thousands of events. Given the format of the relationships in the monitor's database queries, any relationship involving one or more "less than" or "greater than" checks will be transitive in this manner and thus might produce query results with redundant information.

For this reason, it is deemed a monitor efficiency requirement that a query Q(V) (i.e. "find those events related to V according to relationship Q") not return events the relationship of which may be inferred through an additional posing of Q with some other event in the result of Q(V). Stated another way, assume that U and W are in the result of Q(V). This implies that U can not also be in the result of Q(W), since, if it were, the inclusion of U in Q(V)'s result would be redundant through transitivity.

1.3 Definitions

As stated above, the monitor treats events as database tuples and event attributes as tuple components. This allows us to formally define the query problem using more traditional database terminology. We also generalize the query format to allow comparisons of functions of tuple components (such as "time + 5 milliseconds") in addition to just the component values themselves. Of course, the monitor can (almost always does) pose many different queries on the database, each complying with the format below.

Q. Given a tuple v, a set of tuples D, and a relation precedes (denoted by '$\prec$') which imposes an irreflexive partial ordering* on the tuples in D, determine the maximal elements of the set of D's tuples which **precede** v. This query Q(v) is defined as:

$$Q(v) = \left\{ t \in D \;\middle|\; \begin{array}{c} t \prec v \\ \nexists u \in D \; [\, t \prec u \prec v \,] \end{array} \right\}$$

where: D is a set of tuples, a database; $d \equiv |D|$

$\prec$ is an irreflexive partial order relation between tuples

Precedes is defined in terms of comparisons between functions of tuple components. These comparisons may involve the equality or the total (linear) ordering of function values. The complete definition of **precedes** is:

---

* An irreflexive partial order is a quasi order, meaning that $A \prec A$ is false. Though quasi order is the proper description of the ordering we require, few people regularly use this term. Throughout the remainder of this paper, partial order will be used for quasi order except when ambiguity may otherwise result.

$$t \prec v \equiv F_1(t)\ \mathbf{R}_1\ G_1(v)\ \wedge\ F_2(t)\ \mathbf{R}_2\ G_2(v)\ \wedge\ \ldots\ \wedge\ F_k(t)\ \mathbf{R}_k\ G_k(v)$$

where: t, v are tuples; $t \in D$

there are a finite number k comparisons making up the relation

for each i, $1 \leq i \leq k$:

Let C_i be a set whose elements can be partitioned into equivalence classes by a relation '=' and might additionally be totally (linearly) ordered by a relation '<'*

F_i: tuple $\rightarrow C_i$; G_i: tuple $\rightarrow C_i$

$\mathbf{R}_i$ is one of = or $\neq$ over C_i, or <, >, $\leq$, or $\geq$ if applicable**

Each '*FRG*' group above is called a factor in the definition of **precedes**. Each F_i and G_i can be any function of the designated type. Factors which involve '=' or '$\neq$' comparisons are called equivalence factors; those which involve '<', '>', '$\leq$', or '$\geq$' are called total-order factors. For convenience, we define the relation **follows**, denoted by '$\succ$', in terms of **precedes**. The statement "$t \prec u$" is equivalent to "$u \succ t$." Tuple components are designated by dotted notation on a tuple; for instance, $t.c_1$ is the first component of t.

To illustrate such a **precedes** relation, consider a database of 2-tuples over the integers. We wish to define $t \prec v \equiv (t.c_1 = v.c_1) \wedge (t.c_2 < v.c_2)$. There are two factors ($k = 2$), the first of which is an equivalence factor and the second a total-order factor. C_1, F_1, $\mathbf{R}_1$, G_1 are $\mathbb{I}$, $c_1(u)$, '=', and $c_1(u)$, respectively, u the tuple parameter of the functions. Likewise, C_2, F_2, $\mathbf{R}_2$, G_2 are $\mathbb{I}$, $c_2(u)$, '<', and $c_2(u)$, respectively.

1.4 Primary Issues

Three main issues are distinguished in the resolution of this class of queries. This report devotes a section towards addressing each of these.

The first problem is the efficient evaluation of all equivalence factors of a **precedes** relation. Though the solution presented is relatively straight-forward, it organizes a framework for the evaluation of total-order factors and presents an enticing optimization problem.

---

* Allowing partial orders was investigated, but difficulties were found with the convergence of several search strategies when applied to "realistic" partial orders.

** We define $\neq$, >, $\leq$, and $\geq$ according to their classical meanings in terms of < and =.

The second issue is, naturally, the efficient evaluation of all total-order factors of a **precedes** relation. Each relation can include the conjunction of several total-order factors. Though a single total order is trivial to search for maximal elements (remember the query's non-redundancy requirement), the conjunction of multiple total orders results in a partial order which can be difficult to search for maximal elements.

The third issue concerns equivalence factors and multiple non-identical queries over a database. When several queries are posed, it is desirable to find information common between those queries so that some evaluation need only be performed a single time. This report addresses the use of equivalence factors common between queries to form the basis of such optimizations.

1.5 Orthogonal Range Queries

An orthogonal range query[16][3][11] over D, a set of k-tuples, is a query defined as:

$$\mathbf{RQ}(D,\ a_1..b_1,\ a_2..b_2,\ \ldots\ ,\ a_k..b_k) = \left\{ t \;\middle|\; \begin{array}{c} t \in D \\ a_i \le t.c_i \le b_i \end{array} \right\}$$

More simply put, an orthogonal range query is a multi-dimensional (or multi-keyed) search across a database in which a range of values for each of the dimensions (components, keys) may be specified.

The class of queries Q is similar to a class of orthogonal range queries, with the exception of the "maximal elements" requirement. Thus, range query techniques frequently are either incorporated directly into the algorithms presented in this report or are used as a standard for comparison. Since they will soon be important, we take this opportunity to examine the time and space requirements of orthogonal range queries.

Let $m \equiv$ *the maximum number of distinct values in any tuple component.* The search time for the best orthogonal range query algorithms is $O((\log d)^{k-1})$, $k \ge 2$. Insert/delete time is similar, $O((\log d)^k)$. Space requirements are $O(d(\log m)^{k-1})$. As can readily be seen, both time and space considerations make these algorithms useful only for small dimensionalities (realistically, dimension 2, maybe 3, due to space considerations).

Appendix 5.2 describes how a search is performed using common range query data structures.

2. Equivalence Factors

2.1 Requirements

For a single query Q, equivalence factors might be handled by concatenating the factors' C_i values into a single key and using an ordinary search tree, resulting in $O(\log d)$ resolution time*. The goals of this report, however, include the efficient resolution of a set Q of queries, $q \equiv |Q|$, not all of which employ the same equivalence factors. Queries Q_1 and Q_2 in Q, for example, might have three equivalence factors in common, while Q_2 and Q_3 might have none in common. Furthermore, when selecting a resolution algorithm, one must keep in perspective that the algorithm must facilitate efficient maintenance of the data structures necessary for query resolution, given an incrementally constructed database D.

It is the multi-dimensional aspect of this problem which made algorithms for orthogonal range queries[16][3][11] appear promising. Each equivalence factor in the elements of Q could be represented as a separate dimension to the range query. To resolve a particular query $\mathbf{Q} \in Q$, each factor i present in Q would be searched for the single C_i value significant to Q, while each factor in some element of Q but not in Q would be searched along its entire range of values (effectively "wildcarding" that factor). Though range queries are capable of far more elaborate searches, the query problem Q has no obvious use for them.

Let the number of equivalence and total-order factors in a query be distinguished by defining $k_e \equiv$ *the number of equivalence factors* and $k_t \equiv$ *the number of total-order factors* ($k = k_e + k_t$). When discussing sets of queries, it is useful to define $K_e \equiv$ *the number of distinct equivalence factors across all* $\mathbf{Q} \in Q$. Finally, considering only the equivalence factors of the queries in Q, let $m_e \equiv \max_{1 \le i \le K_e} |C_i/D|$, where C_i/D is that subset of C_i induced by application of F_i over the tuples in D (i.e. the number of unique C_i values actually used when evaluating queries over D). Given the above "brute force" range query approach, the search time for the best orthogonal range query algorithms is $(\log d)^{K_e-1}$. Insert/delete time is similar, $(\log d)^{K_e}$, and space requirements are $d(\log m_e)^{K_e-1}$.

If Q needed the full power of range queries, this would be considered good. The fact, however, is that Q does not. The data structures and algorithms chosen for use in query

---

* For brevity in the remainder of this report, all time and space complexity measurements shall be assumed to be asymptotic complexities ("O") unless otherwise stated.

evaluation are derived from those used for orthogonal range queries via the two-stage reduction presented below. This reduction yields search and update times which remove the exponential dependency on K_e at the cost of additional multiplicative factors. Space cost is similarly improved.

2.2 Approach

The first optimization to range queries derives from the "one value or all" nature of the searches across each factor. Range queries are designed to allow any range of values to be selected: that is the purpose of the $\log m_e$ auxiliary tree levels for each dimension, as shown in Appendix 5.2. If one only requires single-value or all-value searches, however, just the top and bottom auxiliary tree levels are required. This corresponds to the auxiliary trees for the entire current database subset and those for the leaves of each subset's dimension's search tree.

The time and space savings for this first reduction are already appreciable. Search time and insert/delete time decrease to $O(K_e \log d)$ and $O(2^{K_e-1} \log d)$, respectively. Space requirements diminish to $O(2^{K_e-1} d)$. There are still, however, factors in these measures which are exponential in K_e. The second reduction removes these complexity factors at the cost of requiring multiple query search structures, perhaps one per query, instead of one structure sufficient for all queries.

The K_e-exponential complexity factors occur because the range query data structure must take into account two possibilities for each dimension. The first is that a given query will indeed search on that dimension; the second is that a given query will "ignore" that dimension. Considering the application domain of a query set Q (monitoring a distributed system), it appears quite likely that there will be a large number of equivalence factors, but that only a few factors will be present in each query. For this case, a substantial majority of dimensions will be ignored when resolving each query.

The second optimization, therefore, constructs a separate search data structure for each $Q_j \in Q$. Each of these structures is of dimension k_{e_j}, not K_e, and is concerned only with searching across the factors actually present in Q_j. The "all values" auxiliary trees are no longer needed, so each dimension has only a single level of auxiliary trees — the level for each distinct value of the dimension.

Figure 1 shows an example of the structure used for resolution of a query with three equivalence factors (three dimensions), the domain of each factor being values in the sets A, B,

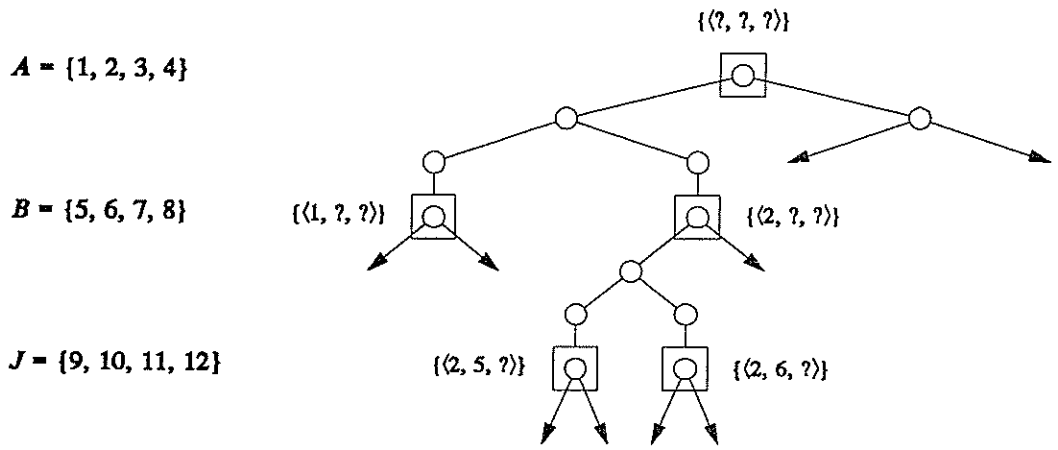


Figure 1. Equivalence-Factor Search Data Structures

and J. Tree roots are shown in boxes. At the root of the base tree, a tuple may have any value for any of the three factors. The base tree sorts by the first dimension, so it has one leaf for each value in set A. For each tree in the first level of auxiliary trees, the tuples in that tree share a property similar to those in standard range query data structures. In a standard structure, the first-dimension values of those tuples would all lie in a certain range. With this modified structure, that range is a single value. The first level auxiliary trees all sort by the second dimension and thus have one leaf for each value in B (so long as there is a tuple with that value, of course). The third dimension is correspondingly handled by the second level of auxiliary trees.

Searching this structure is straight-forward. For each query, search the base tree by the first factor. Next, search the first-level auxiliary tree just found (leaf of the base tree) by the second factor. To finish, search the appropriate second-level auxiliary tree by the third factor. The end of this search is a leaf of a second-level auxiliary tree, the tuples at which comprise the result of the query.

Since the data structure used by the first range query optimization is used for all queries, care must be taken when comparing it to the second optimization, which requires a separate structure for each query. Search time is still in terms of a single query, but insert/delete time and space requirements must be stated in a manner applying to all queries. Since not every query can be applied to each tuple t added to D, $q(t)$ is defined as the number of queries $\mathbf{Q}$ in Q for which t is in the domain of every factor of $\mathbf{Q}$. Let us call this set of queries $Q(t)$. Generally, $q(t)$ will be much less than q. We also define $K_e(t)$ in terms of $Q(t)$. Slightly different from K_e, $K_e(t)$ is the maximum k_{e_j} for any $\mathbf{Q}_j$ in $Q(t)$, not the total of all k_{e_j}.

The second optimized algorithm has a search time of $O(k_{e_j} \log(d/k_{e_j}))$ for query Q_j (time just slightly lower than the obvious). Time to insert or delete a tuple t is $O(q(t)K_e(t)\log(d/K_e(t)))$. Both of these occur when $|C_i|$ is the same for each dimension (i.e. all dimensions have substantial search time) and an $O(\log m_e)$ search through each dimension is forced by having single tuples rooted in all the leaves of each level's tree structure except for the leaf corresponding to the that level's search key, where all the remaining tuples are rooted for the next dimension. Space cost for the algorithm is at absolute worst $O(q\,d \max_{t \in D} K_e(t))$, which occurs when $|C_i| = d$ for the first factor of every query (i.e. when tuples have unique first-factor values; all auxiliary trees have only a single node, but there must still be k_e auxiliary tree levels). This measure is terribly pessimistic since it assumes that $q(t) = q$ for all $t \in D$ and uses the largest $K_e(t)$, as well.

Comparing the first and second optimized query algorithms, search time for the second, $O(k_{e_j} \log(d/k_{e_j}))$, is distinctly better than the $O(K_e \log d)$ time required for the first. The $O(q(t)K_e(t)\log(d/K_e(t)))$ insert/delete time for the second algorithm becomes better than that of our first algorithm, $2^{K_e-1}\log d$, as soon as K_e becomes moderately large. Storage cost for the second algorithm, $O(q\,d \max_{t \in D} K_e(t))$, also becomes better than the $O(2^{K_e-1} d)$ cost of the first algorithm as K_e becomes moderately large.

The only problem with the second algorithm is the waste of time and space forced by entering each tuple t into $q(t)$ data structures. This problem, however, can be substantially avoided by combining the search data structures for queries with equivalence factors in common into a single, though slightly more complex, data structure. Determining the optimal combination of such query structures is the topic of Section 4 in this report.

3. Total-Order Factors

3.1 Requirements

In this section, the problem of evaluating the total-order factors of a **precedes** relation is addressed, ignoring any equivalence factors. To simplify the discussion without loss of generality, we restrict factor definitions to functions directly on tuple components (this is how **precedes** relations were first introduced in the early part of Section 1). The specific restrictions necessary for this are listed below; a query meeting these restrictions is referred to as a Q_t(v) query, instead of as the general Q(v) query.

Restrictions on this section's factor definitions:

1) all (total-order) factors use the '<' relation
2) for each k_t-tuple $t = \langle c_1, c_2, \ldots, c_{k_t} \rangle$, all components $c_i \in \mathbb{I}$
3) for each i, $1 \le i \le k_t$, $C_i \equiv \mathbb{I}$, $F_i(t) = t.c_i$, and $G_i(v) = v.c_i$
4) for all components c_i and all distinct $t,u \in D$, $t.c_i \neq u.c_i$

With respect to restriction 1 above, the algorithms presented here can be readily modified to handle the relations '>', '≤', and '≥', as well. Explicit handling of these other relations will needlessly complicate discussion. Restrictions 2 and 3 simply make each factor an integer comparison between the values of the same component of two tuples. Restriction 4 is useful both for stating algorithms and for stating LEMMA 1 below. With algorithms, the use of distinct component values obviates the need for tedious list operations for cases of multiple tuples with the same component value. With LEMMA 1, the use of distinct component values allows a much simplified problem definition which can ignore the lengthy but non-substantive case analysis involved when components of two tuples have the same value.

Given these restrictions, an example **precedes** relation can now be defined as:

$$t,\ v \text{ are 2-tuples } \langle x, y \rangle$$
$$t \prec v \equiv\ t.x < v.x\ \wedge\ t.y < v.y$$

instead of

$$t, v \in \{\ \langle x, y \rangle \mid x \in \mathbb{I} \wedge y \in \mathbb{I}\ \}$$
$$t \prec v \equiv \exists i[1 \le i \le 2\ \wedge\ F_i(t) < G_i(v)]\ \wedge\ \nexists j[1 \le j \le 2\ \wedge\ F_j(t) > G_j(v)]$$
$$F_1(t) = t.x\ ;\ G_1(v) = v.x$$
$$F_2(t) = t.y\ ;\ G_2(v) = v.y$$

without misunderstanding. This relation is the partial order which results when $t \prec v$ means that both components of t are less than the corresponding components of v (or, more verbosely, that no $t.c_i$ equals $v.c_i$, at least one $t.c_i$ is less than $v.c_i$, and no $t.c_i$ is greater than $v.c_i$).

In fact, the result of imposing any ordering relation of this class across all the k_t-tuples in a database D, $k_t \geq 2$, is a partial order of those tuples (k_t=1 results in a trivial total order). For example, suppose 3-tuples of type $\langle x, y, z \rangle$ with ordering relation $\prec$ defined as:

$$t \prec v \equiv t.x < v.x \ \wedge \ t.y < v.y \ \wedge \ t.z < v.z.$$

And suppose the database is:

$$D = \{\langle 2,1,5\rangle, \langle 3,3,3\rangle, \langle 5,5,4\rangle, \langle 4,6,8\rangle, \langle 6,2,2\rangle, \langle 8,7,6\rangle\}.$$

Figure 2 shows a representation of D in the right-hand diagram. A data structure analogous to this is used by all the Q_t algorithms below and would exist for each query in Q (thus a given tuple t is referenced in $q(t)$ structures). The dotted vertical lines represent the set of values for components x, y, and z. Each tuple is shown as two solid (non-vertical) lines connecting its three specific component values. With this type of diagram, it is easy to visually determine which tuples **precede** each other, which **follow**, and which are incomparable or "cross" each other. The graph G in Figure 2 shows the transitive reduction of the partial order imposed on D by $\prec$. As defined in Section 1, the resolution of the query $Q(v)$ must include all tuples $t \in D$ such that $t \prec v$ and for which there does not exist a tuple $u \in D$ such that $t \prec u \prec v$. Simply put, $Q(v)$ is the tails of all edges with head v in the graph G. This means:

$Q(t_1) = \{\}$	$Q(t_3) = \{t_2\}$	$Q(t_5) = \{\}$
$Q(t_2) = \{\}$	$Q(t_4) = \{t_1, t_2\}$	$Q(t_6) = \{t_1, t_3, t_5\}$

Note that $Q(t_6)$ does not include t_2. That ordering is implied through transitivity.

We now investigate three approaches to resolving Q_t queries. The first algorithm employs classical search techniques. The second is primarily an investigation into the possibility of resolving a query on a tuple v by using data generated during the resolution of that query on previous tuples (i.e. an incremental approach). The last algorithm resolves a Q_t query through use of a series of orthogonal range queries. Each of these three algorithms makes use of the observation that all tuples t in the resolution of Q_t(v) "cross" each other when viewed as in the right-hand diagram in Figure 2 (more formally, the tuples are incomparable).

LEMMA 1. Given tuples t_r, $t_s \in$ Q_t(v), there exists a component c_i such that $t_r.c_i < t_s.c_i$ and there exists a component c_j such that $t_r.c_j > t_s.c_j$.

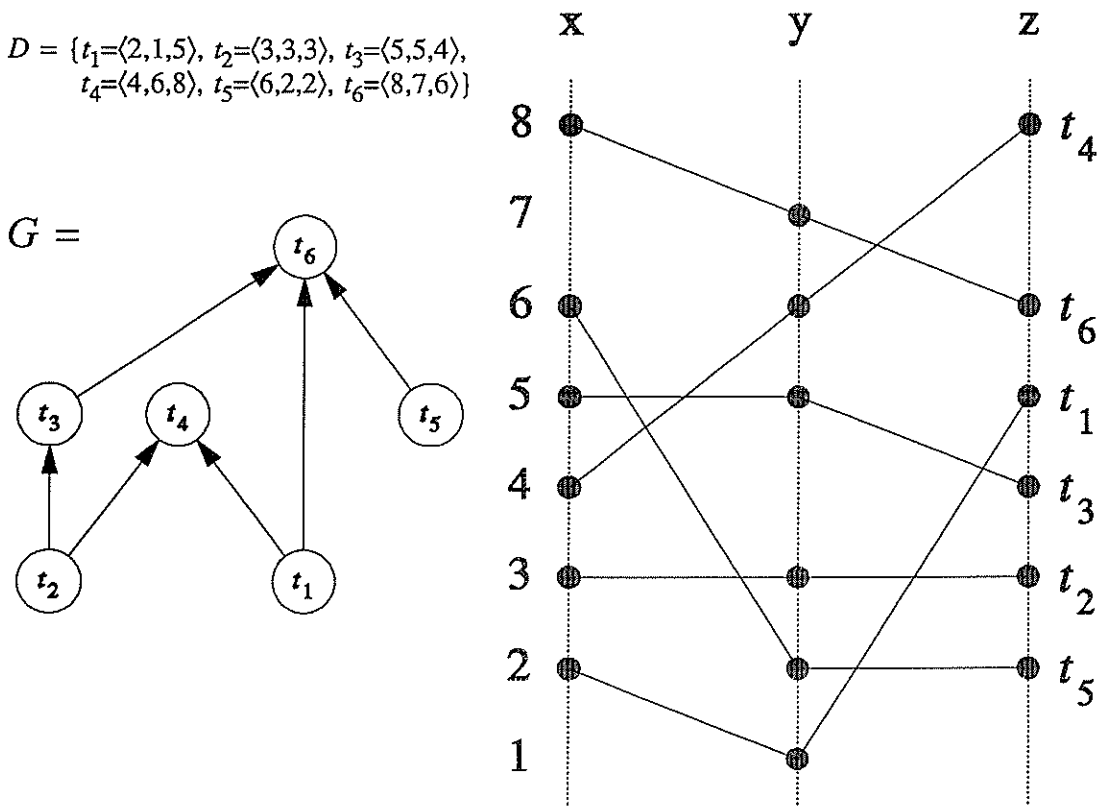


Figure 2. Precedence Resulting From Three Components Each Ordered By '<'

PROOF. This is direct from the definitions of Q and Q_t. If there is no component $c_i \ni t_r.c_i < t_s.c_i$, it means that $t_r \succ t_s$ (remember: component values are unique in Q_t) and thus t_s could not be an element of Q_t(v) due to the "$\nexists u \in D\ [t \prec u \prec v]$" clause of Q's definition. Similarly, if there is no component $c_j \ni t_r.c_j > t_s.c_j$, $t_r \prec t_s$ so t_r could not be an element of Q_t(v).

Q.E.D.

3.2 Non-Incremental Algorithm

Figure 3 shows the first Q_t algorithm, Q_t_NI, using the Pascal-like notation described in Appendix 5.1 and introduced in [6, 7]. As can be seen in the declarations, tuples are treated as simple arrays of integers and the columns in the right-hand diagram of Figure 2 are recorded

```
constants
    k_t : natural := the number of (total-order) components in a tuple;

types
    tuple = array [1..k_t] of integer;
    C_node = record
        val : integer;
        t :   ^tuple;
    end C_node;

globals
    C : array [1..k_t] of srt_set of C_node key val;


function next_candidate(i : natural; bound : tuple) : ^tuple;
    candidate : ^tuple := C[i].prev();   //   the tuple below the latest one found on C[i]
begin
    while candidate ≠ null andif candidate^ ⊀ bound do     //   make sure candidate is admissible
        candidate := C[i].prev();
    endwhile;
    return candidate;
end next_candidate;


function Q_t_NI(v : tuple) : list of ^tuple;
    Qlist : list of ^tuple := [];
    candidate : ^tuple;
    t_X : ^tuple;                    //   inspect all tuples with component values between v and t_X
    i :  natural;                    //   the tuple component on which Q_t is working
begin
    t_X := C[1][v[1]]→t;             //        Find the first tuple ≺ v whose c_1 value is less than
    t_X := next_candidate(1, v);     //   that of v.  All other tuples put into Qlist will "cross" t_X.
    if t_X ≠ null then               //   make sure there is at least one tuple ≺ v
        Qlist += t_X;
        for i in [2..k_t] do
            candidate := C[i][v[i]]→t;          //   find the first tuple ≺ v with c_i between v.c_i and t_X.c_i
            candidate := next_candidate(i, v);
            while candidate ≠ null andif candidate[i] > t_X[i] do     //   examine all such admissible tuples
                if ∄u∈ Qlist [candidate^ ≤ u^] then     //   make sure candidate is maximal
                    Qlist += candidate;
                endif;
                candidate := next_candidate(i, v);      //   next admissible tuple
            endwhile;
        endfor;
    endif;
    return Qlist;
end Q_t_NI;
```

Figure 3. Query by Total-Order Factors (Non-Incremental)

as sorted sets* of ⟨integer, ^tuple⟩ pairs in the array C. Intuitively, each C[i] corresponds to the "used" component values of set C_i. Since values of a given component are unique across tuples, the nodes in each C[i] point to every tuple in D exactly once. Because the example from Figure 2 will be referenced frequently, specifically Q(t_6), Figure 4 shows the contents of C for this D.

The auxiliary routine **next_candidate** should be explained before proceeding with the discussion of **Q_t_NI**. **Next_candidate** "filters out" those tuples which can not, by definition, be in Q_t(v) because they do not **precede** v. Specifically, **next_candidate** returns the next tuple which both **precedes** v and has c_i less than that of the "current" tuple. This is done by searching down C[i] from the current value until some value is found whose associated tuple **precedes** v. For instance, if one were evaluating Q(t_6) and had just inspected t_6 itself along component 2, **next_candidate**(2, t_6) would skip over t_4 because $t_4 \not\prec t_6$ and would go on to return t_3.

The approach taken by **Q_t_NI** is to find a single tuple which is in Q_t(v), then to make use of the observation in LEMMA 1 that all other tuples in Q_t(v) must cross that "seed" tuple. Step by step, the algorithm proceeds as follows to resolve Q_t(v):

- Find a seed tuple which is known to be in Q_t(v). This tuple is called t_X, and is the tuple **preceding** v which has the highest c_1 value. It is found by scanning down C[1] for decreasing values of C_1, starting at $v.c_1$ (i.e. $t_X \prec v$ and no tuple can be between t_X and v).
- Find all tuples which both **precede** v (are admissible) and "cross" t_X as described in LEMMA 1. These tuples are then added to the query result list, Qlist. They are found by scanning, down each component i, for tuples whose c_i value is between

---

D = {t_1=⟨2,1,5⟩, t_2=⟨3,3,3⟩, t_3=⟨5,5,4⟩,
t_4=⟨4,6,8⟩, t_5=⟨6,2,2⟩, t_6=⟨8,7,6⟩}

C = {
{⟨8, t_6⟩, ⟨6, t_5⟩, ⟨5, t_3⟩, ⟨4, t_4⟩, ⟨3, t_2⟩, ⟨2, t_1⟩},
{⟨7, t_6⟩, ⟨6, t_4⟩, ⟨5, t_3⟩, ⟨3, t_2⟩, ⟨2, t_5⟩, ⟨1, t_1⟩},
{⟨8, t_4⟩, ⟨6, t_6⟩, ⟨5, t_1⟩, ⟨4, t_3⟩, ⟨3, t_2⟩, ⟨2, t_5⟩},
}

Figure 4. Example of Q_t_NI Data Structures

---

* Specifically, threaded AVL trees[8][15] are used.

those of $v.c_i$ and $t_X.c_i$ (note: this decreasing-value scan is important for proof of the algorithm). Any such tuple, let us say **candidate**, which **precedes** v must cross t_X for two reasons. First, **candidate** does have a c_i greater than $t_X.c_i$. Second, **candidate** can not have every c_i greater than $t_X.c_i$ because t_X has the highest c_1 value of any tuple **preceding** v. Thus, **candidate** crosses t_X.

- Filter out those tuples which violate the "$\nexists u \in D\ [t \prec u \prec v]$" clause of Q's definition. This is done on a per-tuple basis, before the tuple is added to **Qlist**.

We now show how **Q_t_NI** operates on the example from Figure 2, $Q(t_6)$. The first action taken by **Q_t_NI** is to find t_X. **Q_t_NI** finds $t_6.c_1$, 8, then scans down the values of c_1 until it finds the first tuple $t_X \prec t_6$. This tuple is t_5. **Q_t_NI** next goes into a component-scanning loop. It inspects the tuples associated with values between 7 and 2 on c_2 (that is, between $t_6.c_2$ and $t_5.c_2$), and then the tuples associated with values between 6 and 2 on c_3 ($t_6.c_3$ and $t_5.c_3$).

When scanning c_2, **Q_t_NI** first encounters t_4 but skips over that tuple (in **next_candidate**) because it is inadmissible — it does not **precede** t_6. **Q_t_NI** then encounters t_3, finds it admissible, and adds it to **Qlist**. The last tuple encountered in the c_2 scan is t_2. This is not added to **Qlist**, even though it **precedes** t_6, because it also **precedes** t_3 which is already in **Qlist**. The first tuple encountered in the scan of c_3 is t_1, which is added to **Qlist**. After that, t_3 is encountered a second time and not added to **Qlist** since it is already there. Finally, t_2 is encountered again and not added to **Qlist** for the same reason as before.

Update of the **Q_t_NI** data structures when adding a new tuple t to D (which **Q_t_NI** only accesses indirectly through tuple pointers) requires the insertion of the values of each of t's components into the corresponding C[i] sorted sets (search trees). It is for this update operation that sorted sets are used in C instead of lists. While list traversal would be slightly faster than thread traversal during the down-scans of the values in each C[i], sorted sets are distinctly more efficient than lists when updating each C[i] with the value from a new tuple.

3.3 Proof and Analysis

The proof of **Q_t_NI**, as for other algorithms returning sets, has two obligations. First, it must be shown that **Q_t_NI** never returns tuples not permitted by the definition of Q_t. Second, it must be shown that **Q_t_NI** returns all tuples required by the definition of Q_t.

Proof of the first obligation is itself in two parts. Part 1 shows that no tuple t is added to **Qlist** such that $\exists u \in$ **Qlist** $[t \prec u]$, and Part 2 shows that no tuple t is added to **Qlist** such that $\exists u \in$ **Qlist** $[u \prec t]$. Part 1 derives from inspection of **Q_t_NI** which reveals that all tuple additions to **Qlist** are guarded with this particular invariant. We demonstrate Part 2 by contradiction. If $u \prec t$, the value of every component in t must be greater than that of the corresponding component in u. **Candidate** tuples to be added to **Qlist** are chosen during a descending scan of component values. This contradicts the possibility that $u \prec t$: since t is a **candidate** for addition to **Qlist** after u was added, at least one of t's components must have a value less than that of the corresponding component in u — somehow, u was encountered first, but it could not have been.

To begin the second proof obligation, it is known that $t_X \in$ Q_t(v) because both $t_X \prec v$ and there is no tuple $u \in D$ between t_X and v. By LEMMA 1, therefore, every other tuple $t \in$ Q_t(v) must have the value of at least one component greater than that of the corresponding component in t_X, and also must have the value of at least one component less than that of the corresponding component in t_X. Additionally, the value of no component of a tuple t may be greater than that of the corresponding component in v, since it is required that $t \prec v$.

The conjunction of these requirements is that every $t \in$ Q_t(v) must have at least one component c_i such that $t_X.c_i < t.c_i < v.c_i$ and another component c_j such that $t.c_j < t_X.c_j$. The c_j clause is guaranteed on c_1 by the way in which t_X is determined and the fact that no tuples are added to **Qlist** which do not **precede** v. The range of component values in the c_i clause is exactly what **Q_t_NI** searches after it determines t_X. During this search, the only tuples which are not added to **Qlist** are those which are not allowed by the definition of Q_t. Specifically, these are any tuples which do not **precede** v and any which **precede** some tuple already in **Qlist**. Thus, given this selection procedure in **Q_t_NI**, all tuples which should be in Q_t(v) are added to **Qlist**.

Q.E.D.

For analysis, it is useful to define $r \equiv |\text{Q\_t}(v)|$ and $b \equiv |\{ t \mid t \in D \wedge \exists i [t_X.c_i < t.c_i < v.c_i] \}|$ (that is, the number of tuples with the value of at least one component between the values of the corresponding components of t_X and v). **Q_t_NI** visits each of the k_t components exactly once by performing a scan down the appropriate C[i]. Each of these scans can examine the values of up to b tuples' components. Now, for every value which is associated with an admissible tuple, that tuple will potentially need to be compared with r tuples in **Qlist**. Considering that a tuple compare takes $O(k_t)$ time and remembering that the sorted set operations used by **Q_t_NI** to scan each component are initiated with a single $O(\log d)$ search

followed by a sequence of $O(1)$ thread traversals, we arrive at an $O(k_t^2 br + k_t \log d)$ overall time complexity for **Q_t_NI**.

A simple optimization, however, can substantially reduce this time. Much of the above effort is spent performing comparisons between tuples which have already been examined with respect to either **Qlist** or v. By marking each tuple as it is examined, those repeated comparisons can be eliminated (though **Q_t_NI** must still scan over the marked tuples). This results in an $O(k_t(b+br+\log d))$ time for **Q_t_NI**, of which $k_t br$ is the remaining comparisons, $k_t b$ is the admissibility checks, and another $k_t b$ is scanning past already-marked tuples.

Update of the **Q_t_NI** data structures requires simple $O(\log d)$ component value insertions into each element of **C**. There are k_t elements in C, so the total update time is $O(k_t \log d)$.

3.4 An Incremental Algorithm

One might notice that three categories of tuples are encountered during the component-value scans made by **Q_t_NI**: tuples which cross t_X, tuples which **precede** some tuple already in **Qlist**, and tuples which cross v. Tuples in all but the first category are ignored. Can this discarded information be used for subsequent queries?

The answer is a qualified "yes." Intuition suggests that the tuples which are found to cross v would be prime candidates for inclusion in **Qlist** during subsequent queries in which v takes the role of t_X. This might alleviate the problem of encountering a tuple multiple times during the component scans, since the results of a previous series of scans are already known.

Specifically, the available information is a list of all tuples which cross v and have the value of at least one component between those of the corresponding components in v and t_X. What is needed to resolve a subsequent query $Q\_t(v_2)$, for which v takes the role of t_{X_2}, is all tuples which cross v and for which the value of at least one component is between those of the corresponding components in v and v_2. The cross-v tuples for a range of component values below v is known, but what is needed is the cross-v tuples for a range above v.

If one were asking queries only of the form of $Q\_t(v)$, this situation would present a considerable problem. For the monitor application, however, not only is the query $Q\_t(v)$ made, but also the query $Q\_t_>(v)$ is made. $Q\_t_>(v)$ finds the minimal elements of the set of tuples in D which **follow** v, instead of the maximal elements of the set of tuples in D which **precede** v

($\mathbf{Q\_t}(v) \equiv \mathbf{Q\_t}_{\prec}(v)$). The evaluation of $\mathbf{Q\_t}_{\succ}(v)$ can provide the component value range above v which is needed for the evaluation of subsequent $\mathbf{Q\_t}(v_2)$ queries.

Figure 5 shows how the cross-v tuple information might be used. In Figure 5a, we see that the tuple $t_{X\succ}$ delimits the range of component values scanned during a $\mathbf{Q\_t}_{\succ}(v)$ query. During this scan, t_c was found to cross v. Later, as shown in Figure 5b, v_2 is added to the database and we wish to resolve $\mathbf{Q\_t}(v_2)$. The t_{X_2} for the new query is v. Because of the results from $\mathbf{Q\_t}_{\succ}(v)$, it is known that the only tuple which crosses v in the highlighted component range between v and $t_{X\succ}$ (and thus might be in $\mathbf{Q\_t}(v_2)$) is t_c. The evaluation of $\mathbf{Q\_t}(v_2)$ does not need to scan component values in that range.

There are several problems with this approach even before the difficulties of updating v's crossover list are considered. Observe in Figure 5b that $t_{X\succ}$ crosses v_2. This, or that $t_{X\succ} \succ v_2$, is guaranteed to occur for every query. Otherwise, $t_{X\succ}$, not v, would be t_{X_2}. For components such as x in Figure 5b, this results in a wider range of some component values being scanned than is necessary. While such a $t_{X\succ}/v_2$ crossing can lead to the wasteful comparison of v_2 with cross-v tuples which do not **precede** v_2 (whenever $t_{X\succ}.c_i > v_2.c_i$), this only increases the time complexity

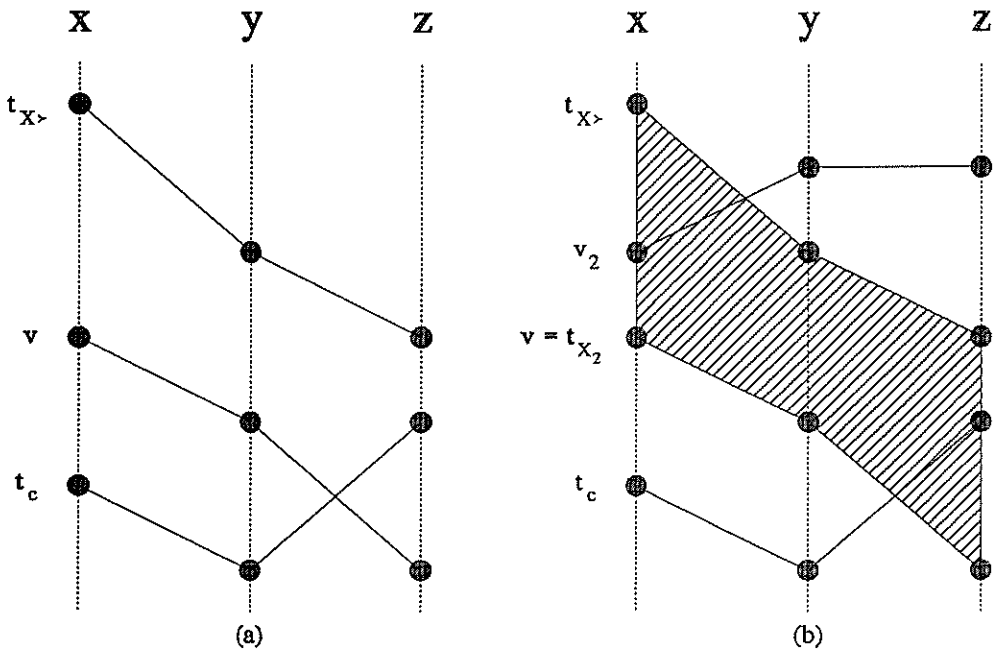


Figure 5. Example for Incremental Total-Order Query Algorithm

of the algorithm by effectively making b larger. What does significantly increase the algorithm's time complexity is that, whenever $t_{X>}.c_i < v_2.c_i$ (as with components y and z in Figure 5b), too narrow a range of component values is scanned than necessary. When this occurs, the algorithm must find some means to finish scanning the "gaps" in the component range between v_2 and v.

There appear to be two approaches to remedy this problem. The first is to perform a search for some alternative to v, let us say v', down each component for which the range of values covered by $t_{X>}$ is insufficient. Cross-v' tuples would then be checked for addition to Q_t(v_2). A second approach is to revert to the non-incremental algorithm Q_t_NI for the troublesome components. Both of these approaches raise the possibility of double-checking cross-v tuples, especially if the $t_{X>}'/v'$ and $t_{X>}/v$ ranges overlap. This means that the overall time complexity of the proposed incremental algorithm can be no better than that of Q_t_NI, though the incremental algorithm will have increased storage requirements and a substantially larger constant multiplier factor in the time complexity.

Update of the incremental algorithm's data structures when a new tuple v is added to D appears best performed by a 2-dimensional range query on each component. This query would find those existing tuples whose "scanned" range includes v. A total of k_t component-range searches is needed, each requiring $O(\log d)$ time. Following each of these is a series of $O(k_t)$ tuple comparisons to determine if v crosses any of the tuples found in the component-range search (along with marking tuples to avoid double-checking, etc.). It is expected that the number of tuples found in each component-range search will be on the order of b. So, the new complexity terms required in order to update the "tuple crossing" lists do not appear to add to the time complexity of the incremental algorithm since Q_t_NI already requires comparable terms. However, the time required to update the component-range structure with each of v's components has a worst-case time of $O((\log d)^2)$ and expected time of $O(\log d)$, putting the incremental algorithm's total worst-case update time at $O(k_t(\log d)^2)$.

Such an update time can easily become appreciable, adding to the algorithm's already considerable constant-multiplier time penalty and increased storage space. For these reasons, further investigation of the incremental algorithm described above has been suspended in favor of the following alternative non-incremental algorithm.

3.5 An Algorithm Using Range Query Techniques

Q_t_ORQ, like **Q_t_NI**, builds **Qlist** by examining all tuples which have the value of at least one component between the values of the corresponding components of v and t_X. Unlike **Q_t_NI**, **Q_t_ORQ** finds that set of tuples through use of a sequence of orthogonal range queries instead of through direct traversal of search tree threads. It is known that general-purpose orthogonal range queries* are quite costly for any application requiring more than a small number of search dimensions. Pleasantly, for the monitor application, any Q(v) usually contains no more than two or three total-order factors. Given this, a range query approach to **Q_t** is feasible.

Figure 6 shows how **Q_t_NI** is modified to use range queries. **Q_t_ORQ** begins by finding t_X in the same manner as **Q_t_NI**. Note that the C search tree is needed only for c_1; D's range query structure performs the role of C[2] through C[k_t]. For each iteration of i from 2 to k_t, a range query (whose format is specified below) is performed instead of a search through C[i]. The tuples returned from those queries are checked for admissibility** and, as in **Q_t_NI**, are checked to make sure that they do not **precede** any tuple already in **Qlist**. If both conditions hold, the new **candidate** tuple is added to **Qlist**.

For each component c_i other than c_1, a range query is issued by **Q_t_ORQ**. As would be expected, the range specified for c_i is those values between $t_X.c_i$ and $v.c_i$. For components c_j "before" c_i ($j < i$), **Q_t_ORQ** needs to exclude tuples it has already examined and so specifies those component values less than $t_X.c_j$. The only reason to specify values for components "after" c_i would be to ensure that each tuple returned by the query **precedes** v. Since it is faster to just explicitly check each tuple against v than to add more specifications to the query, the trailing components allow the full range of component values.

The leading components of c_i's range query are, except for the change in the range specification of c_i, the same as those of c_{i+1}. Because the queries have these range specifications in common, it is possible to retain the intermediate evaluation data for c_i's query and use it to

---

* Unlike the optimized equivalence-factor algorithm in Chapter 2, Q_t_ORQ must use the full power of a general-purpose orthogonal range query.

** It is possible to avoid the comparison against v by explicitly specifying the range of the trailing components in the range query (expressly, use ..$v[i+1]$, ..., ..$v[k_t]$). Putting this check in RQ, however, takes more time than the comparison against v.

```
constants
    k_t : natural := the number of (total-order) components in a tuple;

types
    tuple = array [1..k_t] of integer;
    C_node = record
        val : integer;
        t :   ^tuple;
    end C_node;

globals
    C_1 : srt_set of C_node key val;        //  unlike with Q_t_NI, D handles the other components
    D :  range_query_structure of tuple;

//  Similar to that of Q_t_NI before, but only applies to C_1
//
function next_candidate(bound : tuple) : ^tuple;


function Q_t_ORQ(v : tuple) : list of ^tuple;
    Qlist : list of ^tuple := {};
    candidate : ^tuple;
    t_X : tuple;                            //  inspect all tuples with component values between v and t_X
    i :  natural;                           //  the tuple component on Q_t_ORQ is working
begin
    t_X := C_1[v[1]]→t;                     //      Find the first tuple whose c_1 value is less than
    t_X := next_candidate(v);               //  that of v.  All other tuples put into Qlist will "cross" t_X.
    if t_X ≠ null then                      //  make sure there is at least one tuple ≺ v
        Qlist += t_X;
        for i in [2..k_t] do
            //  scanning down values of c_i
            //
            for candidate ∈ RQ(D, ..t_X[1], ..., ..t_X[i-1], t_X[i]..v[i]) do
                if candidate^ ≺ v andif ∄u∈ Qlist [candidate^ ≼ u^] then
                    Qlist += candidate;
                endif;
            endfor;
        endfor;
    endif;
    return Qlist;
end Q_t_ORQ;
```

Figure 6. Query by Total-Order Factors (Using Range Queries)

"pre-load" c_{i+1}'s query. Given this optimization, the time to evaluate the k_t-1 range queries in **Q_t_ORQ** is comparable to the time to evaluate only two queries, not k_t-1 queries.

All time and space costs for **Q_t_NI** could be expressed in terms of a single $Q_j \in Q$. Unlike **Q_t_NI**, **Q_t_ORQ** directly references D. Therefore, though some of the **Q_t_ORQ** costs are in terms of a single query, some must be in terms of Q. For this purpose, as with Section 2's equivalence factor algorithm, it is useful to define $K_t \equiv$ *the number of distinct total-order factors across all* $\mathbf{Q} \in Q$. As done with m_e, consider only the total-order factors of the queries in Q and let $m_t \equiv \max_{1 \le i \le K_t} |C_i/D|$, where C_i/D is that subset of C_i induced by application of F_i over the tuples in D (i.e. the number of unique C_i values actually used when evaluating queries over D).

Analysis yields a time complexity for **Q_t_ORQ**(v) of $O(k_t(b+br)+(\log d)^{k_t-1})$, derived as follows. The time for checking whether or not a scanned tuple should be put into **Qlist** is $k_t br$, the same time as for **Q_t_NI**. The $k_t b$ admissibility check is also the same as for **Q_t_NI**. Range query time is $b + (\log d)^{k_t-1}$, which compares to the search time of $k_t(b + \log d)$ for **Q_t_NI**. This time appears favorable for small k_t, but must be weighed against the very high memory cost of the range query data structures and the fact that, for small k_t, the $k_t br$ term common to both algorithms might tend to dominate.

Update of the **Q_t_ORQ** data structures requires insertion of each new v into C_1 and into D's range query data structures, an operation requiring $O((\log d)^{K_t})$ worst-case time and $O((\log d)^{K_t-1})$ expected time. Since this update applies to all $\mathbf{Q} \in Q$, the time compares to $O(qk_t \log d)$ worst- and expected-case time for **Q_t_NI**. Space cost for **Q_t_ORQ** is a substantial $O(d(\log m_t)^{K_t-1})$.

Q_t_ORQ would become much more economical if a restricted-domain D_j were searched for each Q_j, thus making $K_e = k_{e_j}$. This is exactly what happens when the equivalence and total-order factors of a query are re-combined as described in the next subsection.

3.6 Queries With Both Equivalence and Total-Order Factors

As described in Section 2, the equivalence factors of a query Q_j are evaluated through use of a highly-optimized range query structure which maintains auxiliary trees only for the leaf nodes of each search dimension. The set of tuples at the leaves of the last set of auxiliary trees have the same values for all equivalence factors. Quite literally, D has been partitioned not only by each query (as desired in the previous subsection) but also by the values of that query's equivalence factors.

Evaluation of total-order factors fully exploits this fine-grained database partitioning. For each query in Q, the "database" to which either **Q_t_NI** or **Q_t_ORQ** is applied is not D, but is rather one of these auxiliary tree leaves' tuple sets. There is one **Q_t_NI** or **Q_t_ORQ** structure per set of equivalence factor values per query. Time and space costs for total-order factor evaluation are significantly reduced. Intuitively, one can view the total-order factor data structures as one additional level of auxiliary trees in the equivalence factor structures.

4. Equivalence-Factor Query Optimization

4.1 Requirements

Generally, a query $Q_r(v)$ has several equivalence factors in addition to a small number of total-order factors. Other queries, let us say $Q_s(v)$, might have one or more equivalence factors in common with $Q_r(v)$. As mentioned in Section 2, sharing intermediate search results between queries with common equivalence factors would save both time and space when resolving a set Q of queries. The topic of this section is to find how best to optimize the equivalence factor search structures for a set of queries so that intermediate results can be shared between queries.

Since total-order factors are not involved in these optimizations, they are ignored for the rest of this section. Furthermore, it is found useful to abstract the conjunction of equivalence factors in a query into a sequence of tokens (letting adjacency indicate conjunction). For example, suppose that $Q_r(v)$ has three equivalence factors. These can be abstracted as **abd** (using letters from the beginning of the alphabet for the tokens). If $Q_s(v)$'s equivalence factors were abstracted as **cab**, we would know that $Q_r(v)$ and $Q_s(v)$ have two factors in common, **a** and **b**.

Continuing this example, how can intermediate results be shared between the resolutions of $Q_r(v)$ and of $Q_s(v)$? First, it must be realized that since the factors in a query are part of a conjunction, their order commutes. $Q_r(v)$ and $Q_s(v)$ can thus also be represented as **bad** and **bac**, respectively. The **ba** "prefix" is common between the queries and is the basis for the shared information. Now, the query tree structure for $Q_r(v)$ contains a base tree for **b** and auxiliary tree levels for **a** and **d**; the structure for $Q_s(v)$ contains a base tree for **b** and auxiliary tree levels for **a** and **c**. To optimize the resolution of the pair of queries, their base trees and first level of auxiliary trees are combined. Only the second level of auxiliary trees remains distinct between the queries. During resolution, the search for those tuples for which both factor **b** and factor **a** are true is only performed once. The results of this search are then used separately to resolve $Q_r(v)$ and $Q_s(v)$ according to factors **d** and **c**, respectively. Figure 7 illustrates such a combined query structure. This practice saves space whenever a tuple is in the domain of both queries, and always reduces combined update and query time.

In order to represent query structures more succinctly for optimization purposes, we introduce one more slight abstraction of the problem. Let the k_{e_j} tree levels in a query Q_j's data structure be represented as an undirected path of length k_{e_j} with vertices labeled by the equivalence factors in Q_j. The unoptimized query structure for a query set Q, therefore, is a forest

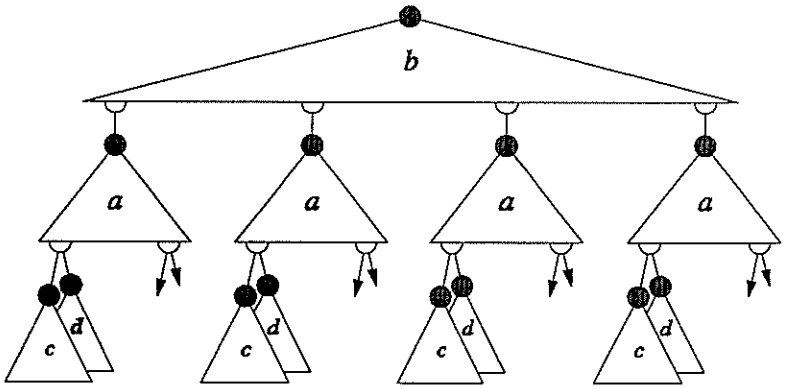


Figure 7. Combined Structure for Two Queries

of q disjoint paths. Optimization of this query structure (if optimization is productive, of course) turns the forest of paths into a forest of trees, where the root vertex of each subtree represents an equivalence factor common between two or more queries. Each query in Q is still represented by a path originating* at the root of some tree in this forest. Figure 8 uses this notation to show the optimization steps for the $\mathbf{Q}_r(v)/\mathbf{Q}_s(v)$ example.

It is not always possible for each equivalence factor to be represented exactly once in an optimized forest. Consider three queries with the factors **ab**, **bc**, and **ca**. Optimization results in some forest isomorphic (discounting vertex labels) with that illustrated in Figure 9. No matter how the forest is constructed, at least two of **a**, **b**, or **c** will be duplicated. To evaluate all the queries in Q, those factors will be encountered as a search dimension more than once.

The query structure optimization problem is formally defined in terms of sets of tokens and a forest of trees. It is stated as a decision problem to aid in a proof in the next subsection.

QOPT. Given an alphabet E, a set Q of sets $\mathbf{Q}_j$ of tokens over E, and a positive integer f. Is there a forest F with the following properties such that the number of vertices in F is no greater than f?

1) Each vertex in the forest is labeled (not necessarily uniquely) with one token $\mathbf{e} \in E$.

2) For each $\mathbf{Q}_j \in Q$, there exists a path $\mathbf{p}_j$ with these two characteristics. First, the origin of $\mathbf{p}_j$ is the root of some tree T in F.

* Since the paths are undirected, origin and terminus are notational conveniences in this discussion. For a query's data structure itself, however, the "origin" of a path has meaning since that identifies the structure's base tree.

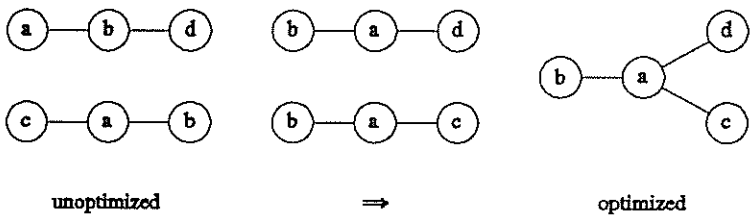


Figure 8. Optimization of Equivalence-Factor Search Structures

Second, the labels of the vertices on $\mathbf{p}_j$ enumerate the tokens in set $\mathbf{Q}_j$ and no tokens not in $\mathbf{Q}_j$.

3) Given the roots of two trees in F, those vertices are labeled with distinct tokens. Similarly, given two subtrees in F which share the same parent vertex, the roots of those subtrees are labeled with distinct tokens.

4.2 NP-Completeness Analysis

Unfortunately, it proves to be the case that the above optimization problem is NP-complete. QOPT turns out to be a multi-level version of the HITTING SET problem as defined in [9] and [5]:

HITTING SET. Given a set U of subsets of a finite set S and a positive integer $h \leq |S|$. Is there a subset $S' \subseteq S$ with $|S'| \leq h$ such that S' contains at least one element from each subset in U?

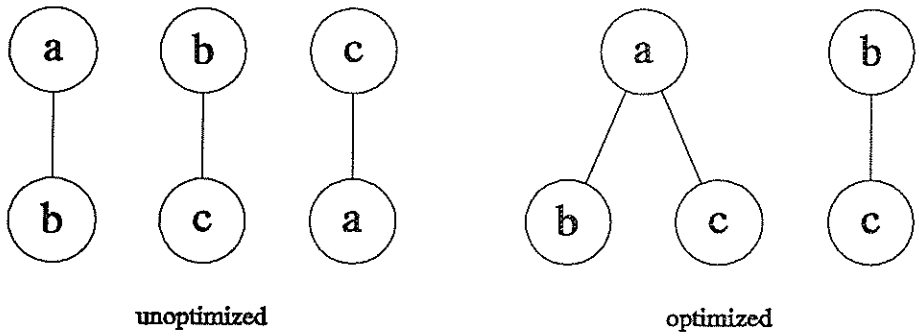


Figure 9. Optimized Query With Duplicate Equivalence Factors

It will be shown that the HITTING SET problem, subject to the restrictions that $\forall \mathbf{U}_j \in U\ [|\mathbf{U}_j| = 2]$ and that no two $\mathbf{U}_j$ and $\mathbf{U}_j'$ are identical, can be solved in polynomial time by any algorithm which solves the QOPT problem subject to the restrictions that $\forall \mathbf{Q}_j \in Q\ [|\mathbf{Q}_j| = 2]$ and that no two $\mathbf{Q}_j$ and $\mathbf{Q}_j'$ are identical. HITTING SET remains NP-complete with these restrictions[5]. To show that QOPT is NP-complete, we must demonstrate that a) it is in NP, and that b) an algorithm which solves QOPT can solve a known NP-complete problem in polynomial time. Let us define $q \equiv |Q|$, $K_e \equiv |E|$, $s \equiv |S|$, $u \equiv |U|$, and $|F| \equiv$ *the number of vertices in F.*

For part a) of this obligation, consider a nondeterministic Turing machine with two functional blocks: a forest generator, and a recognizer. The forest generator accepts E and f, then nondeterministically generates a forest with f vertices, each labeled with some token from E. The NTM can do this in polynomial time.* F, the output of the forest generator, is passed to the recognizer, which also accepts Q. By traversing all paths in F which originate at the root of some tree in F, the recognizer generates a set Q' of token sets with which F is related according to Property 2 in the definition of QOPT. This can be done in deterministic $O(f^2)$ time. Finally, the recognizer compares Q' to Q (deterministic $O(q^2K_e)$ time) to find if $Q \subseteq Q'$. If so, F is a tree of size f which is related to Q (not only to Q') by Property 2 of QOPT. The NTM has decided QOPT in polynomial time, so QOPT is in NP.

Part b) of the proof obligation is shown by proving that there exists a one-to-many mapping to S' from the tokens of the roots of the trees in F. Let QOPT's Q and E correspond to HITTING SET's U and S, respectively. Define E_R as the set of tokens associated with the roots of the trees in F. In QOPT, the enumeration paths $\mathbf{p}_j$ are defined to originate at the root of one of the trees in F. Therefore, E_R contains at least one token from each enumeration path and thus one token from each $\mathbf{Q} \in Q$: E_R corresponds to S'. The question now becomes, "how are f and h related?".

One of the restrictions above guarantees that no two $\mathbf{Q}_j$ and $\mathbf{Q}_j'$ are identical. This implies that no two $\mathbf{Q}_j$ and $\mathbf{Q}_j'$ have coincident enumeration paths; the paths can share the same origin, but they can not share the same terminus. The size of every $\mathbf{Q} \in Q$ is 2; E_R contains the distinct label of the origin of every path — the only other vertices in F are the termini, and they are all

---

* A variety of forest-generation algorithms are available. One such algorithm is used in the test-case generators which provide sample Q sets to implementations of QOPT_TF and QOPT_SO.

unique. $|F| = |\textit{the path origins}| + |\textit{the path termini}| = |E_R| + q$. In other words, when QOPT specifies a value for f, it is specifying a value of $f - q$ for $|E_R|$.

It is now trivial to solve a HITTING SET problem by solving a QOPT problem. Let $Q = U$, $S = E$, and $f = h + q$. Use E_R as a solution to S'; $|E_R| \leq h$.

Q.E.D.

Let QOPT and HITTING SET now be treated as optimization, not decision, problems. A major difference between the definitions of QOPT and HITTING SET is that HITTING SET is concerned only with minimizing a set which intersects (at least) a single element in each $\mathbf{U} \in U$. QOPT is concerned with minimizing, in a way, a group of sets (each "level" of each tree) which must cover every element in each $\mathbf{Q} \in Q$. Consider attempting to solve a QOPT problem by recursively applying a series of HITTING SET problems to each level of F. When the token sets in Q have many tokens in common but have a small number of differentiating tokens, the recursive HITTING SET will fail. Such a situation is illustrated in Figure 10, where **t** and **f** differentiate several "pairs" of sets which have other tokens in common. Recursive HITTING SET would choose to use **t** and **f** for the tree roots. QOPT, concerned with total tree size, would recognize the other common tokens, root the trees with those, and put **t** and **f** at the trees' leaves. It is for this reason that the size of the sets in U and Q were restricted to 2 in the above NP-completeness proof; this discrepancy does not arise with such small sets.

4.3 General Heuristics

There are some basic reductions can be made in order to narrow the scope of the QOPT problem. These involve collapsing queries with duplicate equivalence factors, partitioning the

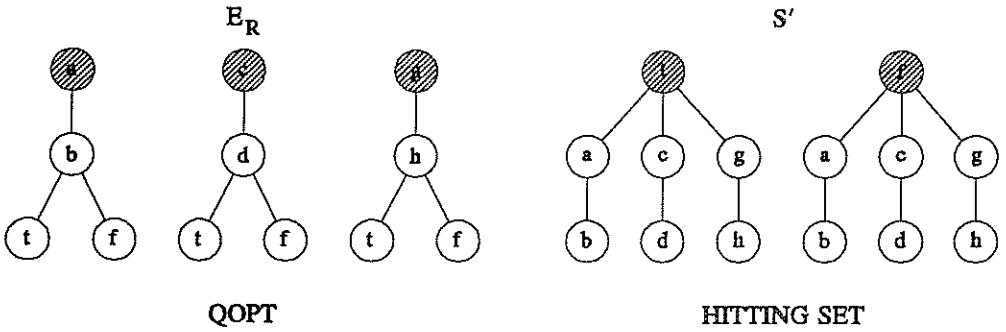


Figure 10. Optimization Differences Between QOPT and Recursive HITTING SET

	original	reduced
1)	Q = {{a b c}, {c d e}, {b a c}}	Q = {{a b c}, {c d e}}
2)	Q = {{a b c}, {d f h}, {c e}, {f h k}}	Q_1 = {{a b c}, {c e}} Q_2 = {{d f h}, {f h k}}
3)	Q = {{a b c}, {b e c}, {e d h}, {d h f}}	Q = {{a b}, {b e}, {e d}, {d f}}
4)	Q = {{a b c}, {b d e}, {b f g}}	root b, Q' = {{a c}, {d e}, {f g}}

Figure 11. Examples of Basic Equivalence-Factor Query Set Reductions

problem so that the queries have some equivalence factors in common, collapsing "mated" equivalence factors, and processing any common equivalence factor as a special case. Stated in terms of the QOPT definition above, the reductions are, respectively:

1) Collapse equivalent sets Q_j and Q_j' into a single set.

2) Partition the set Q into subsets Q_p so that each $Q_r \in Q_p$ has at least one token in common with some $Q_s \in Q_p$, $Q_r \neq Q_s$. Apply the optimization algorithm separately to each Q_p.

3) Collapse any tokens e_x and e_x' such that $\forall Q_j\ [e_x \in Q_j \equiv e_x' \in Q_j]$ into a single token.

4) For any token e_c such that $\forall Q_j \in Q\ [e_c \in Q_j]$, use e_c as the root of a tree common to all paths p_j, construct $Q' = \{Q_j - e_c \mid Q_j \in Q\}$, and apply the optimization algorithm recursively to Q'.

Examples of each of these reductions is shown in Figure 11. Their use should enhance the performance of the following two approximate-optimization algorithms.

4.4 Token-Frequency Algorithm

Our first approximate-optimization algorithm, **QOPT_TF**, follows the line of intuition which suggests that the token occurring most frequently in Q should be the root of a tree in F. Though this intuition is often correct, the previous example in Figure 10 illustrates one scenario

```
types
    token = an identifier (in our case, for an equivalence factor);
    token_set = set of token;          //  here, implement as a list of values
    tree_node = record
        e : token;
        children : list of ^tree_node;
    end tree_node;

globals
    F : tree_node;                     //  root of the query forest QOPT_TF constructs
    Q : set of token_set;              //  the original query set

//  Add a new child to node, labeling this child with token.
//
function add_child(node : ^tree_node, child : token) : ^tree_node;
    child_node : ^tree_node;
begin
    child_node := new tree_node;
    child_node→e := child;
    child_node→children := [];
    node→children &= child_node;
    return child_node;
end add_child;



procedure QOPT_TF(F' : ^tree_node, Q' : set of token_set);
    e_high : token;
    Q_high : set of token_set;
    new_child : ^tree_node;
begin
    e_high := the token occurring most frequently as an element of each Q ∈ Q';
    Q_high := { Q | Q ∈ Q' ∧ e_high ∈ Q };
    new_child := add_child(F', e_high);
    QOPT_TF(new_child, { Q - e_high | Q ∈ Q_high });
    QOPT_TF(F', Q' - Q_high);
end QOPT_TF;
```

Figure 12. Token-Frequency Query Optimization Algorithm

Q = { {a, m, x}, {a, n, x}, {a, o, y}, {a, p, y}, {b, q, x}, {c, r, y} }

Token-frequency selects **a** and one each from {**b, q, x**} and {**c, r, y**} to be roots, leading to 13 tree nodes.

Optimal selects **x** and **y** as roots, resulting in 12 tree nodes.

Figure 13. Example of Non-Optimal Result From Token-Frequency Algorithm

in which **QOPT_TF** generates a less-than-optimal query tree. Another example in which QOPT_TF leads to inefficiency is shown in Figure 13.

Nonetheless, **QOPT_TF** executes quickly and is readily understood. A simple tree node data structure is defined in Figure 12 and a recursive implementation of QOPT_TF is shown which operates on that structure. Execution begins with a call of "QOPT_TF(&F, Q)", referencing the top-level forest and the original query set. QOPT_TF has four basic steps:

- Find the token, e_{high}, which occurs most frequently in the sets making up Q'. Determine every set in Q' which contains e_{high}. This is Q_{high}.
- Add a new child to F'. Label this child with e_{high}.
- Apply **QOPT_TF** to the new child, optimizing the queries which contained e_{high} (less e_{high}, of course, so that there is no endless loop).
- Apply **QOPT_TF** again to F', this time for all the sets in Q' which did not contain e_{high}.*

QOPT_TF has a relatively low time complexity. Finding e_{high} dominates the time in each invocation. Assuming that **token_sets** are represented as lists instead of as bit vectors, this scan takes $O(q k_{e_{max}} + K_e)$ time, letting $k_{e_{max}} = \max_{Q_j \in Q} |Q_j|$, or just $O(q k_{e_{max}})$ since K_e is no larger than required for the case of unique tokens in each $Q_j \in Q$.** After determining e_{high}, an $O(1)$ **add_child** takes place and then two recursive calls are made. It is critical for the analysis to observe that, for this next level of recursion, time for the two immediate e_{high} scans will be

* Alternatively, this could be implemented as a loop instead of as a second recursive call.

** Even for deeper levels of calls to QOPT_TF where q and $k_{e_{max}}$ change, the effective K_e' will still be no greater than $q k_{e_{max}}$ because a linked list of occurring tokens can be made during the frequency scan, eliminating the need to maximize token frequency across each value of the original E.

$O(|Q_{high}|k_{e_{max}})$ and $O((q-|Q_{high}|)k_{e_{max}})$, for a total of still just $O(qk_{e_{max}})$. The question now becomes "how many levels deep can **QOPT_TF** recurse?"

The worst case occurs when all tokens are unique in each $Q_j \in Q$. In that case, only one Q_j will be placed in Q_{high} for each level of recursion. So, after q levels of recursion, there will be at most one element in Q', allowing **QOPT_TF** to trivially finish that branch of the forest in $O(k_{e_{max}})$ time. In total, there are $O(q)$ levels, each at $O(qk_{e_{max}})$ time, plus at worst q $O(k_{e_{max}})$ "leaf" branches, for an aggregate $O(q^2k_{e_{max}})$ time for **QOPT_TF**. This is a tight bound, since the unique-tokens case does reach it.

4.5 Set-Overlap Algorithm

Another line of intuition about how to approach QOPT suggests that those sets which have the most elements in common should be combined into one tree in F. While this algorithm, **QOPT_SO**, does take somewhat more time than **QOPT_TF**, it also produces significantly better results in most test cases.

The revised **tree_node** data structure is shown in Figure 14. Whereas **QOPT_TF** combines several sets by a single token they have in common, **QOPT_SO** combines just two sets at a time by all the tokens that pair has in common. These tokens in common between the sets which were merged to create the node are the node's **leading** tokens; the tokens which are not common between the node's constituent sets make up the node's **children**. In **QOPT_TF**, the basic operation on a **tree_node** is adding a single token as a child; in **QOPT_SO**, the basic **tree_node** operation is combining the **leading** tokens and **children** of two nodes.

QOPT_SO, shown in Figure 15, begins with a call of "QOPT_SO(F, Q)". Initialization involves building separate, uncombined trees for each $Q_j \in Q$ and building a table (**common**) whose values contain the number of tokens in common between each of the uncombined trees. For each **tree**[j], **QOPT_SO** keeps track of the other **tree** with which tree[j] has the most tokens in common. This is done by means of the **ordered_common**[j] heap, keyed by the number of common tokens. We keep track of the **tree** with the most tokens in common with any other **tree** by means of another heap, **max_common**.

The main loop of **QOPT_SO** combines two trees with each pass. Its operation proceeds in three primary phases:

```
types
    token = an identifier (in our case, for an equivalence factor);
    token_set = set of token;          //  here, implement as bit vectors
    tree_node = record
        leading :  token_set;          //  tokens in common between all children
        children : list of ^tree_node;
    end tree_node;
    common_node = record
        in_common : natural;           //  # of leading tokens in common between this tree and tree[idx]
        idx : range [1..q] of natural;
    end common_node;

globals
    F : tree_node;                     //  root of the query forest QOPT_SO constructs
    Q : set of token_set;              //  the original query set

function combine(node1, node2 : ^tree_node) : ^tree_node;
    new_node : ^tree_node;
begin
    new_node := new tree_node;
    new_node→leading := node1→leading ∩ node2→leading;
    node1→leading -= new_node→leading;
    node2→leading -= new_node→leading;
    new_node→children &= node1;
    new_node→children &= node2;
    return new_node;
end combine;
```

Figure 14. Data Structures For Set-Overlap Query Optimization Algorithm

- A **tree, high_common**, is found which has the most tokens in common with some other **tree, other_common**.*
- These two trees are combined into one and placed in the position formerly occupied by **high_common**.
- The **common** data structures are updated to reflect the combined trees. Step one of this cleanup involves removing the **other_common tree** from reference

* As written in Figure 15, an arbitrary resolution is made between ties. A better heuristic, though, counts the number of other **trees** with which each **tree** has one or more tokens in common. Ties are broken in favor of the **tree** which has *some* token in common with the fewest number of other **trees**. This practice **combines** those **trees** with the fewest combination options first; early combination of **trees** with many options might otherwise interfere with some **tree's** only option.

```
procedure QOPT_SO(F : ^tree_node, Q : set of token_set);
    tree : array [1..q] of ^tree_node
    //  Yes, this really should include a heap index for ordered_common to assist deletions.
    //
    common : array [1..q, 1..q] of natural := 0;
    ordered_common : array [1..q] of d-heap [q] of common_node ascending key in_common;
    max_common : d-heap [q] of common_node ascending key in_common;
    high_common, other_common, i, j : range [1..q] of natural;

begin
    //  Initially, all query sets are uncombined.
    //
    for Q_j ∈ Q do                          //  j will range in 1..q
        tree[j] := new tree_node;
        tree[j]→leading = Q_j;
    endfor;

    //  Generate initial information about number of common tokens.
    //
    for j in [1..q] do
        for i in [1..q] do
            if j ≠ i then                   //  don't combine with self!
                common[j, i] := |Q_j ∩ Q_i|;
            else
                common[j, i] := 0;
            endif;
        endfor;
        ordered_common[j] := makeheap(⟨common[j, i], i⟩);       //  implied i in [1..q]
    endfor;

    max_common := makeheap(⟨findmax(ordered_common[j]), j⟩);    //  implied j in [1..q]

    while findmax(max_common).in_common > 0 do
        high_common := deletemax(max_common).idx;
        other_common := deletemax(ordered_common[high_common]).idx;
        tree[high_common] := combine(tree[high_common], tree[other_common]);
        tree[other_common] := null;
        remove other_common from max_common and all ordered_common;
        recalculate common[high_common, ..] and common[.., high_common];
    endwhile;

    F→children &= all non-null elements of tree;
end QOPT_SO;
```

Figure 15. Set-Overlap Query Optimization Algorithm

throughout the data structures. This requires zeroing-out that tree's **common** row and every other **tree's other_common** column. These **trees' ordered_common** heaps will have to be updated, as well, to delete the heap node for **other_common**. Step two of the cleanup involves rebuilding the **common** information for the new **tree** (including re-inserting a node for it into **max_common**) and propagating these leading-token intersection counts to all the remaining **trees'** rows in **common**. Again, this might involve changing the other **trees' ordered_common** heaps and, perhaps, their nodes in **max_common** as well.

QOPT_SO takes a bit longer than **QOPT_TF**, but not terribly so. Note that, unlike for **QOPT_TF**, it is suggested that **token_sets** are best implemented as bit vectors for this algorithm. The first phase of initialization requires $O(qK_e)$ time to build the original uncombined trees. The second initialization phase uses $O(q^2K_e)$ time to build the $q \times q$ **common** structure. Remember that **makeheap** takes only $O(q)$ time per **tree** to build **ordered_common**, not $O(q\log q)$, since all key values are known at build time. Similarly, the **makeheap** for **max_common** takes only $O(q)$ time.

Since one **tree** is combined into another for each pass of **QOPT_SO**'s main loop, there can be at most q - 1 passes. Within each pass, the two **deletemax** operations take $O(\log q)$ time and the **combine** just $O(K_e)$. The significant time is spent in updating **common**. The first stage of this cleanup requires $O(\log q)$ time to remove the reference to **other_common** from **max_common**, and then q $O(\log q)$ **deleteitem** operations to remove **other_common** from every **tree's ordered_common**. The second stage of the cleanup requires $O(qK_e)$ time to recalculate **common[high_common]**, an $O(\log q)$ **insert** into **max_common**, and up to q $O(\log q)$ operations to update other **tree's ordered_common** heaps if their **common[.., high_common]** values were changed by this recalculation. Similarly, if these changes altered the maximum value of an **ordered_common** heap, another $O(\log q)$ update to **max_common** is required for each such modification, though it is believed that this frequency is small.

Adding things up, it is found that each of the $O(q)$ passes through **QOPT_SO**'s main loop takes $O(q(K_e+\log q)$ time. Initialization takes $O(q^2K_e)$ time, so total time for **QOPT_SO** is $O(q^2(K_e+\log q))$.

4.5.1 Non-Optimal Cases for QOPT_SO

As does **QOPT_TF**, **QOPT_SO** fails to yield optimal query structures in some cases. Since **QOPT_SO**, in general, performs significantly better than QOPT_TF, QOPT_SO's non-optimal cases have generated sufficient interest to warrant a more detailed investigation. Two such cases are presented here, the first showing how **QOPT_SO** can produce a query structure of size approaching 5/4 optimal size, and the second showing a QOPT_SO structure which is 4/3 optimal size.

In the following discussion, let **X**, **Y**, and **Z** indicate sets of tokens. Appropriately, let $x \equiv |\mathbf{X}|$, $y \equiv |\mathbf{Y}|$, and $z \equiv |\mathbf{Z}|$. It is given that $\mathbf{X} \cap \mathbf{Y} = \varnothing$, $\mathbf{Y} \cap \mathbf{Z} = \varnothing$, and $\mathbf{Z} \cap \mathbf{X} = \varnothing$.

Figure 16a presents a set of four token sets for which QOPT_SO yields a non-optimal structure if the conditions $x + z > y$, $x \leq y$, and $z \leq y$ hold.* QOPT_SO's initial **common** (overlap) array is displayed in Figure 16b. This shows that, given the above conditions, it is always possible for **QOPT_SO** to select sets 2) and 3) for the first set combination. This choice results in the query structure illustrated in Figure 16c, which has size $2x+y+2z+4$. The optimal structure, size $x+2y+z+4$, would be as shown in Figure 16d.

The maximum size differential between Figure 16c and Figure 16d occurs when $x = y = z$. In this case, **QOPT_SO**'s structure has size $5y + 4$, while the optimal structure has size $4y + 4$. As y increases in size, the ratio of the sizes approaches 5/4.

One might observe that, at first inspection, the heuristic described on page ? could avoid this non-optimal solution. Though this is sometimes true, test cases have been created which nonetheless yield non-optimal results via this scenario. An even better heuristic than that on page ? is to choose the set combination which keeps the most potential choices available for the *next* set combination. This is a form of one-move look-ahead, the logical extension of which is to look ahead two or more combinations in advance. Such a practice, however, quickly leads to the combinatorial explosion of decisions which QOPT_SO is designed to avoid.

A variation on Figure 16a can produce a QOPT_SO/optimal size ratio of 4/3. This is shown in Figure 17a. Two more token sets are required, both of which are supersets of the token set **Y′**. $\mathbf{Y}' \subset \mathbf{Y}$, and $y' \equiv |\mathbf{Y}'|$. For this case, QOPT_SO yields a non-optimal structure when

---

* The condition $x + z > y$ reflects the difference in size between the optimal structure and that generated by QOPT_SO. The conditions $x \leq y$ and $z \leq y$ guarantee that QOPT_SO can choose to first combine sets 2) and 3), which overlap by y tokens (instead of, for instance, sets 1) and 2) which overlap by x tokens).

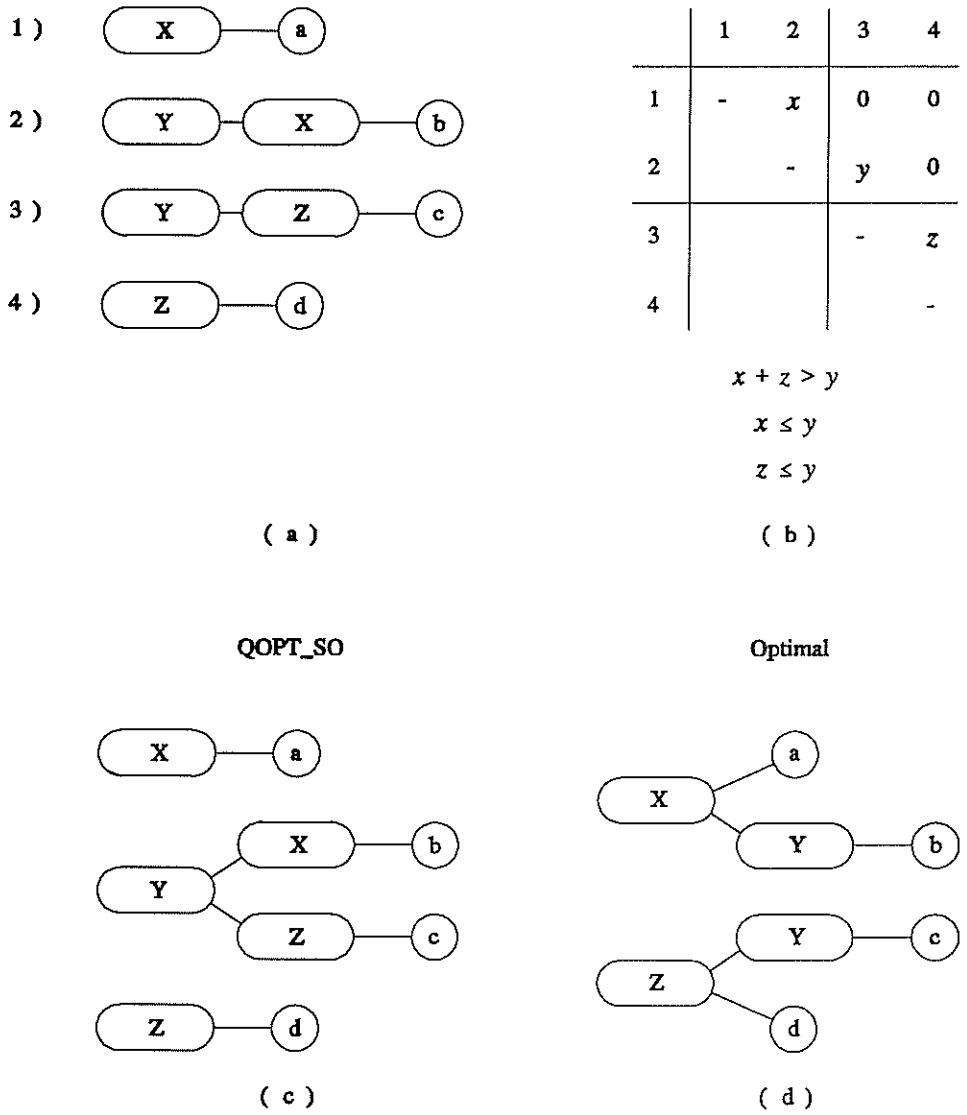
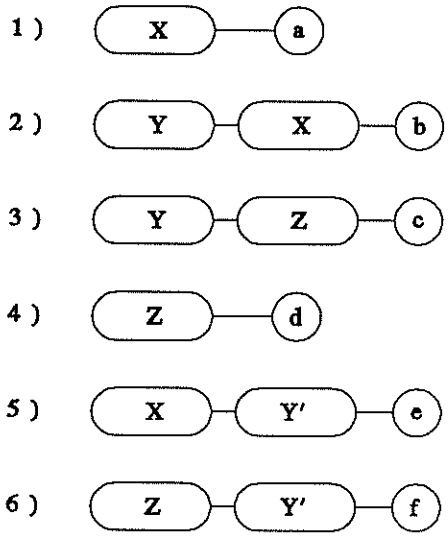


Figure 16. **QOPT_SO** Structure 5/4 Optimal Size

the conditions $x + z + 2y' > y$, $x + y' \leq y$, $z + y' \leq y$, $x \geq y'$, and $z \geq y'$ hold. Once more, Figure 17b displays **QOPT_SO**'s initial **common** (overlap) array.

As with the first example, **QOPT_SO** can always select sets 2) and 3) for the first set combination (given the above conditions). That combination, followed by two more combinations involving sets 5) and 6), can result in the structure shown in Figure 17c. This is of size $2x+y+2z+2y'+6$. The optimal structure is shown in Figure 17d. It is of size $x+2y+z+6$.



(a)

	1	2	3	4	5	6
1	-	x	0	0	x	0
2		-	y	0	$x + y'$	y'
3			-	z	y'	$z + y'$
4				-	0	z
5					-	y'
6						-

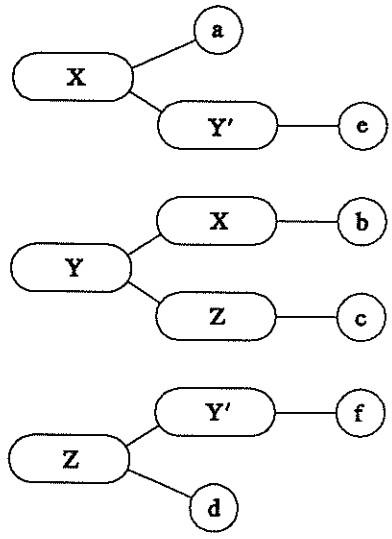
$$x + z + 2y' > y$$

$$x + y' \leq y \quad x \geq y'$$

$$z + y' \leq y \quad z \geq y'$$

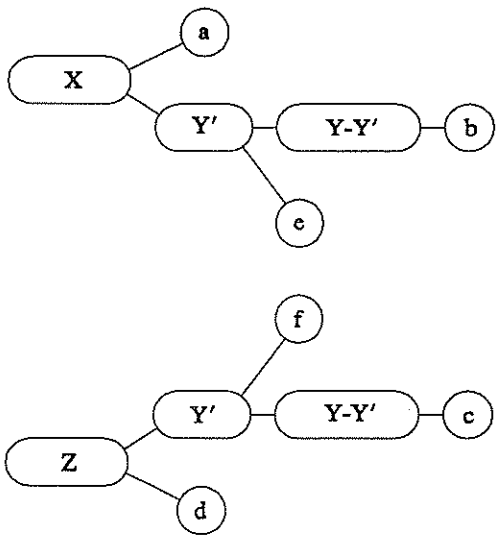
(b)

QOPT_SO



(c)

Optimal



(d)

Figure 17. QOPT_SO Structure 4/3 Optimal Size

For this example, the maximum size differential occurs when $x = z = y'$ and $2x = y$. **QOPT_SO**'s structure size is thus $4y + 6$, while the optimal size is $3y + 6$. A 4/3 ratio is therefore achieved for large values of y.

5. Appendices

Appendix 5.1 Pseudocode Representation

The representation of algorithms in this report is done using pseudocode which resembles a mixture of Pascal, Ada, and C++. All the standard control structures are available, defined types may be expressed, and a variety of operators may be used.

Below are listed the details of this representation. In pseudocode tradition, however, the more obvious operations in our algorithms are generally expressed with a certain amount of English instead of detailed statements (such as "**for** *every* child *of*..." instead of "child:= foo→child; **while** child ≠ **null do**..."). When such use of English is made instead of formal code, this will be clarified by italicizing any English in our algorithms (e.g. "**for** *every* child *of*..." in the above example).

In the following discussion, bold brackets ([ ]) indicate 0 or 1 occurrence of the enclosed item, and bold braces ({ }) indicate 0 or more occurrences. Comments in this pseudocode are as in C++: '//' indicates that the rest of the line is a comment.

5.1.1 Control Structures

Flow of control is Ada-like. Semicolons are statement terminators, not separators, and loop entry statements are paired with matching loop exit statements. Procedures and functions may be defined and nested, following the usual scope rules. Syntax is:

Sequence

statement;
{*statement*;}

Conditional

if *condition* then
    sequence;
else
    sequence;
endif;

Alternative

case *expression* of
    value_list:
        (*sequence*;);
    ...
    others:
        (*sequence*;);
endcase;

Iteration

for *variable* in *range* do
    sequence;
endfor;

Repetition, Test At Entry

while *condition* do
    sequence;
endwhile;

Repetition, Test At Exit

repeat
    sequence;
until *condition*;

Procedure

procedure *proc_name(formal_parameters)*;
    declarations;
begin
    sequence;
    return;
end *proc_name*;

Function

function *func_name(formal_parameters)* :
        result_type;
    declarations;
begin
    sequence;
    return *value*;
end *func_name*;

— where *formal_parameters* is a list, the elements of which are separated by semicolons and have the form *variable_name*{, *variable_name*} : *type*

5.1.2 Operators

assignment:	:=	// *var* := *value*
arithmetic:	+, -, *, /, %	// add, subtract, multiply, divide, modulus
arithmetic assign:	+=, -=, *=, /=, %=	// *var op*= *value* ≡ *var* := *var op value*
comparison:	=, ≠, <, ≤, >, ≥	
logical:	**and, or, xor, not, andif, orelse**	// two "short circuit" operators

5.1.3 Simple and Structured Types

Basic types include the standard **integer**, **real**, **Boolean**, and **character**. Derived types include enumerations and subranges of any ordinal type. Structure is expressed by use of **array**, **record**, and pointer types which may be arbitrarily nested. As with C++, indexing of an array and of a dereferenced pointer to an array is not distinguished; if **a_p** is a pointer to an array, **a_p^[i]** and **a_p[i]** are equivalent. Records can have Pascal-like variant fields. Syntax is:

Subrange

subrange_type =
    range [*first..last*]
        of *base_type*;

Enumeration

enumeration_type =
    (*value*{, *value*});

Array

array_type =
    array [*range*{, *range*}]
    of *base_type*;

Record

record_type = **record**
  field_name : *type*;
  ...
end *record_type*;

Variant Record

record_type = **record**
 {[*field_name* : *type*;]
 [**case** [*tag* :] *type* **of**
  value_list:
    (*field_name* : *type*;
    ...);
  ...
  others:
    (*field_name* : *type*;
    ...);
 endcase;]}
end *record_type*;

Pointer

pointer_type = ^*base_type*;

Pointer Dereference

pointer_variable^
Also,
pointer_variable→
is equivalent to
pointer_variable^.

5.1.4 High-level Structured Types

Collections of elements of any other type may be built as sets, lists, and sorted sets (search trees). The syntax for declaring such collections and the operations allowed with them are as follows:

Sets

Sets are defined as unordered collections of objects with no duplicates. Basic set operations of union, intersection, symmetric difference, proper subset and superset, construction, and element containment may be expressed $\cup$, $\cap$, -, $\subset$, $\supset$, { *element*{, *element*} } and $\in$, respectively.

declaration: *type_name* = **set of** *base_type*;

operators: $\cup$, $\cap$, -, =, $\subset$, $\subseteq$, $\supset$, $\supseteq$, $\in$, and the assignment operators $\cup$=, $\cap$=, and -=

constants: $\varnothing$ — the empty set

Lists

Lists are defined as collections of objects ordered by their sequence of appearance within the list; duplicates are allowed. Operations include concatenation, construction, element reference, and sublist reference expressed by &, [*element*{, *element*}], *list*(*element_number*), and *list*[*element_range*], respectively.

declaration: *type_name* = **list of** *base_type*;

operators: &, (*element_number*), [*element_range*], and the assignment operator &=

constants: [ ] — the empty list

Sorted Sets

Sorted sets are defined as collections of objects ordered by means of a "key" value, with no duplicate key values allowed between two elements. This key may either be the element itself, if the sorted set is of a simple type, or is the value of one field of an element, if the sorted set is of a record type. Operations include insertion and removal of elements and search according to a key.

Insertion of an element into a sorted set either adds an entirely new element or replaces an existing element of the same key. This operation is expressed as *set* + *element*. Removal of an element from a sorted set, expressed as *set* - *element*, fails if the element is not part of the sorted set. Reference to an element by key has many search criteria and returns a pointer to that element (or **null** if no such element is found). The search may be for the element with key equal to the search key ('=' search); for the element with the greatest key less than the search key ('<' search); for the element either with the search key or, if not found, with the greatest key less than the search key ('≤' search); and so on for '>' and '≥' search. Equal-to search is common enough to be expressed as *sorted_set*[*key*]; searches with other criteria are expressed as *sorted_set*(*criterion, key*).

Algorithms which perform a search for a particular element in a sorted set and then scan successive elements of that set starting at that search point are quite common. To this end, operations **next** and **prev** are provided to scan in increasing and decreasing order, respectively. If no further elements exist in that "direction" in the set, these operations return **null**. So that a scan may begin at either the start or end of a sorted set, the operations **first** and **last** are provided. These operations return the appropriate element, or **null** if the set is empty.

declaration: *type_name* = **srt_set of** *base_type* [**key** *field_name*];

operators: +, -, ∈, [*key*] — equivalent to '=' *criterion* below,
(*criterion, key*), where *criterion* is one of =, <, >, ≤, or ≥,
next(), prev(), first(), last(), and the assignment operators += and -=

constants: ∅ — the empty sorted set

Appendix 5.2 Orthogonal Range Query Search

An orthogonal range query[16][3][11] **RQ** over D, a set of k-tuples, is defined as:

$$\mathbf{RQ}(D,\ a_1..b_1,\ a_2..b_2,\ \ldots\ ,\ a_k..b_k) = \left\{ t \;\middle|\; \begin{array}{c} t \in D \\ \forall i\ [1 \le i \le k \supset a_i \le t.c_i \le b_i] \end{array} \right\}$$

RQ is a multi-dimensional (or multi-keyed) search across D in which a range of values for each of the dimensions (keys) may be specified. Let $d \equiv |D|$ and $m \equiv$ *the maximum number of distinct values for any tuple components*. The search time for the best orthogonal range query algorithms is $(\log d)^{k-1}$, insert/delete time is $(\log d)^k$, and space requirements are $d(\log m)^{k-1}$. Because of both the time and the space requirements of these algorithms, they are useful only for small dimensionalities (realistically, dimension 2, maybe 3, due to space considerations).

Though variations exist, the basic strategy employed for building range query data structures is to have a separate set of component search trees for each of the $\log m$ levels of the search tree for the previous component. Intuition shows that the space analysis is correct; this uses a tremendous amount of space. Regardless of this, for small dimensions the space penalty might be worth while if speed is essential for the search of a large database.

Figure 18 shows a diagram of a 2-dimensional ($k = 2$) range query data structure similar to that shown in [16]. A search tree for the first component, c_1, makes up the base tree **B**. For our example, it is easiest to assume that the tuples are stored at the leaves of **B**. Given a range of values $a_1..b_1$ for c_1, there will exist certain interior nodes **x** of **B** for which the c_1 values for all **x**'s descendant leaves fall in the range, but not all of **x**'s parent's descendant leaves' c_1 values fall in the range. These nodes are called critical for the range $a_1..b_1$. In our example, nodes **b** and **f** are critical for the range 1..3. There are at most $\log d$ of these critical nodes for any range, and they may all be found in $\log d$ time.

When we have the critical nodes for a given c_1 range, we know the subtrees containing all the tuples which have c_1 values in that range. We now need to know which of the leaves of those subtrees also have a c_2 value in the range $a_2..b_2$. Each critical node is the root of a subtree of **B**; the algorithm makes use of this by maintaining an auxiliary search tree (sorted by c_2) for each possible subtree of **B**. For the root of **B**, this is simply the entire database sorted on c_2. For interior nodes **b** and **c**, the auxiliary trees each contain half the tuples in the database, again sorted

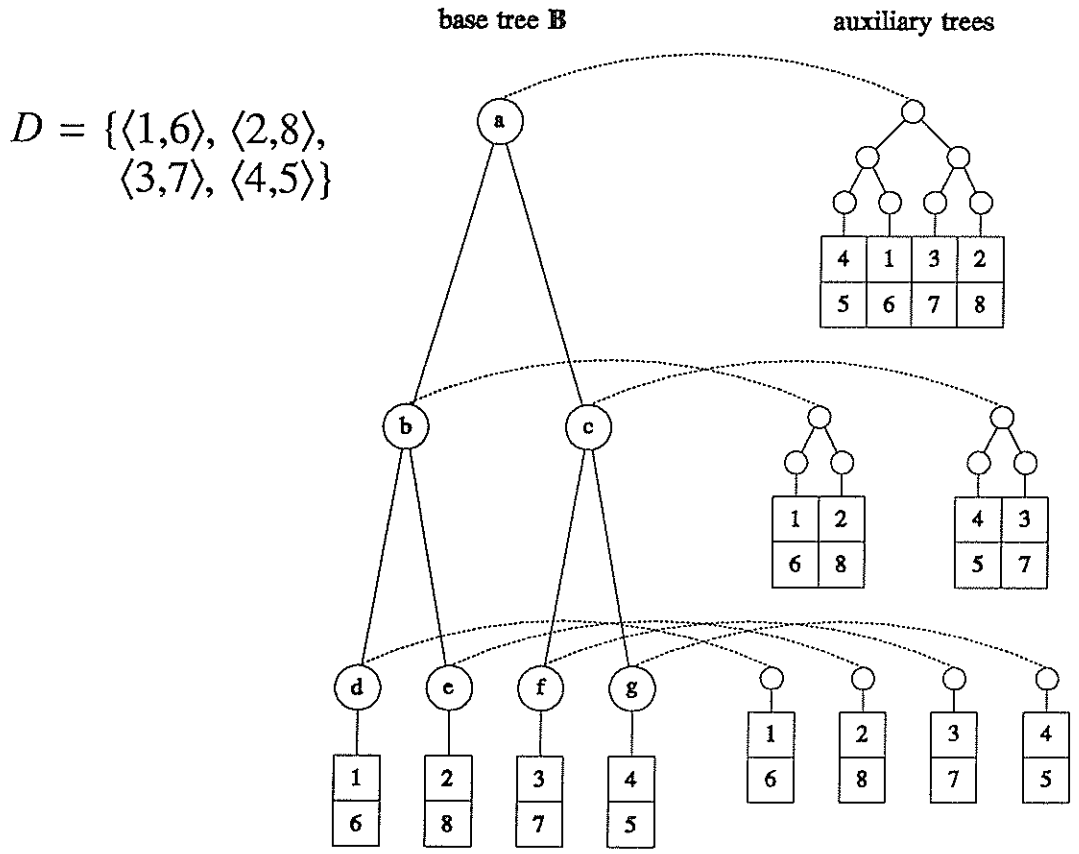


Figure 18. Orthogonal Range Query Data Structures

on c_2. This continues for each level in **B**: for each level in **B**, there is a corresponding level of auxiliary trees which sorts the entire database by c_2.

Given the critical nodes for the c_1 range, we find those nodes' corresponding auxiliary trees and search each of them for those children with c_2 values in the desired range. Since each auxiliary tree contained only those nodes with $c_1 \in [a_1..b_1]$, the result of the c_2 search is exactly what we desire. For example, let us continue our example and search for those tuples with $c_1 \in [1..3]$ and $c_2 \in [4..6]$. We know that the critical nodes for the c_1 range are **b** and **f**. Therefore, we search their auxiliary trees for all nodes with c_2 values in the range 4..6. We find one such tuple, ⟨1,6⟩, in **b**'s auxiliary tree, and no matching tuples in **f**'s auxiliary tree. This single tuple is thus the answer to our query. Higher dimension queries apply this procedure recursively, treating c_1's auxiliary nodes as new base trees with their own auxiliary nodes.

6. Bibliography

1. Bentley, Jon Louis. "Multidimensional Binary Search Trees Used For Associative Searching." Communications of the ACM Vol. 18, no. 9 (September 1975): 509-517.

2. Bentley, Jon Louis, and Friedman, Jerome H. A Survey of Algorithms and Data Structures for Range Searching. Stanford, CA: Computation Research Group, Stanford Linear Accelerator Center. Publication PUB-2189, August 1978.

3. Bentley, Jon Louis, and Maurer, H. A. "Efficient Worst-Case Data Structures for Range Searching." Acta Informatica Vol. 13 (1980): 155-168.

4. Finkel, R. A., and Bentley, Jon Louis. "Quad-trees: A Data Structure for Retrieval on Composite Keys." Acta Informatica Vol. 4 (1974): 1-9.

5. Garey, Michael R., and Johnson, David S. COMPUTERS AND INTRACTABILITY: A Guide to the Theory of NP-Completeness. New York, NY: W. H. Freeman and Company, 1983.

6. Griswold, Victor Jon. Determining Interior Vertices of Graph Intervals. St. Louis, MO: Dept. of Computer Science, Washington University. Technical Report WUCS-90-9, February 1990.

7. Griswold, Victor Jon. Determining Interior Vertices of Graph Intervals (revision of WUCS-90-9). St. Louis, MO: Dept. of Computer Science, Washington University. Technical Report WUCS-90-40, November 1990.

8. Horowitz, Ellis, and Sahni, Sartaj. Fundamentals of Data Structures. Rockville, MD: Computer Science Press, Inc., 1982.

9. Karp, Richard M. "Reducibility Among Combinatorial Problems." Complexity of Computer Computations. Miller, Raymond E., and Thatcher, James W. eds. New York, NY: Plenum Press, 1972: 85-103.

10. Kung, H. T.; Luccio, F.; and Preparata, F. P. "On Finding the Maxima of a Set of Vectors." Journal of the Association for Computing Machinery Vol. 22, no. 4 (October 1975): 469-476.

11. Lee, D. T., and Wong, C. K. "Worst-Case Analysis for Region and Partial Region Searches in Multidimensional Binary Search Trees and Balanced Quad Trees." Acta Informatica Vol. 9 (1977): 23-29.

12. Lueker, George S. "A Data Structure for Orthogonal Range Queries." Proceedings of the 19th Annual Symposium on Foundations of Computer Science (1978): 28-34.

13. Lueker, George S., and Willard, Dan E. "A Data Structure for Dynamic Range Queries." Information Processing Letters Vol. 15, no. 5 (10 December 1982): 209-213.

14. Rivest, Ronald L. "Partial-Match Retrieval Algorithms." SIAM Journal on Computing Vol. 5, no. 1 (March 1976): 19-50.

15. Standish, Thomas A. Data Structure Techniques. Reading, MA: Addison-Wesley Publishing Company, Inc., 1980.

16. Willard, Dan E. "New Data Structures For Orthogonal Range Queries." SIAM Journal on Computing Vol. 14, no. 1 (February 1985): 232-253.