Applications of Range Query Theory to Relational Data Base Join and Selection Operations

DAN E. WILLARD*

Computer Science Department, State University of New York, Albany, New York 12222

Received November 23, 1994

This article applies range query theory to develop join algorithms that run in \(O(l \log^d l + U)\) time, where \(l\) and \(U\) are the sizes of the input and output and \(d\) is usually a small constant. One advantage of these algorithms is that they do not require the storage of an index, and they also use a working memory space guaranteed to be proportional to the size of the input. If the memory space is expanded to \(O(N \log^d N)\), our formalism also leads to the development of very fast indices supporting \(O(\log^d N)\) selection operations. © 1996 Academic Press, Inc.

1. INTRODUCTION

Throughout this paper, \(X\) and \(Y\) denote two sets of tuples, and \(x\) and \(y\) denote two tuple variables ranging over these respective sets. The lower case bar symbols, \(x\) and \(y\), will denote particular tuples in \(X\) and \(Y\), and \(e(x, y)\) denotes a predicate defined over the cross product space \(X \times Y\). This paper will present an efficient algorithm for searching for the ordered pairs \((x, y)\) which satisfy \(e(x, y)\). We will also study three other closely related problems.

Let \(A_1(x), A_2(x), A_3(x)\ldots\) denote attributes of the tuple \(x\). Define an equality atom to be a predicate of the form \(A_i(x) = A_i(y)\), and an order atom to be a predicate of the form \(A_i(x) > A_i(y)\). Our final theorem will be stronger if it also includes two further types of atoms. Define a subsection \(L\) to be a list of tuples. Also, define a tabular section \(T\) to be a list of ordered pairs. Then a list atom is defined to be a predicate of the form \(x \in L\) or \(y \in L\), and a tabular atom is defined as a predicate of the form \((x, y) \in T\).

The predicates \(e(x, y)\) will be called enactments. Their most general version, called the E-8 class, will consist of equality, order, list, and tabular atoms combined in an arbitrary manner by AND, OR, and NOT connectives. Two examples are given below.

\[
e_d(x, y) = \left\{ \begin{array}{l} \{ (x, y) \in T_1 \lor (x, y) \in T_2 \} \land A_1(x) > B_1(y) \\
\land A_2(x) \land A_2(y) \land x \in L_1 \land y \in L_2 \} \right\. \tag{1.1}
\]

2. MAIN RESULTS

We will investigate four types of retrieval operations. The reporting join is defined as an operation that takes the enactment \(e(x, y)\) and the two sets \(X\) and \(Y\) as argument and constructs the set of ordered pairs \((x, y)\) from the cross product space \(X \times Y\) satisfying \(e(x, y)\). Sections 6 through 9 prove these operations are executable in \(O(N + M)\) work space and never need more than \(O(M + U + N \log^d N)\) WH-time (and they often use much less time).

Define an E-7 enactment to be an E-8 predicate that contains no tabular atoms. Our second topic, the on-line reporting problem, will view the set \(X\) as nonexistent. It will assume one is initially given only a dynamically changing set \(Y\) and an E-7 enactment \(e\). Let \(Y(\bar{x})\) denote the set of \(y \in Y\) which satisfy \(e(\bar{x}, y)\). Let us view \(\bar{x}\) as a query seeking to construct this set. Then Section 7 constructs a dynamic data structure \(D(Y)\) such that...
(i) queries run in $O(\log^{d+1} N + U)$ WH-time, where $U$ is the size of the output.

(ii) Insertions and deletions in $D_e(Y)$ also run in $O(\log^{d+1} N)$ AH-time.

Our third search problem, called on-line aggregation, assumes there exists a function $f$ that maps each element $j \in Y$ into an abelian semi-group, and we wish to calculate the quantity $\Phi_j(x) = \sum_{j \in x} f(j)$ for an E-7 enactment $e$. We will display a dynamic data structure, henceforth denoted as $D_e(Y)$ that guarantees aggregation queries and updates run in $O(1)$ WH-time when $d(e) = 0$, and $O(\log^{d+1} N)$ worst-case time when $d(e) \geq 1$.

Finally, define an aggregate join to be a query whose arguments are an E-8 enactment $e(x, y)$ and two sets $X$ and $Y$ and which seeks to construct an array $\Phi_j(\cdot)$, which contains the subtotals $\Phi_j(x_1), \Phi_j(x_2), \Phi_j(x_3), \ldots$ for each $x_i \in X$. Let $d^*(e)$ equal $d(e)$ when $d(e) \leq 1$ and equal $d(e) - 1$ when $d(e) \leq 2$. The procedure from the previous paragraph can perform aggregate joins if one makes $N$ subroutine calls to it, but a more efficient solution will extend the ECDF method [Be80] to calculate the array $\Phi_j(x)$ in $O(N + M)$ space and $O(N + M)$ WH-time when $d(e) = 0$, and $O(N + M)$ space and $O(M + N \log^{d+1} N)$ worst-case time when $d(e) \geq 1$. All four of the preceding results have practical applications under some circumstances. However, the two join algorithms are especially attractive because they never use more than linear space and never require the data to be preprocessed at the onset of the calculation.

There will be insufficient space in this paper, but if one applies fractional cascading [CG86, MN90], the BB$(x)$ range tree method [WL85], and a stronger worst-case version of Proposition 6.3 (which appeared in [Wi78]) then all the preceding reporting algorithms will support worst-case time bounds when $d(e) \geq 1$.

3. LITERATURE SURVEY AND MOTIVATION

An on-line query, where $x$ corresponds to a $k$-dimensional box and $e$ corresponds to a request to retrieve $y$-elements from the interior of the box is called an orthogonal range query. A quite extensive literature has studied this problem [Be75, Be80, BM80, BS80, Ch86, Ch88, Ch90a, CH90b, CG86, Ed81, Ed87, EO82, EO85, Fr81a, Fr81b, LP84, LW77, LW80, LW82, Me84, MN90, OL81, Ov88, PS85, OS90, Sm89, Ya85, Wi78, Wi85, Wi86, Wi87, Wi92, Ya85], and it is closely related to the aspects of our algorithm that process order predicates. Our algorithm extends the prior literature primarily by considering the added complications from equality, tabular, and list atoms, in a relational calculus context. Our two join algorithms will use techniques related to [Be80, EO85] to process order atoms. The dynamic on-line query algorithms will similarly employ [Be80, WL85]’s methods.

A database query that scans an input of cardinality $I$ and produces an output of cardinality $U$ in $O(I \log^{d+1} U)$ WH-time and $O(I + U)$ space will be said to have quasi-linear complexity with exponent $d$. (Note $I \leq 2N + M$ for E-8 enactment joins.) A pragmatic problem in database applications is to devise a broad class of algorithms that can automatically process complicated database queries in quasi-linear efficiency with a small exponent $d$ (ideally with $d$ equal 0 or 1). The chief reason for studying E-8 enactments is that their aggregation and reporting join procedures will enable many relational calculus queries to become quasi-linear efficient, as illustrated in the following examples.

Consider first, the two very simple “binary” relational queries

$$\{\text{FIND}(x \in X), \exists y \in Y: e(x, y)\} \quad (3.1)$$

$$\{\text{FIND}(x \in X), \forall y \in Y: (x, y)\}. \quad (3.2)$$

The aggregate join concept provides an efficient mechanism for answering both these queries. For example, query (3.1) can be resolved by first calculating an array $\Phi_j(x)$, which indicates how many $y \in Y$ satisfy $e(x, y)$, and then outputting those elements $x \in X$ satisfying $\Phi_j(x) \geq 1$. An analogous algorithm will process Eq. (3.2) by outputting those $x \in X$ satisfying $\Phi_j(x) = \text{Cardinality}(Y)$. Both universal and existential quantifiers can thus be processed in quasi-linear time under the aggregate join formalism.

Our next example will explain why the enactment formalism was designed to include listmembership atoms. Let $e_1$ and $e_2$ denote two enactment predicates in the query:

$$\{\text{FIND}((x, y) \in X \times Y): \exists z \in Z: e_1(x, y) \land e_2(x, z)\}. \quad (3.3)$$

One provably quasi-linear algorithm for processing (3.3) consists of the following two steps:

(1) Let $L$ denote the list of those elements $x \in X$ satisfying

$$L = \{\text{FIND}(x \in X) \exists z \in Z: e_2(x, z)\}. \quad (3.4)$$

Construct this set $L$ by using the procedure from the previous paragraph.

(2) Note that the ordered pairs satisfying (3.3) must be the same as those satisfying

$$\{\text{FIND}((x, y) \in X \times Y): x \in L \land e_1(x, y)\}. \quad (3.5)$$
The advantage of rewriting Eq. (3.3) as (3.5) is that its term \( \{ x \in L \land e_i(x, y) \} \) is another E-8 enactment; we can therefore find the ordered pairs satisfying (3.3) by letting a second enactment join algorithm process this derivative expression.

The point of this example is that we can process a 3-variable relational calculus query, similar to (3.3), by performing two “binary” E-8 operations, so that the final algorithm will run in quasi-linear time. The function of the list atom \( x \in L \) in this example has been to store the results from the intermediate query (3.4), so that it can be used by the second operation (3.5).

Our third example is somewhat more complicated. Consider the query

\[
\{ \text{FIND}(x, y, z) \in X \times Y \times Z : e_1(x, y) \land e_2(x, z) \}.
\]  

(3.6)

One correct but very inefficient 3-step algorithm for solving (3.6) is

1. First use an enactment join to find those \((x, y)\) satisfying \(e_1(x, y)\).
2. Next use a second enactment join to find those \((x, z)\) satisfying \(e_2(x, z)\).
3. Let \(V\) and \(W\) denote the two sets constructed by steps 1 and 2. Then the answer to query (3.6) is the “natural join” of these two sets, i.e. it is the set of ordered triples \((x, y, z)\) satisfying \((x, y) \in V\) and \((x, z) \in W\).

The interesting facet is that the preceding procedure is correct but not efficient enough to meet the quasi-linear cost criteria. Consider an example where \(|X| = |Y| = |Z| = N\) and \(|V| = |W| = N^2/2\), but where the output from query (3.6) is empty. Then in this case \(I = 3n, U = 0\), and the preceding algorithm is certainly NOT quasi-linear efficient (because its first two steps will require \(O(N^2)\) time to construct intermediate sets of size \(N^2/2\)).

The interesting aspect is that we can process query (3.6) in quasi-linear WH-time if the E-8 enactments are employed more judiciously. Consider the following alternate procedure:

1. First apply the quasi-linear algorithm for processing query (3.3). Let \(V^*\) denote the set of ordered pairs \((x, y)\) satisfying (3.3).
2. Next use an analogous procedure to construct a set \(W^*\) of ordered pairs \((x, z)\) satisfying:

\[
\{ \text{FIND}(x, y, z) \in X \times Y \times Z : e_1(x, y) \land e_2(x, z) \}.
\]  

(3.7)

3. Finally produce the answer to query (3.6) by taking the “natural join” of \(V^*\) and \(W^*\).

It is easy to prove that unlike \(V\) and \(W\), \(V^*\) and \(W^*\) always satisfy the inequalities \(|V^*| \leq U\) and \(|W^*| \leq U\). These inequalities imply our second algorithm, unlike the first procedure, must always runs in quasi-linear WH-time.

The preceding example is interesting because it can be generalized substantially. Let the capital letter symbols \(R_1, R_2, \ldots, R_k\) denote \(k\) sets of tuples (called “relations” in database terminology). Let the symbol \(Q_i(r_i \in R_i)\) denote either an existential or universal quantifier for a tuple variable \(r_i\) spanning a relation \(R_i\), and let \(e(r_1, r_2, \ldots, r_k)\) denote a predicate consisting of several equality order, tabular, and list atoms concatenated in arbitrary manner by AND, OR, and NOT connectives. In this notation, a general relational calculus query has the canonical form:

\[
\{ \text{FIND}(r_1, r_2, \ldots, r_p) \in R_1 \times R_2 \times \cdots \times R_p : Q_{p+1}(r_{p+1} \in R_{p+1}) \}
\]  

(3.8)

Say a variable \(r_j\) precedes a variable \(r_i\) in the query \(q\) if the quantifier or FIND-clause defining \(r_i\) lies to the left of \(r_j\)’s definition in \(q\). Define this query’s relational graph \(G(q)\) to be a graph that has a directed edge from \(r_j\) to \(r_i\) if these two variables are the binary constituents of some equality, order, or tabular atom and \(r_i\) precedes \(r_j\). A relational calculus query \(q\) will be said to satisfy the RCS condition iff its graph is a tree or a forest with all the paths leading to the roots. [Wi90] displays an algorithm, based essentially on generalizing the preceding examples, which guarantees that every RCS query has a decomposition into efficient E-8 enactment operations. The central part of its proof concerns guaranteeing that the sets constructed in the intermediate stages of the computation have cardinality smaller than the final output set (analogous to the size constraints on \(V^*\) and \(W^*\) in the example of query (3.6) ). The reason for our interest in E-8 enactment joins is that the combination of their quasi-linear performance, proven in the next five chapters of this paper, and the decomposition algorithms of [Wi90] imply all RCS queries run in quasi-linear time. Indeed, many relational calculus queries, which do not fall into the RCS category, also have quasi-linear times, by employing more elaborate methods to decompose them into E-8 components.

Finally, we present an example that explains why tabular atoms were included in the RCS and E-8 formalisms. Let \(T\) denote a subset of the cross product set \(R_1 \times R_2\). For each \(r_1 \in R_1\) and \(r_2 \in R_2\), let \(A^*(r_1)\) and \(A^*(r_2)\) denote an attribute-field that contains an unique value for each tuple \(r_i\). (Such attributes are called primary keys in database terminology [U189] ). Let \(R_3\) be a third relation such that for each \((r_1, r_2) \in T\) there exists a corresponding \(r_3 \in R_3\), with \(A_3(r_3) = A^*(r_1)\) and \(A_3(r_3) = A^*(r_2)\). The introduction of such a third relation \(R_3\) makes tabular atoms semantically
unnecessary, since the “atom” \((r_1, r_2) \in T\) is equivalent to the phrase
\[
\exists r_3 \in R_3: A_1(r_3) = A^*(r_1) \land A_2(r_3) = A^*(r_2).
\] (3.9)

Thus, the relational calculus queries in \(q_1\) and \(q_2\) below are semantically equivalent, and it is reasonable for the reader to inquire why tabular atoms should be introduced, since the query \(q_1\), can specify the same set of tuples as \(q_2\) without the burden of the added notation:

\[
q_1 = \{ \text{FIND}(r_1, r_2) \in R_1 \times R_2 : \exists r_3 \in R_3 : \]
\[
A_1(r_3) = A^*(r_1) \land A_2(r_3) = A^*(r_2).
\] (3.10)

\[
q_2 = \{ \text{FIND}(r_1, r_2) \in R_1 \times R_2 : \]
\[
(r_1, r_2) \in T \land A_4(r_1) > A_4(r_2).
\] (3.11)

The key distinction between these two queries is that the relational graph of \(q_1\) is a tree, but the graph of \(q_2\) is not.\(^1\) Formally, this means that the query \(q_2\) satisfies [Wi90]'s RCS condition, but the logically equivalent query \(q_1\) does not.

The combination of this paper and [Wi90] would not imply that the graph \(q_1\) is logically equivalent to a second query, \(q_2\), which can be processed in quasi-linear time, if we had not included the notion of tabular atoms in our formalism. This fact is significant because many pragmatic database applications contain either the atom \((r_1, r_2) \in T\) or the equivalent phrase (3.9) embedded in a query (essentially because these primitives model the many-to-one, one-to-many, and other sparse representations of the cross-product set in the relational model [U189]). The tabular atom concept was thus incorporated into the E-8 and the RCS formalisms to facilitate the efficient quasi-linear processing of such expressions.

4. DECOMPOSITION METHODS

It will often be convenient to decompose an initial enactment expression \(e(x, y)\) into a sequence of enactment expressions \(e_1(x, y), e_2(x, y), \ldots, e_L(x, y)\). There will be three methods for decomposing an enactment into a sequence \(e_1, e_2, \ldots, e_L\), defined:

(i) The sequence \(e_1, e_2, \ldots, e_L\) will be called a
\[
\text{disjunctive decomposition}
\]
\(e\) iff the disjunction identity \(\{ e(x, y) = e_1(x, y) \lor e_2(x, y) \lor \cdots \lor e_L(x, y) \}\) holds.

(ii) The sequence \(e_1, e_2, \ldots, e_L\) will be called a
\[
\text{non-redundant decomposition}
\]
\(e\) iff condition (i) holds and no ordered pair \((\bar{x}, \bar{y})\) can simultaneously satisfy two of its predicates.

\(1\) The graph of \(q_1\) is not a tree because it has arcs from \(r_2\) to \(r_1\), \(r_2\) to \(r_1\), and \(r_2\) to \(r_1\). The graph of \(q_2\) is a tree because its sole arc is from \(r_2\) to \(r_1\).

(iii) Let the integers 1 and 0 designate the Boolean constants of TRUE and FALSE. Define the sequence \(e_1, e_2, \ldots, e_L\) to be an
\[
\text{arithmetic decomposition}
\]
e iff there exists integer constants \(k_1, k_2, \ldots, k_L\), such that every ordered pair \((\bar{x}, \bar{y})\) must satisfy \(e(\bar{x}, \bar{y}) = \sum_{i=1}^{L} k_i e_i(\bar{x}, \bar{y})\).

We will study eight classes of enactment expressions in this paper, called the E-1 thru E-8 categories. For each \(i \leq 7\), the E-\(i\) class will be a proper subset of the E-\((i+1)\) category, and our algorithm for processing the E-\((i+1)\) category will consist of doing some fixed amount of initial processing and then modularly decomposing the E-\((i+1)\) search problem into a series of subroutine calls to E-i enactment search algorithms. We will often use a disjunctive decomposition \(e_1, e_2, \ldots, e_l\) to calculate \(Y_e(x)\) by setting the latter set equal to \(Y_{e_1}(x) \cup Y_{e_2}(x) \cup \cdots \cup Y_{e_l}(x)\). Similarly, (iii)’s arithmetic decomposition condition implies the summand \(\Phi_e(x) = \sum_{i=1}^L k_i \Phi_i(x)\) will resolve an aggregation query. The nonredundant decomposition satisfies both these conditions, where \(k_1 = k_2 = \cdots = k_L = 1\) in the preceding summand.

If \(e\) is decomposed into a sequence \(e_1, e_2, \ldots, e_L\) and each \(e_i\) uses time of \(O[t(N, M, U)]\) and space of \(O[s(N, M, U)]\), we will infer \(e\) also has a time of \(O[t(N, M, U)]\) and space of \(O[s(N, M, U)]\), where \(e_i\)’s coefficients clearly depend on \(L\). It is permissible for the \(O\)-notation to treat \(L\) as a constant because the value of \(L\) is a function solely of \(e\) (unlike \(N, M, U\)). Several of our algorithms will have unnecessarily large coefficients, for the sake of a brief presentation.

5. AGGREGATION ALGORITHMS FOR
E-3 ENACTMENTS

Our first three enactment classes, the E-1, E-2, and E-3 categories, as well as the related NEG-1 class, are:

E-1. A predicate which is either the Boolean constant “TRUE” or a conjunction of several equality atoms, i.e., as in \(\{ A_1(x) = B_1(y) \land \cdots \land A_k(x) = B_k(y) \}\).

NEG-1. A predicate which is either the Boolean constant “TRUE” or a conjunction of several inequality terms, i.e., as in \(\{ A_1(x) \neq B_1(y) \land \cdots \land A_k(x) \neq B_k(y) \}\).

E-2. A conjunction of an E-1 with a NEG-1 predicate

E-3. An arbitrary predicate whose atoms are equalities concatenated in an arbitrary manner by AND, OR, and NOT connectives.

Define the negative degree of a NEG-1 or E-2 query to be the number of inequality terms in this enactment expression.

Lemma 5.1. Every E-2 and E-3 enactment query can be arithmetically decomposed into a sequence of E-1 enactments.

Proof. We will separately verify Lemma 5.1 for E-2 and E-3 queries.
Verification when \( e \) is an \( E-2 \) query by induction on \( e \)'s negative degree. If \( e \)'s negative degree = 0 then this lemma holds automatically because \( e \) is then an \( E-1 \) enactment. On the other hand, if \( e \) is an \( E-2 \) query of negative degree \( = J > 0 \) then it will appear in a canonical form similar to (5.1), where \( e^*(x,y) \) in that equation is an \( E-1 \) enactment

\[
e(x,y) = \{ e^*(x,y) \land A_2(x) \neq B_1(y) \land \ldots \land A_J(x) \neq B_J(y) \}.
\]

(5.1)

Consider the queries \( e_A \) and \( e_B \)

\[
e_A(x,y) = \{ e^*(x,y) \land A_1(x) \neq B_1(y) \land \ldots \land A_J(x) \neq B_J(y) \} \tag{5.2}
\]

\[
e_B(x,y) = \{ e^*(x,y) \land A_1(x) = B_1(y) \} \tag{5.3}
\]

Since \( e_A \) and \( e_B \) both have negative degree \( = J - 1 \), the principle of induction implies arithmetic decompositions of these enactments into \( E-1 \) operations. Since (5.1) thru (5.3) imply \( e(\tilde{x}, \tilde{y}) = e_A(\tilde{x}, \tilde{y}) - e_B(\tilde{x}, \tilde{y}) \), the union of these two decomposition sequences must provide an arithmetic decomposition of \( e \) into \( E-1 \) enactments (because if \( e_A \) and \( e_B \) can be written in the forms \( e_A(\tilde{x}, \tilde{y}) = \sum_{k=1}^{L-1} k \cdot e_{SA}(\tilde{x}, \tilde{y}) \) and \( e_B(\tilde{x}, \tilde{y}) = \sum_{k=1}^{L-1} (k - c_i) e_{SB}(\tilde{x}, \tilde{y}) \) then certainly \( e(\tilde{x}, \tilde{y}) = \sum_{k=1}^{L-1} (k - c_i) e(\tilde{x}, \tilde{y}) \).

Proof for \( E-3 \) Enactments. Every \( E-3 \) query \( e \) has a nonredundant decomposition into a sequence of \( E-2 \) enactments, \( e_1^*, e_2^*, \ldots, e_M^* \). This implies that \( e(\tilde{x}, \tilde{y}) = \sum_{i=1}^{M} e_i^*(\tilde{x}, \tilde{y}) \). Moreover, each enactment \( e_i^* \) must have an arithmetic decomposition into \( E-1 \) queries, by the previous part of the proof. The union of these sequences must therefore be an arithmetic decomposition of \( e \). Q.E.D.

The decomposition sequences in Lemma 5.1 will never require more than \( 2^k \) terms when \( e \) contains \( k \) atoms, and they will often be much shorter. The \( O(2^k) \) time to construct a decomposition is relatively unimportant because it is done at compile time.

Proposition 5.2. Each \( E-3 \) enactment \( e(x,y) \) can be represented by a data structure \( D'(Y) \) that uses \( O(N) \) space and has an \( O(1) \) \( WH \)-time for performing the aggregate queries \( \Phi'(\tilde{x}) \), insertions and deletions.

Proof. We will first prove Proposition 5.2 for the canonical \( E-1 \) enactment

\[
e(x,y) = \{ A_1(x) = B_1(y) \land \ldots \land A_J(x) = B_J(y) \} \tag{5.4}
\]

For each \( k \)-tuple \((c_1, c_2, \ldots, c_k)\), let \( Y(c_1, c_2, \ldots, c_k) \) denote the subset of elements \( y \in Y \) satisfying

\[
B_1(y) = c_1 \land B_2(y) = c_2 \land \cdots \land B_k(y) = c_k. \tag{5.5}
\]

Also let \( T(c_1, c_2, \ldots, c_k) \) denote the subtotals \( \sum_{y \in Y(c_1, c_2, \ldots, c_k)} f(y) \). Define \( D'(Y) \) to be a hash data structure [DK88, FKS84] whose keys are \( k \)-tuples of the form \((c_1, c_2, \ldots, c_k)\), and which stores an entry \( T(c_1, c_2, \ldots, c_k) \), for each key \((c_1, c_2, \ldots, c_k)\) that has a nonempty \( Y(c_1, c_2, \ldots, c_k) \). This data structure occupies \( O(N) \) space and supports \( O(1) \) time for an insertion or deletion. Also, one can retrieve \( \Phi'(\tilde{x}) \) in \( O(1) \) time by fetching the item \( T_{\{ A_i(\tilde{x}), A_j(\tilde{x}), \ldots, A_i(\tilde{x}) \}} \) (when it exists) and returning zero otherwise. Proceeding to the case of \( E-3 \) enactments, since each \( E-3 \) query can be arithmetically decomposed into \( E-1 \) enactments, it follows that \( E-3 \) enactments also have the prior \( O(1) \) complexities. Q.E.D.

Examples. Let \( e_1, e_2, e_{12}, e, \) and \( e^* \) denote the enactment queries:

\[
e_1(x,y) = \{ A_1(x) = B_1(y) \} \tag{5.6}
\]

\[
e_2(x,y) = \{ A_2(x) = B_2(y) \} \tag{5.7}
\]

\[
e_{12}(x,y) = e_1(x,y) \land e_2(x,y) \tag{5.8}
\]

\[
e(x,y) = e_1(x,y) \lor e_2(x,y) \tag{5.9}
\]

\[
e^*(x,y) = e_1(x,y) \land \neg e_2(x,y). \tag{5.10}
\]

Since \( e_1 \), \( e_2 \), and \( e_{12} \) are \( E-2 \) queries, it is straightforward to develop hash tables, denoted as \( D'_1(Y), D'_2(Y), \) and \( D'_{12}(Y) \), to answer the on-line aggregation queries \( \Phi'_1(\tilde{x}), \Phi'_2(\tilde{x}), \) and \( \Phi'_{12}(\tilde{x}) \). Also, (5.11) and (5.12) show that \( e \) and \( e^* \) arithmetically decompose into \( E-1 \) enactments,

\[
e(\tilde{x}, \tilde{y}) = e_1(\tilde{x}, \tilde{y}) + e_2(\tilde{x}, \tilde{y}) - e_{12}(\tilde{x}, \tilde{y}). \tag{5.11}
\]

\[
e^*(\tilde{x}, \tilde{y}) = e_1(\tilde{x}, \tilde{y}) - e_{12}(\tilde{x}, \tilde{y}). \tag{5.12}
\]

We can thereby infer that:

\[
\Phi'_1(\tilde{x}) = \Phi'_{11}(\tilde{x}) + \Phi'_{12}(\tilde{x}) - \Phi'_{112}(\tilde{x}) \tag{5.13}
\]

\[
\Phi'_2(\tilde{x}) = \Phi'_{12}(\tilde{x}) - \Phi'_{112}(\tilde{x}). \tag{5.14}
\]

Define \( D'_Y \) to be the union of the hash tables \( D'_1(Y), D'_2(Y), \) and \( D'_{12}(Y) \); and let \( D'_Y \) be the similar union of \( D'_1(Y) \) and \( D'_2(Y) \). In our example, the algorithms for calculating the values of \( \Phi'_1(\tilde{x}) \) and \( \Phi'_2(\tilde{x}) \) consist of two-part procedures that first search the substructures \( D'_1(Y), D'_2(Y), \) and \( D'_{12}(Y) \) to calculate the quantities \( \Phi'_{11}(\tilde{x}), \Phi'_{12}(\tilde{x}) \) and \( \Phi'_{112}(\tilde{x}) \) and then use the arithmetic formulas (5.13) and (5.14) to determine the values of \( \Phi'_1(\tilde{x}) \) and \( \Phi'_2(\tilde{x}) \).
6. THE REPORTING ALGORITHMS FOR E-3 ENACTMENTS

This chapter proves that every E-3 predicate $e(x, y)$ can be represented by an $O(N)$ space data structure $D_e(Y)$ that supports $O(1)$ AH-time for insertions and deletions, and uses $O(1 + U)$ WH-time for on-line queries reporting $Y_e(\bar{x})$ sets (where $U$ again denotes the size of the output). Similar to much of the prior literature on range queries, our algorithms for aggregate and reporting operations are quite different from each other. The reason for this is easiest to appreciate if we return to the example of $e^i(x, y)$ in Eq. (5.10). The reporting analog of (5.14)'s aggregation algorithm would first construct the sets $Y_e^i(\bar{x})$ and $Y_{eg}^i(\bar{x})$ and then use the following setsubtraction operation to construct $Y_{e-g}^i(\bar{x})$.

$$Y_{e-g}^i(\bar{x}) = Y_{ei}^i(\bar{x}) - Y_{eg}^i(\bar{x}).$$

(6.1)

The latter algorithm is correct, but highly inefficient because the cost for subtracting aggregates is quite different from the cost for subtracting sets. The former cost is $O(1)$ under most computing models, but the latter cost is proportional to the cardinalities of the sets undergoing subtraction. It will achieve a magnitude $O(N)$ when at least one of these sets has $O(N)$ cardinality. The point is that the operation (5.14) is a reasonable intermediate step for an $O(1)$ time aggregation algorithm, but the analogous subtraction operation (6.1) is not feasible for an $O(1 + U)$ reporting algorithm because its run time can exceed the bound $O(1 + U)$ under many circumstances. The prior literature in range query theory has often provided quite different algorithms for the aggregation and reporting problems, and this example has illustrated why our reporting algorithms will also need procedures that are quite different.

Our discussion of E-3 searches will be divided into two parts, examining first the NEG-1 predicates and then the aggregation and reporting problems, and this example has illustrated why our reporting algorithms will also need procedures that are quite different.

Our discussion of E-3 searches will be divided into two parts, examining first the NEG-1 predicates and then the aggregation and reporting problems, and this example has illustrated why our reporting algorithms will also need procedures that are quite different.

For each NEG-1 predicate $e$, the preceding data structure uses $O(N)$ space and supports $O(1 + U)$ worst-case time to retrieve $Y_e(\bar{x})$ (where $U$ is the output size).

**Proposition 6.1.** For each NEG-1 predicate $e$, the preceding data structure uses $O(N)$ space and supports $O(1 + U)$ worst-case time to retrieve $Y_e(\bar{x})$ (where $U$ is the output size).

**Proof of Memory Space Asymptote.** Easy, since $D_e(Y)$’s memory space is clearly $O(N)$ when $e$’s negative degree $= 0$, and the memory space of a degree $j$ data structure differs from degree $(j-1)$ structure by no more than approximately a factor $4j^2$ (implying a degree $j$ data structure uses $O(N)$ space with a coefficient proportional to approximately $4/(j!)^2$ in the extreme worst case).

**Proof of Retrieval Time.** The retrieval algorithm will be inductively defined according to the value of $e$’s negative degree. If this degree $= 0$ then $e(x, y)$ corresponds to the degenerate predicate TRUE, and the search algorithm will simply return all the records stored in $H_e(Y)$. If the negative degree $= j > 0$ then the algorithm seeking $Y_e(\bar{x})$ will have two steps. It will begin by checking in $O(1)$ time whether any of the ordered pairs $(i, A_e(\bar{x}))$ are marked for $1 \leq i \leq j$. The procedure will then construct $Y_e(\bar{x})$ by executing Procedure P if some marked pair has been found, and executing Procedure Q if none exists.

**Procedure P.** Choose any ordered pair $(i, A_e(\bar{x}))$ which is marked and search its data structure $D_i(Y, A_e(\bar{x}))$ for the $\bar{y} \in Y_e(A_e(\bar{x}))$ satisfying $\hat{e}(\bar{x}, \bar{y})$ (with a recursively defined procedure). This set constitutes the answer to the query $Y_e(\bar{x})$.

**Procedure Q.** (Used only in the alternate case where no marked pairs are available): Answer the query $Y_e(\bar{x})$ by making a brute-force scan thru the list $H_e(Y)$ that itemizes all $Y_e$’s elements. Return the subset of $\bar{y} \in Y$ found satisfying $e(\bar{x}, \bar{y})$. 

Let $\text{COUNT}(i, k)$ denote the number of elements $\bar{y} \in Y$ having $B_i(\bar{y}) = k$. We will say that the ordered pair $(i, k)$ has a **count-ratio** exceeding $\alpha$ if $\text{COUNT}(i, k) \geq \alpha \cdot \text{CARDINALITY}(Y)$.

We will now show how each NEG-1 predicate $e$ can be represented by a data structure $D_e(Y)$ using $O(N)$ space and having $O(1 + U)$ worst-case time for outputting any $Y_e(\bar{x})$ set of cardinality $U$. Let $H_e(Y)$ denote a trivial doubly-linked list itemizing the elements in $Y$. Define $D_e(Y)$ to equal $H_e(Y)$ if $e$’s negative degree $= 0$. Otherwise define $D_e(Y)$ to have two parts:

1. an $H_e(Y)$ list, and
2. a list of specially “marked” ordered pairs $(i, k)$, each pointing to an inductively defined substructure $D_i(Y_e(\bar{x}))$.

This list must include all ordered pairs whose count ratio exceeds $1/2j$, and Proposition 6.3’s update algorithm will allow it to sometimes include a few other ordered pairs (whose count ratios must exceed $1/4j$). Note there will never be more than $4j^2$ marked ordered pairs (because $e$’s negative degree $= j$).

**Proposition 6.3.** For each NEG-1 predicate $e$, the procedure $P$ (using Procedure P if some marked pair has been found, and executing Procedure Q if none exists) will perform a count-ratio exceeding $\alpha$ if $\text{COUNT}(i, k) \geq \alpha \cdot \text{CARDINALITY}(Y)$.

Let $\text{COUNT}(i, k)$ denote the number of elements $\bar{y} \in Y$ having $B_i(\bar{y}) = k$. We will say that the ordered pair $(i, k)$ has a **count-ratio** exceeding $\alpha$ if $\text{COUNT}(i, k) \geq \alpha \cdot \text{CARDINALITY}(Y)$.

We will now show how each NEG-1 predicate $e$ can be represented by a data structure $D_e(Y)$ using $O(N)$ space and having $O(1 + U)$ worst-case time for outputting any $Y_e(\bar{x})$ set of cardinality $U$. Let $H_e(Y)$ denote a trivial doubly-linked list itemizing the elements in $Y$. Define $D_e(Y)$ to equal $H_e(Y)$ if $e$’s negative degree $= 0$. Otherwise define $D_e(Y)$ to have two parts:

1. an $H_e(Y)$ list, and
2. a list of specially “marked” ordered pairs $(i, k)$, each pointing to an inductively defined substructure $D_i(Y_e(\bar{x}))$.

This list must include all ordered pairs whose count ratio exceeds $1/2j$, and Proposition 6.3’s update algorithm will allow it to sometimes include a few other ordered pairs (whose count ratios must exceed $1/4j$). Note there will never be more than $4j^2$ marked ordered pairs (because $e$’s negative degree $= j$).

**Proposition 6.3.** For each NEG-1 predicate $e$, the procedure $P$ (using Procedure P if some marked pair has been found, and executing Procedure Q if none exists) will perform a count-ratio exceeding $\alpha$ if $\text{COUNT}(i, k) \geq \alpha \cdot \text{CARDINALITY}(Y)$.

Let $\text{COUNT}(i, k)$ denote the number of elements $\bar{y} \in Y$ having $B_i(\bar{y}) = k$. We will say that the ordered pair $(i, k)$ has a **count-ratio** exceeding $\alpha$ if $\text{COUNT}(i, k) \geq \alpha \cdot \text{CARDINALITY}(Y)$.

We will now show how each NEG-1 predicate $e$ can be represented by a data structure $D_e(Y)$ using $O(N)$ space and having $O(1 + U)$ worst-case time for outputting any $Y_e(\bar{x})$ set of cardinality $U$. Let $H_e(Y)$ denote a trivial doubly-linked list itemizing the elements in $Y$. Define $D_e(Y)$ to equal $H_e(Y)$ if $e$’s negative degree $= 0$. Otherwise define $D_e(Y)$ to have two parts:

1. an $H_e(Y)$ list, and
2. a list of specially “marked” ordered pairs $(i, k)$, each pointing to an inductively defined substructure $D_i(Y_e(\bar{x}))$.

This list must include all ordered pairs whose count ratio exceeds $1/2j$, and Proposition 6.3’s update algorithm will allow it to sometimes include a few other ordered pairs (whose count ratios must exceed $1/4j$). Note there will never be more than $4j^2$ marked ordered pairs (because $e$’s negative degree $= j$).

**Proposition 6.3.** For each NEG-1 predicate $e$, the procedure $P$ (using Procedure P if some marked pair has been found, and executing Procedure Q if none exists) will perform a count-ratio exceeding $\alpha$ if $\text{COUNT}(i, k) \geq \alpha \cdot \text{CARDINALITY}(Y)$.

Let $\text{COUNT}(i, k)$ denote the number of elements $\bar{y} \in Y$ having $B_i(\bar{y}) = k$. We will say that the ordered pair $(i, k)$ has a **count-ratio** exceeding $\alpha$ if $\text{COUNT}(i, k) \geq \alpha \cdot \text{CARDINALITY}(Y)$.
The preceding algorithm’s \( O(1 + U) \) WH-time follows from the following three observations:

1. The algorithm satisfies this bound when \( e \)'s negative degree equals zero because it will then use time \( O(N) \), and \( U = N \) in the degenerate case where \( e \) is TRUE.

2. If \( e \) has a negative degree \( > 0 \) and Procedure \( P \) is called, the time bound \( O(1 + U) \) is immediate from the principle of induction.

3. Since an ordered pair \((i, k)\) in the data structure \( D_e(Y) \) is marked whenever \((i, k)\)'s count ratio exceeds \( 1/2 \), Procedure \( Q \) can be called only when simultaneously \( e \) has a negative degree \( j > 0 \) and each of the \( j \) distinct count ratios of \((1, A_1((\vec{x})), 2, A_2((\vec{x})), \ldots, (j, A_j((\vec{x}))) \) fall below \( 1/2 \). The latter condition implies that

\[
U = \text{CARDINALITY}(Y((\vec{x}))) \geq \frac{1 - \frac{j}{2^j}}{} \cdot \text{CARDINALITY}(Y) = \frac{1}{2} \cdot \text{CARDINALITY}(Y) = \frac{N}{2}.
\]  

(6.4)

Since the time for the brute-force scan in Procedure \( Q \) is \( O(N) \), this time must also be bounded by \( O(1 + U) \), by the fact that \( U \geq \frac{N}{2} \).

Q.E.D.

**PROPOSITION 6.2.** Every E-2 and E-3 enactment can also be represented by a data structure \( D_e(Y) \) that occupies \( O(N) \) space and uses \( O(1 + U) \) WH-time to retrieve any \( Y_e((\vec{x})) \).

**Proof for E-2 Predicates.** Every E-2 expression \( e(x, y) \) can be written in the canonical form (6.5), where \( e^*(x, y) \) in that equation is a NEG-l enactment:

\[
e(x, y) = \{ e^*(x, y) \land A_1(x) = B_1(y) \land A_2(x) = B_2(y) \land \cdots \land A_k(x) = B_k(y) \}.
\]  

(6.5)

Let \( Y(c_1, c_2, \ldots, c_k) \) again denote the set of \( \vec{y} \in Y \) satisfying \( B_j(y) = c_j \land B_k(y) = c_k \land \cdots \land B_k(y) = c_k \). Define the data structure \( D_{e^*}(Y) \) to have the following two parts:

1. Its first section will be a collection of Proposition 6.1’s NEG-l data structures of the form \( D_{x^*}(Y(c_1, c_2, \ldots, c_k)), \) such that for each tuple having \( Y(c_1, c_2, \ldots, c_k) \neq \emptyset \), our new data structure \( D_{e^*}(Y) \) will contain one corresponding substructure of the form \( D_{x^*}(Y(c_1, c_2, \ldots, c_k)) \);

2. The second part of \( D_{e^*}(Y) \) is a hash file, denoted as \( H(e) \), whose keys are \( k \)-tuples of the form \( (c_1, c_2, \ldots, c_k) \) and which stores a record for the tuple \( (c_1, c_2, \ldots, c_k) \) if and only if \( Y(c_1, c_2, \ldots, c_k) \neq \emptyset \). This record consists of a pointer to the data structure \( D_{x^*}(Y((c_1, c_2, \ldots, c_k))). \)

The retrieval operation \( Y_e((\vec{x})) \), can then be performed by first hashing to find the substructure \( D_{x^*}(\{ Y(A_1((\vec{x})), A_2((\vec{x})), \ldots, A_k((\vec{x})) \}) \) and then invoking Proposition 6.1’s search algorithm to find the elements \( y \) in this substructure satisfying \( e^*(\vec{x}, y) \).

**Proof for E-3 Enactments:** Let \( \theta_1 \theta_2 \ldots \theta_n \) be a nonredundant decomposition of \( e \) into E-2 terms. Define \( D_e(Y) \) as the union of \( D_{\theta_i}(Y) = D_{\theta_i}(Y) \ldots \) Consider a procedure that retrieves each \( Y_e((\vec{x})) \) from \( D_e(Y) \) and then outputs their union. This procedure runs in \( O(1 + U) \) time and \( D_e(Y) \) occupies \( O(N) \) space, where the coefficients in the O-notation are never more than the sum of the coefficients of the \( e_i \) (they can often be less when further optimizations are applied).

**PROPOSITION 6.3.** The data structures in Propositions 6.1 and 6.2 support \( O(1) \) AH-time for insertions and deletions.

**Proof Sketch.** Our proof focuses on sketching an \( O(1) \) update algorithm for Proposition 6.1’s NEG-1 data structures. The E-2 and E-3 update algorithms for Proposition 6.2’s data structures are the obvious generalizations of the former.

We will add two features to dynamize Proposition 6.1’s data structure. The first will be a hash table \( T \) that uses \( O(N) \) space and stores the values of \( \text{COUNT}(i, k) \) for each key \((i, k)\) that has a nonzero count. The second new feature, called \( L \), will be a list, in descending order, of the integers \( p \) such that some ordered pair \((i, k)\) has \( \text{COUNT}(i, k) = p \neq 0 \). Each entry \((i, k)\) in the hash table will point to \( p \)'s position in this list, and \( p \) will have corresponding reverse pointers. Insertions and deletions in \( Y \) then require only \( O(1) \) WH-time to adjust these data structures when one implements them with a fairly routine algorithm (essentially because insertions and deletions cause \( \text{COUNT}(i, k) \)'s value to change by only one).

Let \( j \) again denote the negative degree of the enactment in Eq. (6.2). Following each insertion and deletion, our algorithm will use the list \( L \) to scan its \( 2^j \) largest ordered pairs for whether there exists any unmarked ordered pair \((i, k)\) whose count-ratio exceeds \( 1/2 \). If such an element is found, the algorithm will scan \( O(N) \) time building its \( D_{\theta_i}(Y((\vec{x}))) \) data structure (and thereby “marking” \((i, k)\)). Also, the algorithm will spend \( O(N) \) time deallocating the \( D_{\theta_i}(Y((\vec{x}))) \) data structure of any previously marked ordered pair whose count-ratio falls below \( 1/4 j \). The only further work of the algorithm will consist of some straightforward bookkeeping and applying the preceding “mark” and “deallocate” rules recursively to every \( D_{\theta_i}(Y((\vec{x}))) \) data structure whose negative degree \( > 0 \). More details about the algorithm are omitted for the sake of brevity. Essentially, the mark and deallocate operations are the only aspects of the algorithm that can exceed \( O(1) \) WH-time, and an
amortization proof will show these operations have an \( O(1) \) 
AH-cost. (The coefficient here should be usually small, but 
it can be approximately as large as \( 4^{j+1}((j+1)!)^2 \) in 
the extreme worst case.)

Q.E.D.

In \[ Wi78 \], a substantially more complicated version of
Proposition 6.3’s algorithm was presented that also guaranteed
\( O(1) \) WH-time. Our main interest in Propositions 5.1 
through 6.3 arises because they imply that aggregation and 
reporting joins run in linear time and space when 
\( d(e) = 0 \). With further efforts, the preceding propositions can have 
their coefficients reduced for the special case of joins, but 
there is insufficient space for that topic here.

7. THE E-7 REPORTING ALGORITHMS

In this section, \( A(x) \) will usually be a real number, but it 
can also equal \(-\infty \) or \( +\infty \). A \( y \)-range term is an expression of the form

\[
A_1(x) \prec B^*(y) \prec A_2(x).
\]  

(7.1)

The letter \( B^* \) is called Eq. (7.1)’s \textit{range attribute}. Define 
an E-4 term to be an enactment that is either an E-3 
predicate or the conjunction of an E-3 predicate with several 
\( y \)-range terms, each using a different range attribute \( A_1, A_2, \ldots \). Let \( e^*(x, y) \) denote an E-4 predicate of degree 
\( d(e^*) = k - 1 \) and let the symbol \( e \) denote the degree = \( k \) 
predicate defined

\[
e(x, y) = e^*(x, y) \land A_1(x) < B_0(y) < A_2(x).
\]  

(7.2)

The range tree theorem of Bentley \[ Be80 \] implies that one 
can build a data structure \( D_N(Y) \) out of lower order data 
structures of the type \( D_{N_0}(\cdot) \) with only a factor \( \log N \) 
increase in retrieval time and memory space, and Lukeler and Willard \[ LW82, WL85 \] have shown how this method 
can also be made dynamic. The previous literature can be 
summarized as follows:

Define a range-tree representation of \( Y \) to be a two-part 
data structure. Its first section, called the \textit{base}, will be a 
binary tree of height \( \log N \) whose leaves are the elements of 
\( Y \) arranged in order of increasing \( B_i(y) \) value. Henceforth, 
\( YSET(v) \) will denote the subset of \( Y \) descending from the 
tree node \( v \). The range tree will assign each node \( v \) a pointer 
to an “auxiliary” data structure \( AUX(v) \), consisting of an 
alternate representation of \( YSET(v) \). The version of range 
trees for E-4 enactments in \[ Wi78 \] had set \( AUX(v) = D_{N_0} \) 
\( YSET(v) \). Its retrieval algorithm for performing query 
(7.2) consists of the following two steps:

1. Define a node \( v \) to be \textit{critical} with respect to (7.2) if 
every \( y \in YSET(v) \) satisfies \( A_1(x) < B_0(y) < A_2(x) \) and \( v \)’s 
father does not meet this criterion. Use a binary search to 
find the \( O(\log N) \) or fewer critical nodes in the range tree in 
\( O(\log N) \) time.

2. For each critical node \( v_i \), search its \( AUX(v_i) \) field for 
the \( y \) satisfying \( e^*(x, y) \). Let \( Y_i \) denote the set of tuples 
returned by this search. The sets \( Y_1, Y_2, \ldots \) are disjoint, 
and the answer to the query (7.2) is found by taking their union.

An easy induction proof shows that the above data 
structure for E-4 enactments occupies \( O(N \log^{d(e)} N) \) space 
and allows on-line reporting queries to run in time 
\( O(U + \log^{d(e)} N) \). The similar \( O(\log^{d(e)} N) \) AH-time for 
insertions and deletions follows from the combination of 
Proposition 6.3 and \[ WL85 \]’s method for rebalancing 
range trees.

It is also possible to devise a memory-compressed variant 
of a range tree that uses linear space and \( O(N \log^{d(e)} N + U) \) 
time for doing reporting joins over E-4 enactments. The intuitive idea is to use a method, similar to \[ Be80, EO85 \], 
where we employ a modified range tree such that each 
\( AUX(v) \) structure is constructed for only a short period of 
time, and all the \( x \) in \( X \) that need to search \( AUX(v) \) 
are required to do so during this short period.

In order to define this algorithm formally, let the terms 
\( YSET(v) \), critical node, and base tree have the same 
definitions as before. We introduce the following new definitions:

(i) \( XCRIT(v) \) is the set of \( x \in X \) which have \( v \) as a critical 
ode.

(ii) \( XSET(v) \) is the union of \( XCRIT(v) \) with the 
\( XCRIT(v) \) sets of \( v \)’s descendants in the base. (There is an alternate 
equivalent definition of \( XSET(v) \) which some readers 
may prefer. Let \( Range \) \( (v) \) denote the half-open interval 
\( (L_v, U_v] \) such that all \( y \)-records descending from \( v \) have 
their \( B_i(y) \) value lying in this interval. Then \( XSET(v) \) 
can be equivalently defined as those \( x \in X \) where \( (A_1(x), \ldots, A_2(x)) \) 
and \( (L_v, U_v] \) have a nonempty intersection.)

(iii) The base tree \( T \) is said to be \textit{j-specialized} 
iff only nodes of depth \( j \) have auxiliary fields and these fields consist 
of two lists itemizing the elements in \( XSET(v) \) and \( YSET(v) \).

Define \textbf{TRANSFORM} \( (j) \) to be a procedure that:

(A) spends \( O(N \log N) \) time building a base tree \( T \) 
and putting it in a zero-specialized state when \( j = 0 \);

(B) converts a \( (j-1) \)-specialized base into a \( j \)-specialized 
state in \( O(N) \) time when \( j > 0 \).

It is easily to develop algorithms for \( A \) and \( B \) that run in 
\( O(N \log N) \) and \( O(N) \) times. (Their formal 
expressions have been omitted for the sake of brevity.) Assume \( e \) 
is related to \( e^* \), as shown in Eq. (7.2). Then the algorithm for 
executing \( e(x, y) \)’s reporting join will consist of a \textbf{LOOP}
FROM $j = 0$ TO $\log N$, which performs the following two steps for each $j$:

1. Apply the procedure TRANSFORM($j$) to build a $j$-specialized base.

2. Make a straightforward scan over all the $x \in XSET(v)$ of each depth $=j$ node to construct the corresponding XCRIT($v$) sets in $O(N)$ time. Then execute a recursively defined reporting join to find all the $(x, y) \in$ XCRIT($v$) $\times$ YSET($v$) satisfying $e^*(x, y)$. Let LIST($v$) denote the output of this operation; the LIST($v$) sets for $v \in T$ are disjoint and their union forms the answer to $e^*$'s reporting join.

**PROPOSITION 7.1.** *The reporting join of an E-4 enactment of degree $d$ runs in $O(N \log^d N + U)$ WH-time and $O(N)$ space.*

Proof Sketch. If $d(e) = 0$ then the result follows from the previous section. Otherwise, $e(x, y)$ will appear in the canonical form (7.2), and we can inductively assume that $e^*$ satisfies the claim. The preceding algorithm was designed so that $e^*$'s runtime will be a factor $O(\log N)$ slower than the processing time for $e^*$, by essentially the usual range tree argument. Moreover, the reporting join algorithm will use only $O(N)$ space essentially because its LOOP requires only one depth level of auxiliary fields to be constructed during each particular iteration. (Note that the $O(N)$ space includes a small coefficient proportional to $d(e)$, so that there is adequate space to recursively process $e^*$ in Step 2 of the LOOP.)

Q.E.D.

Comment. [EO85] has shown that the join time can be reduced to $O(N \log^{d-1} N + U)$ when $d(e) \geq 2$ and the E-4 enactment has no equality atoms. Proposition 7.1's result can be transformed into a worst-case bound, when $d(e) \geq 1$, by essentially using fractional cascading [CG86, MN90]. Also from the combination of [Be80, BS77] and the lexicographic reductions in the next section, we will be able to infer that E-4 aggregate joins run in an analogous $O(N \log^{d-1} N)$ worst-case time and $O(N)$ space when $d(e) \geq 2$.

Say a list atom is an x-unit if it specifies a subset of $X$ (i.e., it is an expression of the form $x \in L$). Define an x-unit to be any concatenation of such atoms using the AND, OR, and NOT connectives (including the Boolean constant TRUE designating the empty concatenation). Define y-units similarly. The symbols $\alpha(x)$ and $\beta(y)$ shall denote $x$ and $y$-units. Also, if $e^*(x, y)$ is an E-4 enactment, define $e^*(x, y) \land \alpha(x)$ and $e^*(x, y) \land \beta(y)$ to be respectively E-5 and E-6 enactments. Recall that E-7 enactments were defined to consist of equality, order, and list atoms, combined in an arbitrary manner by AND, OR, and NOT connectives.

**PROPOSITION 7.2.** *Each of the E-5, E-6, and E-7 classes of degree $d(e)$ have the same time–space complexities as the E-4 enactment terms of the same degree, for the four problems of on-line reporting, join reporting, on-line aggregation, and join aggregation.*

Proof. Similar reasoning applies to all four tasks. Our proof considers only on-line reporting. It will first consider E-6 enactments and then examine the more general E-7 class.

**Proof for E-6 Enactments.** In the context of the definition of E-6, let $Y^* = \{ y \in Y | y$ satisfies $\beta(y) \}$. Define $D_0(Y^*)$ to be the data structure $D_0(Y^*)$. A retrieval algorithm seeking to construct $Y_0(x)$ will begin by calculating in $O(1)$ time the Boolean value for $\alpha(x)$ (by essentially employing hashing to test for $x$'s membership tests in the relevant lists $L_1, L_2, L_3, ...$). If $\alpha(x) = \text{FALSE}$, it will return NULL as the answer to the query. Otherwise, it will use the E-4 formalism to find the subset of $D_0(Y^*)$ satisfying $e^*(x, y)$ and return this set as the answer to the query asking for $Y_0(x)$. This algorithm implies that E-4 and E-6 on-line reporting queries have the same retrieval complexity, and a similar argument shows they have the same insertion and deletion costs.

**Proof for E-7 Enactments.** An immediate generalization of the E-6 case, since every E-7 enactment of degree $d$ has a nonredundant decomposition into E-6 terms of degree $d$.

Q.E.D.

The preceding proof was very simple. Some readers may wonder why it was necessary to devote even a short proof to this subject. The reason is that the algorithms in [Wi84,Wi90] typically interject list atoms into the intermediate stages of their computations, to reduce a complicated $k$-variable query into a more efficient sequence of 2-variable enactment join operations. For example, query (3.3)'s algorithm employed such interjections, and the faster algorithm for processing (3.6) employed them to construct $V^*$ and $W^*$. The generalized form of list-atom interjection, called a QL-reduction in [Wi90], is necessary to process most RCS queries. The main anticipated application of Propostion 7.2 will be for the QL-reductions required to process RCS-like expressions.

### 8. LEXICOGRAPHIC REDUCTIONS AND OTHER OPTIMIZATIONS

The term E-4A predicate will refer to an enactment that is either an E-1 predicate or the conjunction of an E-1 predicate with several $\gamma$-range terms. We will say that an E-4A, enactment has order $(k, d)$ if it contains exactly $k$ equality atoms conjuncted with exactly $d$ $\gamma$-range terms.
Equation (8.1) is an example of an E-4A predicate of order (2, 2):
\[
A_j(x) = B_j(y) \land A_j(x) = B_{j+2}(y) \land A_{j+1}(x) < B_{j+2}(y) < A_{j+1}(x) \land A_j(x) < B_j(d) < A_j(x).
\tag{8.1}
\]

Also the term orthogonal range query of dimension \(d\) refers to an E-4A enactment with \(d\) \(y\)-range terms and no equality atoms (similar to the two-dimensional query below),
\[
A_j(x) < B_j(y) < A_j(x) \land A_j(x) < B_j(d) < A_j(x).
\tag{8.2}
\]

An interesting fact is that each \(d\)-dimensional orthogonal query algorithm can be modified to process E-4A enactments of order \((k, d)\) without a degradation in efficiency. This algorithmic transformation is easiest to illustrate by example. Let \(G = (g_1, g_2, g_3)\) and \(H = (h_1, h_2, h_3)\) denote two ordered triples, and say \(G < H\) when one of the following conditions hold:
1. \(g_1 < h_1\)
2. \(g_1 = h_1 \land g_2 < h_2\), or
3. \(g_1 = h_1 \land g_2 = h_2 \land g_3 < h_3\).

In this notation, it is apparent that the E-4A query of order \((2, 2)\) in Eq. (8.1) is equivalent to the two-dimensional orthogonal range query
\[
\{A_1(x), A_2(x), A_3(x)\} < \{B_1(y), B_2(y), B_3(y)\}
\leq\{A_1(x), A_2(x), A_3(x)\} \land A_3(x)
\leq B_3(y) < A_3(x)
\tag{8.3}
\]

Thus, a computer programmer need not write special software to handle Eq. (8.1), since he can instead borrow from the literature on orthogonal queries to process the equivalent (8.3).

The term lexicographic reduction will refer to this method for applying \(d\)-dimensional orthogonal range query algorithms to process an E-4A enactment of dimension \((k, d)\). Define an E-6A enactment to be a conjunction of an E-4A enactment with an \(x\)-unit and a \(y\)-unit. Lexicographic reductions obviously generalize to E-6A enactments, via Proposition 7.2’s method. An important distinction, however, is that every E-7 enactment can be arithmetically decomposed into a sequence of E-6A enactments, but the analogous disjunctive decomposition will often not exist. The former decomposition facilitates aggregation queries, while the latter is intended to process reporting queries. Sometimes, the reporting algorithms will be quite different from the aggregation algorithms for this reason.

In particular, consider the aggregation and reporting join problems. Edelsbrunner and Overmars have shown that a batch of \(d\)-dimensional orthogonal reporting queries are executable in linear space and \(O(N \log^{d-1} N + U)\) time when \(d \geq 2\), and the ECDF method [Be80] can be adapted to provide a similar \(O(N \log^{d-1} N)\) time for the comparable aggregation problem. For \(d(e) \geq 2\), the lexicographic reduction principle thus implies one can calculate an aggregate join in time \(O(N \log^{d(e)-1} N)\) for any E-7 enactment.

But the same disjunctive decomposition is not available for reporting joins because of the potential absence of the needed disjunctive decomposition sequences! Indeed, it is an open question whether the analogous \(O(N \log^{d(e)-1} N + U)\) time is possible for all E-7 reporting joins. Rather the best known join algorithm follows from the previous section and requires \(O(N \log^{d(e)-1} N + U)\) time for a worst-case enactment \(e\).

Similarly, the dynamic reporting data structure \(D_j(Y)\) from the previous section requires \(O(N \log^{d(e)-1} N)\) for the hardest enactment \(e\), whereas the lexicographic reduction technique, combined with [Wi87]’s method, enables \(O(N((\log N)!(\log \log N))^{d(e)-1})\) space data structures to support \(O(\log^{d(e)}(N))\) time for aggregate retrievals and updates when \(d(e) \geq 1\). Indeed, when \(d(e) \geq 2\) and the aggregate query is a COUNT rather than an abelian semigroup SUM request, the preceding memory space can be reduced to \(O(N \log^{d(e)-2} N)\) under the combination of Chazelle’s data structure [CH88] and lexicographic reductions. Also, for static on-line aggregate queries, the option is available to either reduce the search time by a log \(N\) factor (in the group-SUM model) [Wi85], or to reduce the memory space to \(O(\log^{d(e)-2+\epsilon} N)\) [Ch88], for any \(\epsilon > 0\).

A very large number of this paper’s data structures can speed up slightly further in theory by using fusion trees and Dietz’s fast-counting method [AH95, Di92, FW90, Wi92], but it appears such speedups have little pragmatic significance. The literature on orthogonal range queries [Be75, Be80, BM80, BS77, BS80, Ch86, Ch88, CG86, Ed81, EO85, LW77, LW80, LW82, MN90, OL81, Ov88, OS90, Sm89, Wi87, Wi85, Wi86, Wi87, Wi92] has displayed a large number of on-line algorithms whose performance depends on the size of the memory space, the ease of performing insertions and deletions, the model of computation, etc. All these results and trade-offs obviously generalize through lexicographic reductions to all E-7 aggregations and to the subset of reporting enactments that are either E-6A enactments or have an efficient disjunctive decomposition into E-6A components. In particular, while the results from the previous chapter are the best known method for the hardest E-7 reporting enactment of arbitrary degree, some types of improvements are possible for the E-6A subclass by applying one of [Ch86, Ch88, Ed81, EO85, CG86, MN90, Wi85, Wi92] in the context of...
9. THE E-8 ENACTMENTS

The E-8 class was defined in Section 1; it differs from E-7 enactments by additionally including tabular atoms. In this section \( e(x, y) \) denotes an E-8 enactment, and \( e(x, y) \) the E7 enactment begotten by replacing each of \( e' \)'s tabular atoms with the Boolean constant of FALSE. The enactment \( e_x \) is called a \( \pi \)-reduction of \( e \), and an example is

\[
e(x, y) = \{(A_1(x) > B_1(y) \land (x, y) \in T_1) \}
\]

\[
\lor (A_2(x) = B_2(y) \land \text{NOT} \ (x, y) \in T_2)\}
\]

\[
e'(x, y) = \{(A_1(x) > B_1(y) \land \text{FALSE}) \}
\]

\[
\lor (A_2(x) = B_2(y) \land \text{TRUE})\}
\]

Our algorithm for performing reporting and aggregation joins over E-8 enactments will essentially be a two-phase method. It will first employ the E-7 formalism to perform a join on the reduced predicate \( e_x(x, y) \) and then use this information to help formulate the answer to the query \( e(x, y) \). In order to formally define this method, let \( \text{ADD}(X, Y, e) \) and \( \text{SUBTRACT}(X, Y, e) \) denote the two subsets,

\[
\text{ADD}(X, Y, e) = \{(\bar{x}, \bar{y}) \in X \times Y \text{ satisfying } e(\bar{x}, \bar{y}) \land \text{NOT } e_x(\bar{x}, \bar{y})\}
\]

\[
\text{SUBTRACT}(X, Y, e) = \{(\bar{x}, \bar{y}) \in X \times Y \text{ satisfying } e_x(\bar{x}, \bar{y}) \land \text{NOT } e(\bar{x}, \bar{y})\}
\]

2 Let \( \partial e \) denote the number of distinct attributes of \( x \) appearing in \( e(x, y) \)'s order predicate; i.e., \( \partial e \) is the analog of \( \partial e \) which counts \( x \)- rather than \( y \)- attributes. Let \( X_e(\bar{y}) \) denote the subset of \( x \in X \) satisfying \( e(\bar{x}, \bar{y}) \). Unlike on-line queries and aggregate joins, the reporting join problem is fully symmetrical; i.e., it can be absorbed by either constructing the array \( Y_e(\bar{x}) \) for \( \bar{x} \in X \), or the array \( X_e(\bar{y}) \) for \( \bar{y} \in Y \). Since often \( \partial e \neq \partial e \), a designer of database software should certainly assure his algorithm selects whichever computation is more efficient. An ideally optimal software package, indeed, should also allow for the possibility of hybrid solutions that do not involve strictly computing either \( Y_e(\bar{x}) \) or \( X_e(\bar{y}) \). For example, it is possible that \( \partial d = \partial e \), but the reporting join can be computed in \( O(N \log N + U) \) time by using the fact that \( e \) has a disjunctive decomposition into \( e_1 \land e_2 \) and that \( Y_{e_1}(\bar{x}) \) and \( X_{e_2}(\bar{y}) \) can be calculated in \( O(N \log N + U) \) time. Also, in contexts where several iterations of Propositions 7.1's algorithms are nested within one another, it can be desirable for selected iterations to interchange the roles \( x \) and \( y \) play in Eq. (7.2) in the middle of the iterated search.

We will use the \( \text{ADD}(X, Y, e) \) and \( \text{SUBTRACT}(X, Y, e) \) sets in the proof of Proposition 9.1 to infer the answer of an \( e(x, y) \) query from the answer of an \( e_x(x, y) \) query.

PROPOSITION 9.1. Let \( M \) denote the number of ordered pairs stored in the tables \( T_1, T_2, T_3, \ldots \) of the E-8 enactment \( e \), and let \( U \) and \( U_e \) denote the sizes of outputs from \( e \) and \( e_x \)'s reporting joins. Assume \( O(N \log^d N) \) and \( O(N \log^d N + U_e) \) are the costs for \( e_x \)'s aggregate and reporting joins. Then the corresponding join costs for \( e \) are \( O(N \log^d N + M) \) and \( O(N \log^d N + U + M) \).

Proof. The algorithms for performing the aggregation and reporting joins are quite similar procedures, consisting essentially of the following three steps:

1. First perform the join operation over the reduced predicate \( e_x(x, y) \) by invoking the E-7 search procedures. This step will produce a set \( \text{START} \), listing all the ordered pairs \((\bar{x}, \bar{y})\) satisfying \( e_x(\bar{x}, \bar{y}) \) for the case of a reporting join. It will output an array \( \Phi(\cdot) \) satisfying \( \Phi_x(\bar{x}) = \Phi(\bar{x}) \) for the case of an aggregate join.

2. Next construct the two sets \( \text{ADD}(X, Y, e) \) and \( \text{SUBTRACT}(X, Y, e) \). This step can be executed in \( O(M) \) time by first building a table \( T = T_1 \cup T_2 \cup T_3 \ldots \) and then checking each \((\bar{x}, \bar{y})\) in this table for whether it satisfies the two conditions \( e(\bar{x}, \bar{y}) \) and \( e_x(\bar{x}, \bar{y}) \). Those ordered pairs satisfying only the first condition are put into the set \( \text{ADD}(X, Y, e) \). Similarly, \( \text{SUBTRACT}(X, Y, e) \) shall store those ordered pairs satisfying only the second condition.

(We note we need only examine the members of \( T \) to construct \( \text{ADD}(X, Y, e) \) and \( \text{SUBTRACT}(X, Y, e) \) because only the \((\bar{x}, \bar{y}) \in T \) can satisfy \( e_x(\bar{x}, \bar{y}) \neq e_x(\bar{x}, \bar{y}) \).)

3. Finally, combine the information from steps 1 and 2 to answer the join query. For the case of a reporting join, the answer is produced by constructing the set

\[ \text{START} \cup \text{ADD}(X, Y, e) - \text{SUBTRACT}(X, Y, e). \]

For the case of an aggregation join, we take the array \( \Phi(\cdot) \), calculated in step 1, and change its stored values from \( \Phi_x(\bar{x}) \) to \( \Phi(\bar{x}) \), by executing a LOOP that

(a) increases \( \Phi(\bar{x}) \) by an amount \( f(\bar{y}) \) for each \((\bar{x}, \bar{y})\) in the set \( \text{ADD}(X, Y, e) \); 

(b) decreases \( \Phi(\bar{x}) \) by \( f(\bar{y}) \) for each \((\bar{x}, \bar{y})\) in \( \text{SUBTRACT}(X, Y, e) \).

The join procedures (above) are correct because the \( \text{ADD} \) and \( \text{SUBTRACT} \) sets contain the only ordered pairs where \( e(\bar{x}, \bar{y}) \neq e_x(\bar{x}, \bar{y}) \). We will do the time analysis for the reporting join first. Using Proposition 9.1's notation, the three steps of the reporting join run in times \( O(N \log^d N + U_e) \), \( O(M) \), and \( O(M + U_e) \), respectively. This implies
a combined time $O(N \log^d N + U_x + M)$ for the whole algorithm. Moreover, it must be the case that $U \geq U_x - M$ (because the outputs for $e$ and $e_u$, can disagree only on ordered pairs lying in $T$). The latter inequality implies $O(N \log^d N + U_x + M) \leq O(N \log^d N + U + M)$. Hence, $e$'s reporting join runs within the time asymptote claimed by Proposition 9.1. The similar result for aggregation joins follows because steps 2 and 3 of the aggregation join run in $O(M)$ time. Q.E.D.

Proposition 9.1 can be obviously generalized to on-line queries. 3 Proposition 9.1 is intended for applications where $M \ll N^2$ and $U \ll N^2$, and its algorithm is more efficient than the $O(N^3)$ brute force walk through the cross product space $X \times Y$ under such circumstances. There are many examples in database applications where these two sparsity conditions hold. The discussion of Eqs.(3.9)- (3.11), at the end of Section 3, explained how the notion of a tabular atom substantially enhances the mathematical and expressive powers of the E-8 and RCS formalisms.

The underlying purpose of the E-1-E-8 search algorithms was to serve as subroutines to enable the efficient quasi-linear processing of RCS-like relational calculus queries. Previous more informal descriptions of these papers have appeared in the conference papers [Wi84,Wi90]. Paige et al. have very generously credited [Wi78,Wi83,Wi84] for having partially influenced some facets of their evolving implementation of SETL. [Pa79, Pa84, PH86, PK82, CP78], called RAPTS. At present, RAPTS is the only experimental project that has attempted to use a portion of our proposed RCS and E-8 formalisms. Some readers may wish to examine it. We would also be very pleased to give future software implementers advise about how the E-8 search algorithms and [Wi90]'s related RCS decomposition algorithms can be implemented in a very pragmatic manner in relational database applications.

ACKNOWLEDGMENTS

I am very grateful to Ugo Gagliardi for suggesting that I study relational database optimization. I thank Amir Ben-Amram, Robert Paige, and Asaph Zemach for their comments on the presentation.

REFERENCES


APPLICATIONS OF RANGE QUERY THEORY


