EFFICIENT PROCESSING OF RELATIONAL CALCULUS EXPRESSIONS USING RANGE QUERY THEORY

(Extended Abstract)

by
Dan E. Willard*
State University of New York at Albany
Albany, New York 12222
and consultant to
Bell Communications Research

Goal

In this paper we define a language based on a broad subset of the relational calculus and show all expressions in this language can be evaluated in space O(N) and time $O(N \log^d N)$, where d is a small constant whose value depends on the particular predicate and where N is the number of records stored in the data base. We currently have no hard statistics, but a reasonable guess seems to be that a standard sequential random access machine can handle 95% or more of commercial requests in time O(N log N) and memory O(N) using this technique.

1. Introduction

During the last decade two separate theories about retrieval have been evolving. The first is relational theory [Co70,Ul82]; it seeks to develop easy-to-use nonprocedural data base systems. The second theory is quite new [Be75, Be80, BM80, BS77, BS80, CY83, Ed81, EO83, EW83, Fr81, LW77, LW80, OL82, Wi78a, Wi78b, Wi78c, Wi82, WL84, Ya82, Ya83], and it has recently been called Range Query Theory. It develops particular data structures for executing specific queries efficiently. The first attempt to bridge these two approaches appeared in Willard's dissertation [Wi78a]. In Section 7.5, Willard defined a subset of the relational calculus henceforth called RCS, and indicated each such expression could be evaluated by using specific algorithms from the other sections of Willard's dissertation. The result was that for each RCS relational calculus expression there would exist a constant d such that the expression could be evaluated in $O(N \log^d N)$ time and space. Initially, this result did not appear to be particularly practical because $O(N \log^d N)$ is a large amount of space. Since that time, [BC81, BG81a, BG81b, Da83, GS82, JK83, WY76, YO79, Ya81] have shown how to execute certain manageable but much smaller subsets of relational queries in at least moderately good time and space, various articles have studied the purely logical aspects of acyclic data base schemas [BFMMUY81,Fa83,Hu83,Li82,Za76] and Paige [Pa84] independently reconstructed two of the approximately ten techniques from Willard's dissertation [Wi78a] (which correspond to COUNT but not all FIND retrievals on the subset of E-8 predicates built out of equality and tabulations). At

*In 1976-1978, this work was partially supported by ONR grant N00014-76-C-0914 while the author was at Harvard. Theorem 7.5L of [Wi78a] describes the variation of our result that was derived at that time.

the same time among the advances in range theory were that Willard [Wi78b] and later [Ed81,Ch83] reduced the time for orthogonal range queries by a factor of log N using quite different algorithms that had distinctly different implications in a dynamic environment, and Edelsbrenner and Overmars [EO83] developed a memory reduction technique that is a generalization of a memory reduction in Bently-Shamos [BS77]. This reduction applies only to batch processes, but these turn out to be precisely the queries needed to reduce from $O(N \log^d N)$ to $O(N)$ the memory for the aspects of Willard's dissertation relevant to RCS. A large number of data base scientists (over fifteen authors in the articles just cited) have predicted most relational applications will be acyclic. If they are correct, then the algorithms herein, which correspond to the static portion of Willard's dissertation improved with the methods of [EO83,Wi78b], are significant. Bear in mind that the coefficient associated with our space asymptote $O(N)$ can become as small as $(1+\epsilon)$ and that d in our time $O(N \log^d N)$ is an extremely small integer almost always ≤ 1 because we treat equality predicates differently from order predicates.

A classification of terminology: The terms "decomposition" and "decomposable searching" will often be used in this paper. They should not be confused and actually refer to entirely different concepts. The former refers to a process of splitting one relational language query into a sequence of subqueries. The latter views data structures as the primitive objects and attempts to perform more complicated queries by combining data structures, sometimes recursively. Examples of articles on relational language decomposition include [BC81, BG81a, BG81b, GS82, WY76, YO79, Ya81], and examples of articles on decomposable data structures are [Be80, BS80, EO83, OL82, Wi78a, WL84]. The data structures in the decomposable searching theory are sometimes defined recursively within it [BS77, Be80, LW80, Wi78b, WL84], other times stem from traditional search theory [AHU74, Kn73, NRD78, HS78a, HS78b], they may come from multidimensional search theory [BS75, Be75, BM80, CY83, EW83, LW77, Wi78c, Wi82, Ya83], and some of work of Paige et al. [GP84, Pa82, Pa84, Pk82] should be considered decomposable data structure theory although it uses a different terminology. The final result of this paper was announced in Theorem 7.5L of Willard's dissertation [Wi78a] except for a larger memory space, but it is perhaps easier to think of the RCS algorithm as the first synthesis of the decomposable and decomposition searching schools which uses [BS77 or Be80, EO83,Wi78b] to improve [Wi78a]'s main theorem.

The present conference paper is only a condensation of material discussed in much greater detail in our unabridged 80-page report [Wi83b]. This conference paper differs from our other conference paper [Wi83a] by focusing on decomposability rather than decomposition, and it is deliberately less efficient than our detailed report [Wi83b] for the sake of simplifying the presentation. A third conference paper by Willard [Wi84a] complements all our other results by explaining the importance of drawing random samples of size $\Xi N^{2/3}$. Among all the papers cited in this paragraph, the present paper is the best introduction to our approach, and it is probably sufficiently simple for classroom presentation.

2. Overview of Paper

Throughout this paper, S will denote the set of tuples printed by a query, and |S| will designate its cardinality. We will measure complexity in terms of a time measure, defined in [Wi78b,Wi81], called worst-case locate time. A query that prints |S| distinct tuples in an actual time $\leq O(F+|S|)$ will be said to have a worst-case locate cost $O(F)$. Many recent articles (for instance [Be80, Ch83, EW83, LW80]) are now using this complexity measure because it is unavoidable to spend $O(|S|)$ time printing a list of |S| distinct tuples. Intuitively, locate time accounts for the overhead beyond the trivial and unavoidable printing (or output) costs.

Our main theorem will define a language called RCS and prove all queries in this language are doable in locate time $O(N \log^d N)$ and space $O(N)$, where N is the amount of storage used by the data base and d is some constant whose value depends on the particular query typically ≤ 1.

Although our theorem is stated as a locate rather than strict worst-case result, these two complexities actually reduce to the same quantity

for many RCS queries because the output size will be $\leq N$ when the retrieve clause is a COUNT, SUM, TEST, or 1-variable FIND statement (see next section). For such cases, $O(N \log^d N)$ locate time implies $O(N \log^d N)$ strict worst-case time.

We will use the term quasi-linear time to describe the asymptote $O(N \log^d N)$ and our paper largely addresses the emerging electronic technologies where it will be cost-effective to store an entire dataase of N elements in the main memory. In such a technology, a quasi-linear number of executed CPU operations will be within the realm of practicality because it amounts to only a few seconds of computer time. Our main theorem will not assume any specialized hardware such as systollic chips; it will show a standard random access machine without parallel processing abiities is adequate for a language as broad as RCS. This result should become increasingly practial as the cost of random access memory chips drop in the future since the exponent is typically ≤ 1. The theorem is significant because the queries processed are very complicated relational requests. Except for a larger memory space, our result was stated six years ago in Theorem 7.5L of [Wi78a].

3. Statement of the Query Problem

The acronym RCS stands for Relational Calculus Subset. All relational calculus expressions are not included in this language because some of the conjectures in NP-completeness theory imply $\forall k \exists j$ such that some j-variable calculus expression requires more than N^k locate time. Instead RCS is an attempt to define a subset of the relational language which is simultaneously broad enough to include most of the requests likely to come from the commercial user while narrow enough to exclude the intractible expressions. To make RCS broad enough, we incorporate into this language calculus expressions that would not be considered as either tree or acyclic-based in the relational literature mentioned in section 1. Our algorithm will execute all RCS expressions in space $O(N)$ and time $O(N \log^d N)$, for some constant d on a database of size N.

Following the terminology in data base theory, the term relation refers to a table that has j rows and m columns. A record corresponds to a row in this table and is often called a tuple. A particular column field of the record x is called an attribute and is denoted x.a. A data base schema is defined as a collection of several relations, and the terms Rx, Ry and Rz will denote the relations over which the record variables x, y and z span. $|R|$ will denote the size of the relation R, i.e. its number of rows; the cardinality of the data base consisting of the relations $R_1 \ldots R_k$ will be defined as $\Sigma|R_i|$, and it will be denoted as simply N.

Each relation will be assumed to contain one attribute which uniquely identifies it. Codd calls this attribute a primary key, and its defining characteristic is that no two records store the same value in this field. We will denote x's primary key as x.prime. For reasons explained at the end of this section, the expressive power of our language will be enhanced substantially if we divide all relations into two categories called esets and associations (between esets), following the terminology of the entity-relationship model. Five types of atomic predicates will be used in this paper. These are defined below:

[1] Equality Predicates: These are expressions of the form x.a = y.b where Rx and Ry are relations of type eset.

[2] Order Predicates: These are expressions of the form $x.a \geq y.b$ or $x.a > y.b$ where Rx and Ry are relations of the type eset.

[3] Unary Comparison Predicates: These are expressions of one of the forms $x.a = c$, $x.a > c$, $x.a \geq c$, $x.a_1 = x.a_2$ or $x.a_1 > x.a_2$ where Rx is again an eset and c is a constant.

[4] Unary List Predicates: These are expressions of the form: $\exists y \in Ry: y.a = x.prime$ where Rx is an eset and Ry an association.

[5] Tabular Predicates: These are expressions of the form:

$$\{\exists z \in Rz: z.a = x.prime \text{ and } z.b = y.prime\} \quad (3.1)$$

where Rx and Ry are relations of type eset and Rz is an associative relation.

The symbols $u_1(x)$, $u_2(x)$... and $T_1(x,y)$, $T_2(x,y)$... will denote unary list and tabular atomic predicates respectively; at the end of this section we will explain how these concepts and the distinction

between associations and esets has major implications on performance.

An unquantified normal predicate will be defined as a logical expression which concatenates a series of atomic predicates with the logical connectives of AND, OR and NOT. For example if x,y, z and w are variables ranging over esets then equation (3.2) is an unquantified normal expression:

$$\{[x.a = y.b \& y.b > z.c] \text{ or } \text{NOT } [T(x,y) \& U(w)]\} \quad (3.2)$$

Logical quantifiers in our relational language will be denoted as Q_1, Q_2, etc. We use this term in a context somewhat more general than the traditional set-theoretic context. The quantifier Q(x) will count the number of x in Rx that satisfies a specified condition and then apply a specified function which returns a boolean value. For instance, the quantifier $\exists$ x returns a value TRUE iff COUNT(x) $\geq$ 1, and the quantifier $\forall$ x will return TRUE when COUNT(x) equals the cardinality of Rx. A more novel quantifier might return TRUE when 5 < COUNT(x) < 10. Any function which requires O(1) time to return a boolean value as a function of COUNT(x) will be considered a quantifier in the language RCS.

The final part of our calculus language will be the retrieve-clause: This will be an expression of one of the forms:

(i) FIND($x_1, x_2 \ldots x_i$)
(ii) COUNT(x_1, x_2)
(iii) SUM(x_1, x_2).
(iv) TEST.

If $r(x_1 x_2 \ldots x_i)$ is a retrieve clause, $j \geq i$, and $e(x_1, x_2 \ldots x_j)$ is a normal unquantified relational expression then a sentence of the form below is called a quantified calculus expression

$$\{r(x_1 x_2 \ldots x_i)(Q_{i+1}(x_{i+1}) \ldots Q_j(x_j): e(x_1 x_2 \ldots x_j))\}$$

The meaning of such an expression depends on the type of its retrieve clause, and it is defined below:

(i) If r is a clause of the type "FIND" then return all the i-tuples which satisfy the expression to the right of the retrieve clause and belong to the cross product $Rx_1 \times Rx_2 \times \ldots Rx_i$

(ii) If r is a clause of the type COUNT (x_1, x_2) then the calculus expression is an order to construct an array I( ) defined as follows: For each $x_1 \in Rx_1$, $I(x_1)$ is the number of different elements x_2 and Rx_2 satisfying the expression to the right of the retrieve clause.

(iii) SUM clauses are a straightforward generalization of COUNT to any commutative semigroup. (Note: SUM and COUNT clauses always have two arguments.)

(iv) If r=TEST then return the boolean value derived from evaluating the expression to the right of this clause.

One additional comment on notation: Since the variables x, y, and z are always assumed to belong to the respective relations Rx, Ry and Rz, we do not formally write them into our relational calculus expressions. That is, our shorthand notation will have (3.4) stand as an abbreviation for (3.5)

$$\{\text{FIND}(x)\ \forall y\ \exists z\text{: } e(x,y,z)\} \quad (3.4)$$

$$\{\text{FIND}(x \text{ in } Rx)\ \forall y \in Ry\ \exists z \in Rz\text{: } e(x,y,z)\} \quad (3.5)$$

Throughout this paper, q will denote a normal relational calculus expression. The graph of this expression, denoted G(q), will be defined to have vertex set corresponding to the eset variables appearing in q and a directed edge from x to y iff these two vertices appear together in some atomic predicate and the retrieve or quantifier clause defining x lies left of y's definition in q. For instance, the graph of equation (3.6) is a tree.

$$\{\text{FIND}(x,y)\ \forall z\ \exists w\text{: } x.a > w.b \ \&\ T(x,y) \ \&\ y.b = z.c\} \quad (3.6)$$

RCS is defined as the subset of the normal relational calculus whose graph is a tree or forest with all paths leading towards the leaves. Equation (2.6) is therefore an example of a member of RCS. We stress there are no other restrictions on the language RCS: its atomic predicates can have any one of five forms, and there can be an arbitrary number of these connected in arbitrary manner by the symbols of "AND", "OR" and "NOT". We will now provide some intuition for why it would be

interesting to develop an algorithm for performing all RCS queries in $O(N \log^d N)$ time and $O(N)$ space.

The query class would not be adequately broad if we excluded the notion of associative relations and therefore also excluded tabular predicates. For instance, if Rx, Ry and Rz are each relations of the type eset then equation (3.7) would not belong to RCS because its graph is not a tree.

$$\{FIND(x,y) \exists z: x.a > y.b \ \&\ x.prime \quad (3.7)$$

$$= z.a \ \&\ y.prime = z.b\}$$

However, if one thinks of Rz as an associate relation and if T(x,y) denotes the tabular predicate in equation (3.1) then (3.7) is clearly equivalent to the elements satisfying:

$$\{FIND(x,y): x.a > y.b \ \&\ T(x,y)\} \quad (3.8)$$

The first point is that the latter equation is an RCS expression and this formalization is what we need to describe those expressions which at first appear to be troublesome but can nevertheless be processed efficiently. The number of predicates added to our query language after introducing the notion of tabular predicates based on associative relational is considerable. Although we have no statistics, we guess that >95% of commercial queries can be converted into RCS expressions.

The second although lesser advantage of associative relations is that they make the data base language more accessible to a casual user. For instance, Equation (3.8) is much easier to read than (3.7). The point about the user-friendly nature of tabular primitives has appeared in the literature, on entity relationships, and it will not be discussed further here except to repeat that dividing relations into the categories of esets and associations allows us to broaden the class of predicates that can be processed efficiently. The gist of our data base research is that all RCS queries can be processed quasi-linear time and linear space with the exponent d in the time $O(N \log^d N)$ almost always ≤ 1 under the more elaborate variation of our method appearing in [Wi83b]. Treating $O(N \log^d N)$ as a measure of CPU time, this result will mean very complicated relational calculus queries can be executed in a small number of seconds under future hardware technologies where entire data base files are stored in main memory. Section 4 will discuss our algorithm's decomposability modules and section 5 decomposition.

4. Decomposability Analysis

The term E-8 predicate refers to an unquantified normal predicate that has precisely two eset variables. (This terminology comes from Willard's dissertation, and seven subsets of E-8 were also defined there.) There are precisely four kinds of relational calculus expressions that our language allows us to build using E-8 expressions and retrieve clauses containing at least one variable. These are listed below:

$$\{FIND(x)Q(y): e(x,y)\} \quad (4.1)$$

$$\{FIND(x,y): e(x,y)\} \quad (4.2)$$

$$\{COUNT(x,y): e(x,y)\} \quad (4.3)$$

$$\{SUM(x,y): e(x,y)\}. \quad (4.4)$$

The algorithms that perform the operations above will be called E-8 queries.

Let D(e) denote the number of distinct y-attributes in e(x,y)'s order predicate comparisons with x, and T(e) the number of distinct attributes in the combination of e's equality and order atoms. These quantities are usually small numbers; for instance $D(e_1) = 2$, $T(e_1) = 3$, $D(e_2) = 0$, $T(e_2) = 1$, $D(e_3) = 0$, and $T(e_3) = 1$ in the equations below:

$$e_1 = \{y.b_1 > x.a_1 \ \&\ y.b_2 \neq x.a_2 \ \&\ y.b_3 > x.a_3\} \quad (4.5)$$

$$e_2 = \{[y.b_1 = x.a_1 \ \&\ T.(x,y)] \text{ or } y.b_1 = x.a_2\} \quad (4.6)$$

$$e_3 = \{y.b_1 > c \ \&\ u(x) \ \&\ y.b_1 = x.a_1\} \quad (4.7)$$

One of the two main theorems in the unabridged version of our paper [Wi83b] states that all E-8 queries run in locate $O(N \log^{D(e)} N)$ time and $O(N)$ space, and that most of these queries actually meet the better time asymptote $O(N \log^{D(e)-1} N)$.

Unfortunately, [Wi83b] is quite long, and we will therefore prove the following weaker result which is essentially the same as [Wi83b] except that it replaces D(e) with the somewhat larger exponent T(e) in its runtime.

Theorem 4.1. If T(e)=0 then each of the four types of E-8 queries given in equations (4.1) thru (4.4) can be executed in locate time and memory O(N). If $T(e) \geq 1$ then the worst-case time but not memory should be increased to $O(MAX(N \log N, N \log^{T(e)-1} N))$.

Let the phrase y-based unary term refer to a predicate which consists of one or more y-based unary atoms concatenated by the connectives of "and", "or" and "not", and define an xy-tabular term to be a similar combination of xy-tabular predicates. If θ_1 and θ_2 are one of the symbols "<" and "$\leq$," the three forms "$x.a_1\ \theta_1\ y.b\ \theta_2\ x.a_2$", "$x.a\ \theta_1\ y.b$" or "$y.b\ \theta_1\ x.a$" are called xy-range terms. Our analysis of E-8 predicates is simplified if we examine the following increasingly broad subclasses of E-8 defined below:

B-1: A predicate which is either the boolean constant "TRUE" or a conjunction of several xy-range terms. Equation (4.8) offers a simple example of a B-1 predicate called the standard orthogonal range query:

$$x.a_1 < y.b_1 < x.c_1 \wedge x.a_2 < y.b_2 < x.c_2 \wedge \ldots x.a_k < y.b_k < x.c_k \qquad (4.8)$$

B-2: A B-1 predicate or a conjunction of a B-1 predicate with a y-based unary term.

B-3: A B-2 predicate or a conjunction of a B-2 predicate with an x-based unary term.

B-4: A B-3 predicate or a conjunction of a B-3 with an xy-tabular term.

Our first goal will be to prove that the subclasses B-1, B-2, B-3, and B-4 of E-8 predicates satisfy the claim of Theorem 4.1. This analysis will actually be quite simple since the time bound $O(N \log^{T(e)-1} N)$ in this paper is much easier than the tighter bound $O(N \log^{D(e)-1} N)$ appearing in our longer unabridged paper [Wi83b]. (The latter is necessary for a practical implementation of our algorithm, but the former is sufficient to explain its intuition.)

The formal statement of the Edelsbrenner-Overmars theorem indicates that the retrieval operations COUNT(x,y), FIND(x,y) and SUM(x,y) are doable in time $O(N \log^{T(e)-1})$ and space O(N) for any B-1 predicate e satisfying $T(e) \geq 2$. A brief summary of the Edelsbrenner-Overmars algorithm is given in the Appendix, but the reader can understand the rest of this article without examining it if he simply accepts that equation (4.8) can be processed in time $O(N \log^{T(e)-1} N)$ and space O(N) when $T(e) \geq 2$. A good amount of credit for the Edelsbrenner-Overmars theorem should be given to Bentley-Shamos [BS77] who used a special case of this technique for the case of calculating ECDF statistics. [EO83] can be thought of as a generalization of [BS77]'s memory saving technique to the data structures of Willard [Wi78b] where a new factor log N savings in time appeared.

Lemma 4.2. The retrieval clause COUNT(X,Y) runs in locate time $O(N \log^{T(e)-1} N)$ and space O(N) for the cases of B-1, B-2, B-3, and B-4 predicates of degree $T(e) \geq 2$, and its locate time $= O(N \log^{T(e)} N)$ when $T(e) \leq 1$.

Our proof of Lemma 4.1 will be divided into 3 parts which verify it for the increasingly more general cases of B-2, B-3 and B-4. The proof will assume that the class B-1 satisfies Lemma 4.1 since this fact is trivial to verify when $T(e) \leq 1$, and it is otherwise a consequence of the Edelsbrenner-Overmars theorem.

Verification for B-2 Predicates: By definition, every B-2 predicate can be written in the form $\{e^*(x,y)\ \&\ U(y)\}$ where e^* is B-1 and U(y) a y-based unary term. The algorithm for B-2 will consist of the following two steps:

1) Scan Ry and construct the subset of this relation, denoted R_y^*, which satisfies U(y).
2) Apply the B-1 COUNT algorithm to evaluate the subset of R_y^* satisfying $e^*(x,y)$. The resulting array I(x) is also the answer to the B-2 query.

The algorithm above clearly satisfies Lemma 4.1's time and space claims, since step 1 increases the costs beyond the B-1 algorithms by an inconsequential additive amount O(N).

Q.E.D.

Verification for B-3 Predicates: Again easy. By definition, every B-3 predicate can be written in the form {e*(x,y) & U(x)} where e*(x,y) is B-2 and U(x) an x-based unary predicate. Therefore we calculate $I_e(x)$ with the following three-step procedure:

1) Scan Rx and calculate the boolean value U(x) for each of its members.
2) Apply the B-2 algorithms to produce the array $I_{e*}(x)$.
3) Treating U(x) as an array of integers 0 or 1, apply the formula (4.9) to calculate $I_e(x)$.

$$I_e(x) = U(x) \cdot I_{e*}(x) \qquad (4.9)$$

Since the B-3 algorithm uses only O(N) more resources than the B-2 procedure, it is once again obvious Lemma 4.2 is satisfied.
Q.E.D.

Verification for B-4 Predicates: Let $T_1T_2\ldots T_j$ denote the xy-tabular predicates in our data base, and $R_1R_2\ldots R_j$ the subset of $R_x \times R_y$ that belong to the tables or relations defining these tabular predicates. Let H denote a hash table whose record-set is the union $R_1 \sqcup R_2 \sqcup \ldots R_j$ and which in some fashion indicates for each record which tables it belongs to and what are the attributes of its x- and y-components. Assume the B-4 predicate e(x,y) is written in the form e*(x,y) & u(x,y) where e* is a B-3 predicate and u(x,y) is a tabular term. Let $I_e(x)$ and $I_{e*}(x)$ denote the count arrays for these predicates, and define I(x) to equal $I_e(x)-I_{e*}(x)$. Our algorithm for constructing the array $I_e(x)$ for a tabular predicate in quasilinear time and linear space consists of the following three-step procedure:

1) Call the B-3 algorithm to construct the array $I_{e*}(x)$;
2) In O(N) time, construct H and then scan it to build the array I(x);
3) Calculate the desired array $I_e(x)$ in O(N) time by performing the array addition operation $I_e(x) \leftarrow I_{e*}(x) + I(x)$.

The correctness of the procedure above is transparent. It is also clear that steps 1 and 3 are sufficiently inexpensive to meet Lemma 3.2's time and space claims. Step 2 is slightly more subtle and the next paragraph explains how to execute it in time O(N).

Assume the integers of 0 and 1 represent the boolean constants of TRUE and FALSE. For any ordered pair (x,y), define $\Delta(x,y)$ to be the difference e(x,y) - e*(x,y). Then this quantity will correspond to an integer between -1 and 1. We will construct the array I(x) by initially setting it to zero, walking down the table H, and setting $I(x) \leftarrow I(x) + \Delta(x,y)$ for each record (x,y) stored in this table. The two central points are that each I(x) increment operation runs in O(1) time because of the information stored in the table H and that there can be no more than O(N) such operations performed because H's size is $\leq$ the sum of the cardinalities of $|R_1|$ thru $|R_j|$ which $\leq N$ because the latter quantity is defined as the sum of the sizes of all the associative and eset relations in the data base. Therefore, step 2 runs in time O(N) completing the last step of our lemma proof. Some readers may now have reservations about how Section 1 defined N, and we address those questions in the next paragraph.
Q.E.D.

Comment 1: At fist, it may appear that the last step of our proof used an unrealistic cost model because if Rx and Ry had cardinalities of say $\equiv O(M)$, then each tabular set R_i could have size $O(M^2)$. This reasoning could lead some readers to suspect that while our result is correct, N is still much larger in our article than in common computer science notation. There are two good reasons for believing this problem is less important than it appears and should not arise in practical situations:

1) The notion of a tabular predicate was introduced by us to represent a concept comparable to associations in the entity-relationship model or to the ownership relations in the network model. In CODASYL, these tables are guaranteed to have size $\leq M$ when $|Rx|=|Ry|=M$ because of the constraints imposed by the many-to-one and one-to-many conditions. Even if the condition is many-to-few or few-to-many, the associations generating tabular predicates will have sizes $\leq O(\text{Max}(|Rx|,|Ry|))$. Our estimate that N is more nearly approximated by the small asymptote O(M) rather than

the huge $O(M^2)$ appears to be also corroborated by the frequent presence of sparse sets in database applications involving CODASYL-like associations and by the observation that non-sparse subsets of the cross-product Rx X Ry are usually represented as implicit set definitions (involving equality, order and unary atoms).

2) The typical value of N also corroborates the usefulness of Lemma 3.1's algorithm. Whenever the number of tuples relevant to the query is $\leq 10^6$ or 10^7, the invocation of O(N log N) CPU operations is cheap and will become increasingly attractive as the falling cost of chips makes possible storing whole files in main memory.

Now we will give our proof of Theorem 3.1 with the caveat that since the exponent in $O(N \log^d N)$ is $\cong$ T(e)-1 rather than D(e)-1, this treatment does not represent our recommended algorithm. Rather it appears in [Wi83b]. This short conference paper seeks to persuade the reader to examine [Wi83b] as well as the background literature cited in the Appendix, and it presents the material in the deliberately most trivial form to entice the reader. Our proof of Theorem 4.1 and all the other discussion in this section and the next section should be viewed in that light. The proof will be divided into 4 cases which separately examines the 4 types of E-8 queries defined in equations (4.1) thru (4.4).

Proof that Theorem 4.1 holds for the case of the query {COUNT(x,y): e(x,y)}: For each E-8 predicate of degree T(e), it is easy to prove there exists a sequence of B-4 predicates $e_1 e_2 \ldots e_k$ of degree $\leq$ T(e), such that the following equality necessarily holds:

$$\{\text{COUNT}(x,y)\text{: } e(x,y)\} = \sum_{j=1}^{K} \{\text{COUNT: } e_j(x,y)\}$$

The equation above shows we can evaluate the left side by making K separate calls to B-4 algorithms that run in quasilinear time. Whenever the time increases by a factor unrelated to the database-relation size N, it is considered a coefficient change rather than an asymptotic one. Hence {COUNT(x,y): e(x,y)} has the same asymptote as {COUNT(x,y): $e_j(x,y)$}, but its coefficient is larger by a factor of approximately K. An example of this transformation would be if e is the predicate (4.10) and its count is the sum of the counts for (4.11) and (4.12)

$$e(x,y) = \{x.a_1 < y.b_1 \text{ or } x.a_2 < y.b_2\} \quad (4.10)$$

$$e_1(x,y) = \{x.a_1 < y.b_1\} \quad (4.11)$$

$$e_2(x,y) = \{x.a_1 \geq y.b_1 \text{ and } x.a_2 < y.b_2\} \quad (4.12)$$

Q.E.D.

Proof of Theorem 4.1 for {SUM(x,y): e(x,y)}: Same as for COUNT(x,y): e(x,y) since Lemma 4.2 and the paragraph above easily generalize.
Q.E.D.

Proof of Theorem 4.1 for {FIND(x)Q(y): e(x,y)} where Q is either the existential or universal quantifier: The first step of the algorithm would use the previous paragraphs to calculate $I_e(x)$ = {COUNT(x,y): e(x,y)}. The second step would find the subset of Rx that satisfies an existential or universal quantifier by scanning the array $I_e(x)$ and making a list of the subset satisfying respectively $I_e(x) \geq 1$ and $I_e(x) = |R_x|$.
Q.E.D.

Proof of Theorem 4.1 for retrievals of the form {FIND(x,y): e(xy)}: First, the reader is reminded that Theorem 4.1 is invalid when the complexity model is strict worst-case time rather than locate time because a FIND retrieval that prints N^2 element requires $O(N^2)$ time in the traditional cost models. However, the 2-variable FIND operations are efficient in the locate model, and the algorithm for executing them is mostly analogous to COUNT. There are some minor distinctions especially with regards to Lemma 4.2, but these fairly straightforward adjustments are omitted for the sake of brevity.
Q.E.D.

The next section will illustrate the significance of Theorem 4.1 by showing how it becomes part of a broader algorithm system. Examples are known in more distant branches of orthogonal range theory where the cost of FIND and COUNT are different, and where rigorously proven upper [Wi78b] and lower [Fr81] bounds show they will never match! Indeed, the stronger version of our theorem in [Wi83b] has

slightly different complexities of $\equiv O(N \log^{D(e)-1}N)$ and $O(N \log^{D(e)}N)$ for some COUNT and FIND operations. The main result in our dissertation [Wi78a] was a yet stronger variation of Theorem 4.1 where retrievals are defined slightly differently and where dynamic data structures are shown to exist with $O(\log^{D(e)}N)$ or $O(\log^{D(e)-1}N)$ complexity for insert, delete and simultaneously FIND, SUM and COUNT. We stress that the techniques presented in this section may be useful for classroom presentation because of their simplicity, but they are less eloquent and less practical than our unabridged paper which changes the exponent to $\equiv D(e)-1$. (The most difficult aspect of [Wi83b] is FIND rather than the SUM and COUNT commands. The exponent can be lowered to D(e)-1 for the latter by using concatenated keys in a straightforward manner.)

5. Decomposition Theory

Our full-length 80-page paper [Wi83b] shows how to use Theorem 4.1 to prove that the entire language RCS can be processed in quasilinear time and linear space. The algorithm can be thought as an integration of the last section's decomposability theory with acyclic decomposition theory [BC81,BFMMUY81,BFM83,BG81a,BG81b,Da83,Fa83,GS82, Hu83, JK83,Li83,WY83,YO79,Ya81,Za76], but it was not originally conceived in this manner. An early version of our result having the same time but less efficient memory space appeared in Theorem 7.5L of [Wi78a] before nearly all of the latter literature appeared. Since decomposition was the main topic of our other conference paper on predicate retrieval [Wi83a], the discussion here will be brief and consist of two examples summarizing [Wi83a]'s presentation.

Example 5.1: Let e_1, e_2, e_3 and e_4 denote four E-8 expressions and consider the RCS query

$$\{FIND(x,y) \exists w \exists z \forall v: e_1(x,y) \& e_2(x,z) \& e_3(x,w) \& e_4(x,v)\} \quad (5.1)$$

Let $L_2 L_3$ and L_4 denote the subsets of Rx and Ry satisfying the E-8 queries

$$L_2 = \{FIND(x) \exists z: e_2(x,z)\} \quad (5.2)$$

$$L_3 = \{FIND(x) \exists w: e_3(x,w)\} \quad (5.3)$$

$$L_4 = \{FIND(x) \forall v: e_4(x,v)\}. \quad (5.4)$$

Then if $u_i(x)$ denotes the unary-list predicates associated with the list L_i, equation (5.5) is an example of an "E-8 reduction" whose solution set is equivalent to the original query in (5.1).

$$\{FIND(x,y): e_1(x,y) \& u_2(x) \& u_3(x) \& u_4(x)\} \quad (5.5)$$

The point of this example is thus that the execution of the E-8 queries in equation (5.2) thru (5.5) constructs the set of elements satisfying (5.1). A consequence of the last chapter is that this construction meets the efficiency constraints of quasilinear locate time and linear space.

Example 5.2: Sometimes, the result of a RCS query is best produced if one executes several E-8 queries and then uses the traditional natural join algorithm for acyclic queries to combine the results. For instance, suppose we wish to find the set of records satisfying

$$\{FIND(w,x,y,z) \forall v: e_1(x,y) \& e_2(x,z) \& e_3(x,w) \& e_4(x,v)\}. \quad (5.6)$$

One algorithm for solving (5.5) would consist of the following 2-step procedure.

A) Use the procedure from the previous example to efficiently find the set of records satisfying equation (5.1) as before, as well as the sets satisfying the following symmetrical permutations of that example:

$$\{FIND(x,z) \exists w \exists y \forall v: e_1(x,y) \& e_2(x,z) \& e_3(x,w) \& e_4(x,v)\} \quad (5.7)$$

$$\{FIND(w,x) \exists y \exists z \forall v: e_1(x,y) \& e_2(x,z) \& e_3(x,w) \& e_4(x,w)\} \quad (5.8)$$

B) Calculate the set of records satisfying equation (5.6) by taking the natural joins of the sets defined in Step A.

The curious aspect of the procedure above is that it can be proven to be efficient as well as correct. That is, its memory space is linear and its time is quasilinear in the locate cost model.

The algorithm in the example above was simple, but is was not constructed casually. There are other simple algorithms combining E-8 theory with acyclic natural joins that are logically equivalent to the process above but inefficient. One example of a poor synthesis of acyclic joins with E-8 theory is the following alternate 2-step procedure:

A) Use the E-8 algorithm indicated in equation (5.4) to produce the unary predicate $u_4(x)$ as in the previous Example 5.1.

B) Use the E-8 algorithms from the previous chapter to find the set of records satisfying (5.9) thru (5.11), and then calculate the set satisfying (5.6) by taking their natural join with the traditional acyclic joining algorithm

$$\{FIND(x,y): e_1(x,y) \ \& \ u_4(x)\} \qquad (5.9)$$

$$\{FIND(x,z): e_2(x,z) \ \& \ u_4(x)\} \qquad (5.10)$$

$$\{FIND(x,w): e_3(x,w) \ \& \ u_4(x)\} \qquad (5.11)$$

The surprising fact is that the algorithm above is inefficient although it appears to be deceptively similar to the previous paragraph. The intermediate sets produced by operations (5.9) thru (5.11) could have $O(N^2)$ size even when the final output is far smaller. Thus, the second algorithm cannot have quasilinear time in the locate model while the presence of the existential quantifiers in equation (5.1), (5.7), and (5.8) can be used to prove this problem never arises for the first algorithm. Thus a certain amount of circumspection is needed to combine E-8 theory with join theory in the proof of the following proposition:

Theorem 5.3: All RCS queries support quasilinear time and linear space in the locate cost model.

The proof of Theorem 5.3 can be thought of as a very elaborate synthesis of acyclic join and decomposable data structure theoreies, but this is not how it was originally conceived. A counterpart of Theorem 5.3 using inefficient memory space but good time appeared in Proposition 7.5L of Willard's dissertation [Wi78a] somewhat before acyclicty and its implications for natural joins became topics in the database literature. The decomposition half of our algorithm for processing RCS appears in our other conference paper [Wi83a], and it is therefore unnecessary to give more details here. We also recommend you examine [Wi83b] and [Wi84a]. The former article explains how to lower significantly the exponent d in the time $O(N \log^d N)$, and the latter describes a useful rule about statistical sampling involving the primacy of samples of the size $N^{2/3}$ which is imperative to understand how the coefficient inside the O-notation should be controlled for both RCS and many other data base problems. Not until you read these articles will you fully understand our data base formalism. Some readers may also wish to read the surey of decomposability literature in the Appendix and the various articles it cites. We stress that the article you just finished reading has omitted the most subtle aspects of our proposal that give it added mathematical eloquence and practical significance because we felt our first paper at an international data base conference should be written in a style suitable for classroom presentation.

Since the subject defined in this paper is neither classical relational nor range query theory, it may need a new name. Call the synthesis of these two subjects Predicate Retrieval Theory.

Acknowledgements

I warmly thank my thesis advisor, Prof. Ugo Gagliardi for making the extremely timely suggestion in 1975 that I study relational calculus optimization. I thank several of my computer science friends: Eugene Kramer, Sam Kashdan, Larry Kerschberg, Robert Paige and his wife Nieba, Shlomo (Dick) Tsur, and Carlo Zaniolo for urging me to persist. I thank Jon Bentley for passing me some useful literature and my Bell Labs management: B. Gopinath, Ed Lien, and Eric Wolman for their patient support over the last five years.

Appendix

This Appendix offers a brief summary of the Edelsbrenner-Overmars algorithm and the background literature on decomposable data structures. Our discussion will be very brief because it would be

inappropriate to duplicate the details of [EO83]'s algorithm. Actually, it is unnecessary for the reader to examine this section if he accepts the claim in [EO83] that a batch of N d-dimensional orthogonal range queries can be executed in $O(N \log^{d-1} N)$ time and $O(N)$ space for $d \geq 2$ on a database of $\leq N$ tuples. The main purpose of this Appendix is to summarize the background literature to [EO83]'s treatment so that the reader can understand where he can find more material.

The term augmented tree refers to a data structure first introduced in [BS77,Be80] and later discussed in [Ed81,LW80,EO83,Wi78a,Wi78b,WL84]. This data structure has two parts. Its first section is called the base and consists of a binary tree where the records are stored in some specified order. Henceforth, we will call an element of Ry a y-record. A possible base would be a tree that stores the y-records in order of increasing attribute value $y.b_2$. The second part of an augmented tree is its set of auxillary fields. For each internal node v of the base, AUX(v) will consist of some data structure describing v's descendants. An example would be AUX(v) fields that enumerate the y-records descending from v in order of increasing $y.b_1$ value.

The data structure above is called a first-generation 2-fold tree, or more briefly, a 2-fold tree. One can apply this definition recursively: A d-fold tree has a base sorted by $y.b_d$ and has (d-1)-fold trees as auxillary fields. The first basic result from [BS77,Be80] was that these trees permit one to perform orthogonal range queries in time $O(\log^d N)$; that is $O(\log^d N)$ is the time needed by COUNT and FIND to collect the y-records satisfying:

$$A_1 \leq y.b_1 \leq B_1 \;\&\; \ldots \;\&\; A_d \leq y.b_d \leq B_d \quad (6.5)$$

Two other observations in [BS77] were that $N \log^{d-1} N$ represented both the memory cost and time to build the data structure, and that batch ECDF statistics calculate in time $O(N \log^{d-1} N)$ and space $O(N)$ under a slightly modified data structure.

Five major discoveries were made largely independently about augmented trees in the year following [BS77]'s conference presentation. Willard [Wi78b] showed that the retrieve time could be reduced to $O(\log^{d-1} N)$ in all static-on-line and some dynamic applications with a data structure called the second-generation d-fold tree. (Several years later, [Ed81,CH83] developed alternate methods that had a complexity $O(\log^{d-1} N)$ for FIND but not COUNT.) A general technique for performing insertions and deletions in this data structure was developed independently by Lueker and Willard and will soon appear in [WL84]. Fredman discovered a lower bound [Fr81] matching the latter technique's upper bound. Chapters 4 through 6 of Willard's dissertation [Wi78a] introduced some special dynamic variants of augmented trees, which in 1978 appeared to have only theoretical interest. Bentley and Saxe [BS81] introduced the word "decomposability" into the vocabulary of data structure theorists, and illustrated the notion of augmented trees in its greatest generality.

During the next few years much progress has been made in other areas of multi-dimensional and decomposability theory [BM80,CY83,EW83,OL81,Wi78c,Wi82,Ya82,Ya83], and Edelsbrenner and Overmars announced in [EO83] that the memory space of essentially all augmented trees could be reduced to O(N) at no loss in time for the special case where N queries are performed in batch! The essence of their proposal is that only one of the augmented tree's fields should be constructed at a particular moment in time, and all the queries intending to search that field should do so precisely when it is constructed. This method, called memory streaming, was used previously in the Bentley-Shamos ECDF batch algorithm, but Edelsbrenner and Overmars were the first to propose it as a general algorithmic technique.

Memory streaming changes the significance of Chapters 4 through 6 of Willard's dissertation radically. It reduces the memory use of Willard's batch data structures from quasilinear to linear in N (an important distinction since we are discussing main memory). The second-generation d-fold tree [Wi78b] reduces almost all [Wi78a]'s time complexities by a factor log N. The resulting complexities were presented in Section 3, and are proven in Sections 4 and 5.

References
(with articles' formal titles omitted to save journal space)

[Ab74]Abrial, in Data Base Management, North Holland Publishing Co., 1974, 1-60.

[ABU79]Aho, Berri & Ullman, ACM TODS, (4) 1979, 297-314.

[AHU74]Aho, Hopcroft & Ullman, The Design and Analysis of Computer Algorithms, 1974.

[BC81]Bernstein & Chiu, JACM, 28, 1981, 25-40.

[Be75]Bentley, CACM, 18 (1975), 509-517.

[Be80]Bentley, CACM, 23 (1980), 214-228.

[BFMMUY81]Berri, et al., 13th ACM STOC (1981), 355-362.

[BFMY83]Berri, et al., JACM, 30, 1983, 479-514.

[BG81a]Bernstein & Goodman, SIAM J. Comput., 10 (1981), 751-771.

[BG81b]Bernstein & Goodman, Inf. Syst., (6) 1981, 255-264.

[BM80]Bentley & Maurer, Acta Inf., 13 (1980), 155-168.

[BS77]Bentley & Shamos, 15-th Allerton Conf. on Comm. Contr. and Comp., (1977), 193-201.

[BS80]Bentley & Saxe, J. Algorithms, 1 (1980), 301-358.

[Ch76]Chen, TODS, (1), 1976, 9-36.

[CH80]Chiu & Ho., 1980 SIGMOD Conf., 160-168.

[CH83]Chazelle, 24-th IEEE FOCS, (1983), 122-132

[Co70]Codd, CACM, 13 (1970), 377-397.

[CY83]Cole & Yap, 24-th IEEE FOCS, (1983), 112-121.

[Da83]Dayal, 2-nd ACM PODS, 1983, 125-137.

[Ed81]Edelsbrenner, Bulletin of EATCS, (15) 1981, 34-40.

[EO83]Edelsbrenner & Overmars, "Batch Solutions to Decomposable Search Problems," Univ. Utrecht RUU-CS-83-8, 1983.

[EW83]Edelsbrenner & Welzl, "Halfplanar Range Search in Linear Space and $O(n^{.695})$ Query Time," Univ. of Graz Report F-111, 1983.

[Fa83]Fagin, JACM, 30 (1983), 514-551.

[FKS82]Fredman, Komlos & Szemeredi, 23-rd IEEE FOCS, 1982, 165-170.

[Fr81]Fredman, JACM, 28 (1981), 696-706.

[GP84]Goldberg & Paige, "Strenn Processing," Rutgers TM, 1984.

[GS82]Goodman & Shmueli, TODS, 4 (1982), 653-678.

[HK80]Hecht & Kerschberg, "Update Semantics for the functional data base model," Bell Labs technical memorandum 1980. Write Kershberg at Univ. of South Carolina to get a copy.

[HS81a]Horowitz & Sahni, Fundamentals of Data Structures, 1978.

[HS81b]Horowitz & Sahni, Fundamentals of Computer Algorithms, 1978.

[Hu83]Hull, JCSS, 1983.

[JK83]Jarke & Koch, ACM SIGMOD Symp., (1983), 196-206.

[Kn73]Knuth, The Art of Computer Programming, vol. 3, 1973.

[KP81]Koenig & Paige, 7-th VLDB, 1981, 306-318.

[KTY82]Kerchberg, et al., ACM TODS, (4) 1982, 653-677.

[Li82]Lien, JACM, 29 (1982), 333-362.

[LST81]Lien, Shapiro & Tsur, 7-th VLDB, 1981, 306-318.

[LW77]Lee & Wong, Acta Inf., 9 (1977), 23-29.

[LW80]Lee & Wong, ACM TODS, 5 (1980), 339-347.

[MSY82]Maier, Sagiv & Yannakakis, J. ACM, 28 (1982), 680-695.

[MU82]Maier & Ullman, 1-st PODS, 1982, 34-39.

[OL82]Overmars & Leeuvwen, Acta Inf. 17(1982)267-286.

[Pa79]Paige, Ph.D. dissertation, NYU, 1979.

[Pa81]Papademitriyou, 1981 ACM SIGMOD Conference.

[Pa84]Paige, to appear in Advances in Database Theory.

[PK82]Paige & Koenig, ACM TOPLAS, 1982, 402-454.

[RND77]Reingold, Nievergelt & Deo, Combinational Algorithms: Theory and Practice, 1977.

[RSL78]Rosenkrantz, Stearns & Lewis, ACM TODS, (3) 1978, 178-198.

[Sh81]Shipman, ACM TODS, 6 (1981), 140-173.

[SK77]Sibley & Kerschberg, Proc NCC, AFIPS, 1977, 85-96.

[SLR76]Stearns, Lewis & Rosenkrantz, 17-th IEEE FOCS, 1976, 19-32.

[Ul82]Ullman, Principles of Database Systems, 1982.

[Wi78a]Willard, Ph.D. thesis, Harvard University, 1978. Published in Outstanding Dissertations in Computer Science, Garland Publishing, New York, 1979. The Garland copies are priced as hard-cover books; you can save money by asking your librarian to purchase a copy.

[Wi78b]Willard, "New Data Structure for Orthogonal Queries," first draft was Harvard TR-22-78 (1978), second draft appeared in Proceedings 1982 Allerton Conference, third draft to appear in SIAM J. Comp.

[Wi78c]Willard, "Balanced Forests of K-d* Trees as a Dynamic Data Structure," Technical Report TR-23-78 (1978), Center for Research in Computing Technology, Harvard University, 1978. Revised and reissued as "K-d Trees in a Dynamic Environment," Technical Report No. 80-1, Department of Computer Science, The University of Iowa, 1980. To appear in CACM.

[Wi82]Willard, SIAM J. Comp., (11) 1982, 149-162.

[Wi83a]Willard, 21-st Allerton Conf. on Comm., Contr., and Comp., 1983, 663-675.

[Wi83b]Willard, SUNY Albany Technical Report 83-3, 1983.

[Wi83c]Willard, 21-st Allerton Conf., 1983, 656-662.

[Wi83d]Willard, Inf. Proc. Let., 24 (1983), 81-84.

[Wi84a]Willard, at 1984 Int. Col. on Automata, Languages and Programming.

[Wi84b]Willard, at 1984 ACM STOC Conference.

[Wi84c]Willard, to appear in JCSS, June 1984.

[WL84]Willard & Leuker, to appear in J. of ACM.

[WY76]Wong & Youssefi, ACM TODS, (1976), 233-241.

[Ya81]Yannakakis, 6-th VLDB, (1981), 82-94.

[Ya83]Yao, 15-th ACM STOC, (1983), 258-264.

[YO79]Yu & Ozsoyoglu, COMPSAC, 1979, 306-312.

[Za76]Zaniolo, Ph.D. dissertation, UCLA, 1976.

[Ya82]Yao, 14th ACM STOC (1982), 128-136.