Journal of Computer and System Sciences **65,** 295–331 (2002)
doi:10.1006/jcss.2002.1848

An Algorithm for Handling Many Relational Calculus Queries Efficiently[1]

Dan E. Willard

Department of Computer Science, University at Albany - SUNY, Albany, New York 12222
E-mail: dew@cs.albany.edu

Received November 30, 1999; revised February 27, 2001

This article classifies a group of complicated relational calculus queries whose search algorithms run in time $O(I \text{ Log}^d I + U)$ and space $O(I)$, where I and U are the sizes of the input and output, and d is a constant depending on the query (which is usually, but not always, equal to zero or one). Our algorithm will not entail any preprocessing of the data. © 2002 Elsevier Science (USA)

1. INTRODUCTION

During the last 20 years the cost of computer memory has dropped by a factor of 10,000. This change seems to suggest that certain algorithms from Computational Geometry about multi-dimensional retrieval may possibly carry different implications today for database design than they did in the 1970s and 1980s.

This distinction arises because most of Computational Geometry's range query algorithms [3–6, 12–15, 18, 19, 21, 22, 34, 35, 37–39, 42–44, 56, 59, 61–64, 68, 69, 71] used a main memory model of the computer, where they sought to optimize *only CPU time* and completely ignored disk-access costs. In the past, these algorithms would not have been very meaningful in a database setting, where performance depended mostly on the costs of disk accesses. However, in a context where computer memory sizes have now grown by a factor of 10,000 during the last 20 years, the picture seems to have changed. We will show how these geometric algorithms naturally interface with the database literature about acyclic queries [1, 2, 8, 9, 25, 26, 49, 52, 55, 70, 72].

Our previous JCSS paper [67] also addressed this topic. It displayed an efficient algorithm for doing relational algebra selection and join operations, where the joins were required to have *only two* relations as input, and they computed the subset of

[1] Research partially supported by NSF under Grant CCR 99-02726.

the cross-product set that satisfied an arbitrary join-selection condition. Our goal in the present article is to explore how far one can generalize these results to substantially more complicated relational calculus queries that have k relations as input for arbitrary k.

It will turn out that we cannot handle efficiently all relational calculus queries with k variables. The recent article by Papadimitriou and Yannakakis [49] demonstrates that if the common conjecture of NP-completeness theory are correct, then the most general forms of relational calculus queries remain intractable. On the other hand, Papadimitriou and Yannakakis did note that acyclic relational database queries are different, and they have a potential to be efficient. We share this perspective and hope our article will reinforce it.

This article will suggest a method for processing acyclic relational calculus queries, where at a crucial juncture, the mainline algorithm (defined in Section 5) will make a series of subroutine-calls to one of the E-8 algorithms from our prior JCSS article [67]. It will essentially ask these subroutines to do the hard part of the work. Because Sections 2.1 and 3 of the present article will summarize in sufficient detail our previous work [67], the reader can appreciate the further ideas now presented here without having examined our earlier paper.

Throughout our discussion, q will denote a relational calculus query whose input has a cardinality of I and produces an output of cardinality U. Let us say that q has *quasi-linear complexity with exponent* d if the query can be performed in $O(I \operatorname{Log}^d I + U)$ worst-case hashing time and using $O(I + U)$ worst-case space. (Worst-case hashing time is defined as a measure of cost that is worst-case in every respect except that it assumes average luck when hashing. It will be henceforth denoted as *WH-Time*.) All our algorithms will have a quasi-linear performance complexity. Their methodology will essentially be a hybridization of acyclic relational database theory [2, 8, 9, 25, 26, 49, 52, 55, 70, 72] with the search methods from range query theory [3–6, 12–15, 18, 19, 21, 22, 34, 35, 37–39, 42–44, 56, 59, 61–64, 66, 68, 69, 71], as Sections 2 and 3 shall explain.

Vardi [57, 58] has defined two different methods for measuring search costs, called *Data Complexity* and *Combined Complexity*. The former views a query as a fixed constant and measures its runtime as a function of its input size I and its output size U (as we do). The latter does not view the query as a fixed constant. It measures runtime in terms of I, U and the query's length, denoted as L. Papadimitriou and Yannakakis [49] have noted that as approximations of reality both methods contain a degree of reasonableness. In particular, Data Complexity estimations of costs are reasonable when either the query is moderately small or its runtime parameter does not run fully out of control as L grows.

Our "quasi-linear" measurements of $O(I \operatorname{Log}^d I + U)$ WH-time and $O(I + U)$ space are obviously done in the "data-complexity" model of costs, since they contain no "L" terms. It turns out that both the values of d and the coefficient inside the O-notation are hidden quantities depending on L. In an extreme worst-case setting, our costs will certainly be problematic in the Combined-Complexity model of cost. Certainly, we wish neither to minimize this point nor encourage the reader to overlook it. What makes our algorithms firstly tempting, however, is that the *absolute worst-case* occurs only rarely in most realistic practical settings (i.e., most queries will have a sufficiently small value of L for its influence on the runtime coefficient to be adequately small.) Moreover, it will turn out that a *very broad class* of relational database queries will fit into our RCS language (formally defined in the next section). Our main theorems will imply that for this very broad language, every query operation is "quasi-linear" executable.

2. GENERAL FRAMEWORK

This section will introduce our main notation and apply it to state our principal theorems. It will also review the literature on acyclic databases and explain its relationship to these theorems.

2.1. Properties of E-8 Enactments

Throughout this paper, X and Y will denote two sets of tuples, and x and y will denote two tuple variables ranging over these respective sets. The lower case symbols, $\bar{x}$ and $\bar{y}$, will denote particular tuples in X and Y. Let $A_1(x), A_2(x), A_3(x)\dots$ denote attributes of the tuple x. Define an *equality* atom to be a predicate of the form $A_1(x) = A_2(y)$, and an *order* atom to be a predicate of the form $A_1(x) > A_2(y)$. Our final theorem will be stronger if it also includes two further types of atoms. In that regard, 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 *tabular atom* is defined as a predicate of the form $(x, y) \in T$.

Using our terminology from [67], a predicate will be called an *E-8 Enactment* iff it consists of equality, order, list and tabular atoms combined in arbitrary manner by AND, OR and NOT connectives. Two examples are given below:

$$e_1(x,y) = \{((x,y) \in T_1 \ \vee \ (x,y) \in T_2) \ \wedge \ A_1(x) > B_1(y) \ \wedge \ A_2(x) > B_2(y) \ \wedge \ x \in L\}, \quad (1)$$

$$e_2(x,y) = \{[A_1(x) < B_1(y) \ \wedge \ A_2(x) > B_1(y) \ \wedge \ A_3(x) = B_3(y)] \ \vee \ \neg A_4(x) = B_4(y)\}. \quad (2)$$

The symbol $d(e)$, called an *enactment degree*, will denote the number of *distinct* y-attributes in e's order atoms. Also, we will sometimes employ the symbol $d^*(e)$ defined below:

1. $d^*(e) = d(e) - 1$ when $d(e) \geqslant 2$.
2. $d^*(e) = d(e)$ when $d(e) \leqslant 1$.

For instance, Eqs. (1) and (2) will have $d(e_1) = 2$, $d^*(e_1) = 1$ and $d^*(e_2) = d(e_2) = 1$. Also, we will use the following notation:

1. N_x and N_y will denote the cardinalities of the sets X and Y.
2. N_t will denote the cardinality of the tabular sections employed by the enactment predicate e. For example in Eq. (1), N_t will denote the combined cardinality of the two tabular sections T_1 and T_2.
3. REPORT(e, X, Y) will denote the set of tuples $(\bar{x}, \bar{y})$ from the cross-product set $X \times Y$ satisfying $e(\bar{x}, \bar{y})$. Also, N_e will denote cardinality of REPORT(e, X, Y).

4. Let $f(\bullet)$ denote some function that maps elements of Y into a semigroup. Then $\Phi_e^f(\bar{x})$ will denote the $\sum f(\bar{y})$ over those elements $\bar{y} \in Y$ satisfying $e(\bar{x}, \bar{y})$. Also, $\Phi_e^f(\bullet)$ will denote an array that stores an aggregate quantity $\Phi_e^f(\bar{x})$ for each ($\bar{x} \in X$).

5. We will omit the exponent f from the notation "$\Phi_e^f(\bar{x})$" in the special degenerate case where $f(y) = 1$ for all y-elements. (In this case, the array Φ can be thought of as the result of a "COUNT" operation.)

To develop four procedures for processing enactment predicates, our prior article employed mostly some of Computational Geometry's range query theory algorithms. Two of its four algorithms, called the *Reporting and Aggregate Joins*, were generalizations mostly of the prior work of Bentley [4], Edelsbrunner and Overmars [19] and Willard [59, 61]. We will also use them often in the present article. So long as the reader understands the definitions and runtime characteristics of these procedures (listed immediately below), he will not need to be familiar further with their algorithmic details:

(A) The *Reporting Join* will be a procedure that will construct the set REPORT(e, X, Y) in no more than $O((N_x + N_y) \operatorname{Log}^{d^*(e)} N_y + N_t + N_e)$ WH-time and using no more than $O(N_x + N_y + N_t + N_e)$ memory space.

(B) Given as input a 4-tuple (e, f, X, Y), the *Aggregate Join*, will be a procedure that will construct the array $\Phi_e^f(\bullet)$ in a WH-time never exceeding $O((N_x + N_y) \operatorname{Log}^{d(e)} N_y + N_t)$ and using a memory space never exceeding $O(N_x + N_y + N_t)$.

The main distinction between the costs of procedures (A) and (B) is that N_e only influences A's costs. (This distinction is important because $N_e = N_x \cdot N_y$ in many applications.) Both these procedures will assume their input sets, X and Y, have had absolutely no preprocessing. Thus, any type of index, used to construct their final outputs, will be built in the midst of their computations.

Finally, we wish to close our summary of Willard's [67] Reporting and Aggregate Join procedures by noting that these algorithms are more efficient because their memory space sizes do not contain an "$(N_x + N_y) \operatorname{Log}^{d^*(e)} N_y$" quantity similar to their runtime magnitudes. The omission of this $(N_x + N_y) \operatorname{Log}^{d^*(e)} N_y$ quantity should be credited to the special memory-savings techniques developed by Bentley for doing aggregations with his ECDF algorithm [4], and by Edelsbrunner–Overmars [19] for their comparable memory space conservation for batch-reporting tasks.

It should also be mentioned that the formal exponents, $d(e)$ and $d^*(e)$, mentioned in Items (A) and (B) are very conservative estimates. *For many but not all* enactments e, our join algorithms from [67] *will actually produce* quasi-linear times with exponents d that are lower than these cautious estimates. Typically, but not always, d will equal zero or one.

2.2. New Results

We will explore how to generalize Willard's [67]. Reporting and Aggregation algorithms for the case where more than two variables are present. In particular, let the capital letter symbols $R_1\ R_2\ \dots\ R_k$ denote k sets of records. Let $Q_i\ (r_i \in R_i)$ denote an existential or universal quantifier for a variable r_i spanning a relation R_i. Let $e(r_1, r_2, \dots, r_k)$ 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 relational calculus "Find" query is denoted as

$$\begin{aligned}&\{\text{FIND}(r_1 r_2 \dots r_p) \in R_1 \times R_2 \times \cdots \times R_p \\ &\quad Q_{p+1}(r_{p+1} \in R_{p+1}) Q_{p+2}(r_{p+2} \in R_{p+2}) \dots Q_k(r_k \in R_k) : e(r_1 r_2 \dots r_k)\}.\end{aligned} \tag{3}$$

Let q denote the query above. Say its variable r_i *precedes* the variable r_j iff the quantifier or FIND-clause defining r_i lies to the left of r_j's definition in Eq. (3). Define this query's relational graph $G(q)$ to have a directed edge from r_j to r_i iff these two variables are the binary constituents of some equality, order or tabular atom and if r_i precedes r_j. Say the relational calculus query q satisfies the *RCS condition* iff its graph is a tree or forest with all paths leading to the roots.

Our main goal in the present article will be to present an algorithm that guarantees that every such "RCS FIND" query q runs in $O(I \operatorname{Log}^d I + U)$ WH-time and uses $O(I + U)$ space, where I denotes the cardinality of the input and U denotes the cardinality of the output (see the footnote[2] for I's formal definition). Moreover, our "quasi-linear" algorithm for obtaining this result will rely on a *decomposition method* that breaks the k-variable RCS query into a series of subroutine calls to the E-8 Reporting Join and Aggregation procedures of [67].

There is also one corollary to our main formalism that will broaden its main domain of the applicability significantly. Let the symbol "$\oplus$" denote an $O(1)$ time aggregation operator that admits an inverse operator (such as Addition, non-zero Multiplication or Count). Define a *Relational Calculus Aggregation Query* to be a database search whose output is the same as the output of the following 2-step process:

1. First, find the subset of $X \times Y$ that satisfies the RCS query below:

$$\begin{aligned}&\{\text{FIND}(x, y) \in X \times Y\ Q_1(r_1 \in R_1) Q_2(r_2 \in R_2) \dots \\ &\quad Q_k(r_k \in R_k) : e(x, y, r_1, r_2, \dots r_k)\}.\end{aligned} \tag{4}$$

2. Next, for each $\bar{x} \in X$, calculate a quantity $\text{Agg}(\bar{x})$, which is defined to be the sum of the $f(\bar{y})$-values (under aggregation operator "$\oplus$") for those y-records where the ordered pair $(\bar{x}, \bar{y})$ is one of Eq. (4)'s output elements. Output the set of ordered pairs $(\bar{x}, \text{Agg}(\bar{x}))$, where $\bar{x} \in X$.

This search process will be called an *RCS Aggregation Query* when query (4) satisfies the RCS graph property. The notational symbol "ListAgg^f" (below) will formally indicate presence of an RCS-aggregation query:

$$\begin{aligned}&\{\text{ListAgg}^f(x, y) \in X \times Y\ Q_1(r_1 \in R_1)\ Q_2(r_2 \in R_2) \dots \\ &\quad Q_k(r_k \in R_k):\ e(x, y, r_1, r_2, \dots r_k)\}.\end{aligned} \tag{5}$$

[2] The "input size" I designates the sum of the cardinalities of all the relations R_i that are input, together with the cardinalities of inputed tabular sections T_i, associated with the tabular atoms used in the query q.

If one were to execute the RCS aggregation query in the exact chronological 2-step manner, implied by the description above, then its performance would be governed by $O(I \operatorname{Log}^d I + J)$ WH-time and $O(I + J)$ space, where I denotes the cardinality of the input and J denotes the cardinality of Eq. (4)'s output. However, Section 5.3 will show that there is a better way to perform this task that instead runs in $O(I \operatorname{Log}^d I)$ WH-time and uses $O(I)$ space.

Section 5.3's algorithm is interesting because there has been an extensive discussion about database aggregation in the recent literature about OLAP queries [10, 16, 17, 24, 28–32, 40, 41, 45, 50, 51, 73, 76]. The virtue of Section 5.3's algorithm is that it requires no preprocessing of the data prior to the start of the algorithm, and it can compute the desired aggregation table in an efficient manner for *extremely complicated* relational calculus-like queries.

2.3. Review of Literature on Acyclic Database Schemes

This section will explain how the notion of an RCS query, with its graph-like query properties, is closely related to the literature on acyclic databases [1, 2, 8, 9, 20, 25, 26, 49, 52, 55, 70, 72]. Our work concerning the RCS language, sketched in rudimentary forms in [59, 60, 65], brings added perspective to the theory of acyclic databases. It demonstrates that all queries in the RCS language lend themselves to a form of acyclic optimization.

The notion of an acyclic database scheme is a broadly encompassing concept that has a large number of very elegant applications, many of which are unrelated to our particular purposes. A detailed description of some of the uses of acyclic database schemes has been provided by Beeri *et al.* [2]. Their Theorem 3.4 establishes a 12-way equivalence between different database conditions that explains, among other facts, how several different articles were converging in the late 1970s and early 1980s from various perspectives upon an idea that in some respects they had numerous equivalent representations and properties. Some formal aspects of the acyclicity concept are related to the notions of a lossless join and database join-dependency conditions, which are commonly cited in the database textbooks to reduce redundancy and improve database expressibility. Other aspects are related to database optimization problems. This latter feature is closely connected to our interest in RCS optimization.

The best way to summarize this connection is to let r_1, r_2, r_3 and r_4 denote four database relations whose attribute sets are respectively (A, B), (B, C), (C, D), and (D, A). Let t denote a 4-tuple whose attributes have names A through D, and let $\Pi_{AB}(t)$ denote an ordered pair that has identical values on its AB attributes as t. In this notation, the "Natural Join" $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$ is defined as the set of tuples t where $\Pi_{AB}(t)$, $\Pi_{BC}(t)$, $\Pi_{CD}(t)$ and $\Pi_{DA}(t)$ belong to the respective relations of r_1, r_2, r_3, and r_4. Beeri *et al.* [1, 2] have used the term "cyclic" to characterize this join-query, but they would call the join-query $r_1 \bowtie r_2 \bowtie r_3$ "acyclic." They use this terminology because:

1. If one thinks *roughly* of each relation r_i as representing the edge of a graph then there is a natural cycle inherent in $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$, by starting at "$A$", following the edge r_1 to B, and then proceeding, respectively, to the further nodes of

"*C*", "*D*" and "*A*" by following the respective edges of r_2, r_3 and r_4. The presence of this "*ABCDA*" cycle is the basic reason that [1, 2] would characterize this join-query as "cyclic".

2. Since the query $r_1 \bowtie r_2 \bowtie r_3$ contains no r_4 edge (and thus contains no analog of the "*ABCDA*" cycle), Beeri *et al.* [1, 2] refer to it as "acyclic."

From the standpoint of what is relevant to our research, the critical aspect of the theory of acyclicity is that it implies that all acyclic natural joins (roughly similar to join-query (2)) can be executed efficiently by decomposing them into efficiently operating modular parts. On the other hand, Beeri *et al.* [2] prove that there is no analog of this result for cyclic queries, because their Theorem 3.4 implies that there would then be no available access to the type of semi-join-like database full-reducer operations, that have been used very successfully by Bernstein and Chiu [8], Bernstein and Goodman [9], and Yu *et al.* [72].

The preceding paragraph's overview of acyclic join queries had deliberately omitted many details because we were trying to focus only on those aspects of the literature that are relevant to RCS database optimization problems. For instance Beeri *et al.*'s [2] formal definition of an acyclic join is substantially more complicated than what is evident from the previous paragraph's examples because it uses hyper-graphs and hyper-edges in its definition of acyclicity rather than ordinary graphs and edges.

In essence, join-acyclicity is applicable to our research as a device for *modularly decomposing* a larger k-variable relational calculus query satisfying the RCS condition into some smaller efficient 2-variable E-8 enactment components.

Much of the focus of our study of acyclic optimization is, however, different from the emphasis of the research of say Bernstein and Goodman [9], Yannakakis [70] and Yu *et al.* [72]. This is because we examine a relational calculus rather than relational algebra language. Several topics studied by Yannakakis [70], such as testing database dependencies, inferring other dependencies, connecting these concepts to Lien's notion of a loop-free Bachman scheme [36] and Zaniolo's notion of a simply connected scheme [74], etc., are quite important but not directly related to our main objectives in the present paper: therefore, we refer the reader directly to Yannakakis's paper for more about them. Of interest to us is the fact that Yannakakis's algorithm [70] and the related work of Bernstein and Goodman [9] and Yu *et al.* [72] can compute a relational algebra projection from the intermediate result begotten from an acyclic-join operation. Since the relational projection operation is similar to an existential quantifier in the relational calculus language, this facet of these algorithms is roughly analogous to a special form of Eq. (3)'s RCS query where all its quantifiers Q_i are existential quantifiers and its atoms within its body expression $e(r_1 r_2 \ldots r_k)$ are composed of, say, a conjunction of equality atoms. Moreover, it is evident that one can further generalize these algebraic algorithms to more complicated $e(r_1 r_2 \ldots r_k)$ that are comprised of, say, an arbitrary combination of equality and list atoms linked together in an arbitrary manner by the AND and OR connective symbols.

Our work concerning RCS extends the theory of acyclic databases by showing how the *quasi-linear* search complexities generalize to the full RCS language, with its relational calculus features. These features permit Eq. (3)'s body expressions $e(r_1 r_2 \dots r_k)$ to be comprised of an arbitrary combinations of equality, list, tabular and order atoms—linked together in an arbitrary manner by the AND, OR and NOT connective symbols. Also, RCS allows the quantifier Q to be any one of an existential, universal or generalized quantifier (where the latter "generalized notion" will be defined in Remark 4.2). Moreover, the RCS operation of "ListAgg" facilitates the speed of many aggregation queries in an OLAP environment.

3. FOUR EXAMPLES OF E-8 ENACTMENT PROCEDURES

This section will give four examples summarizing Willard's [67] algorithm for processing E-8 enactments. Because we intend [67] to be the main source about this subject, these examples will not be fully informative. Moreover, it is "technically" unnecessary for the reader to examine this section because Sections 4–6 treat the E-8 reporting and aggregate join algorithms as essentially *Black Boxes*, whose performance complexities are adequately summarized by Items (A) and (B) of Section 2.1. *No additional information* about the E-8 procedures is "strictly" needed in Sections 4–6. Thus, it is reasonable for a reader to either to skip this section entirely, or to examine our four examples with meticulous care.

Notation Employed in our Examples: For a fixed set of tuples Y and a fixed E-8 enactment $e(x, y)$, the symbol $D_e(Y)$ will denote an *on-line data structure*, which given an input $\bar{x}$ is capable of finding the subset of tuples $\bar{y} \in Y$ satisfying $e(\bar{x}, \bar{y})$. This set of tuples will be denoted as $Y_e(\bar{x})$. Also, for a function $f(\bullet)$ that maps the elements of Y into an abelian group, the symbol $\Phi_e^f(\bar{x})$ will denote the $\sum f(\bar{y})$ for those elements $\bar{y} \in Y$ satisfying $e(\bar{x}, \bar{y})$. An on-line data structure that allows one to calculate $\Phi_e^f(\bar{x})$ in poly-logarithmic time will be typically denoted as $D_e^f(Y)$.

Although the main purpose of our discussion will be to illustrate examples of the E-8 Reporting and Aggregate Join Procedures, we will sometimes veer from this topic and discuss on-line poly-logarithmic search processes, as well. We do so in those cases where the off-line procedure is essentially the same as the on-line search algorithm. In such cases, it is natural to discuss both topics together.

Example 3.1. We used the term E-3 Enactment in [67] to refer to the subset of E-8 enactments whose only atoms are equality atoms (connected in an arbitrary manner by the AND, OR and NOT connective symbols). Two examples of E-3 enactments are given below:

$$e(x, y) =_{\mathrm{df}} \{A_1(x) = B_1(y) \ \vee \ A_2(x) = B_2(y)\}, \tag{6}$$

$$e^*(x, y) =_{\mathrm{df}} \{A_1(x) = B_1(y) \ \wedge \ A_2(x) \neq B_2(y)\}. \tag{7}$$

Let $Y_{B_1, B_2, \dots B_k}(c_1, c_2, \dots c_k)$ denotes the subset of Y satisfying

$$B_1(y) = c_1 \ \wedge \ B_2(y) = c_2 \ \wedge \ \dots B_k(y) = c_k. \tag{8}$$

Also, $I^f_{B_1, B_2, \dots B_k}(c_1, c_2, \dots c_k)$ denotes the $\sum f(\bar{y})$ for the $\bar{y} \in Y$ satisfying (8).

Let $H^f_{B_1,B_2,\dots B_k}$ denote a hash file that allows one to find $I^f_{B_1,B_2,\dots,B_k}(c_1, c_2, \dots, c_k)$'s value in $O(1)$ time, and let N_y denote the cardinality of Y. Section 5 of [67] proved that every E-3 enactment $e(x, y)$ can be associated with an $O(N_y)$ space dynamic data structure, called $D^f_e(Y)$, that allows us to calculate in $O(1)$ time the value of $\Phi^f_e(\bar{x})$. For instance, $D^f_e(Y)$ will equal the union of the hash indices $H^f_{B_1}$, $H^f_{B_2}$ and $H^f_{B_1,B_2}$ when e corresponds to Eq. (6)'s E-3 enactment. One can calculate the value of $\Phi^f_e(\bar{x})$ by doing three $O(1)$ time searches into these hash indices and then using the following equation to derive the answer:

$$\Phi^f_e(\bar{x}) = I^f_{B_1}(A_1(\bar{x})) + I^f_{B_2}(A_2(\bar{x})) - I^f_{B_1,B_2}(A_1(\bar{x}), A_2(\bar{x})). \tag{9}$$

In the example of Eq. (7), $D^f_{e^*}(Y)$ will equal the union of the two aggregate hash indices $H^f_{B_1}$ and $H^f_{B_1,B_2}$. Its analogous arithmetic computation will be

$$\Phi^f_{e^*}(\bar{x}) = I^f_{B_1}(A_1(\bar{x})) - I^f_{B_1,B_2}(A_1(\bar{x}), A_2(\bar{x})). \tag{10}$$

Section 5 of [67] formally proves that this methodology generalizes to all E-3 on-line queries. It also indicates that an analogous Aggregation Join algorithm[3] will consume $O(N_x + N_y)$ WH-time, and it will use $O(N_x + N_y)$ space for any E-3 enactment.

Example 3.2. One reason why the preceding example was interesting is that its $O(N_x + N_y)$ WH-time is quite evidently an attractive complexity for doing an aggregate join for a main-memory resident data structure. Another reason for presenting Example 3.1 is that it will enable us to discuss the crucial distinctions between the Aggregate and Reporting Join problems.

In particular, *a very curious facet* of the E-3 Reporting Join is that it is logically correct *but extremely inefficient* to extend Example 3.1's methodology to Reporting Joins. For instance, consider a reporting algorithm that is the same as Example 3.1's aggregation procedure except that it replaces the operations of aggregate-addition and aggregate-subtraction with set-union and set-subtraction. Using the notation from Example 3.1, the analogs of Eqs. (9) and (10) for calculating the values of $Y_e(\bar{x})$ and $Y_{e^*}(\bar{x})$ would then be

$$Y_e(\bar{x}) = Y_{B_1}(A_1(\bar{x})) \cup Y_{B_2}(A_2(\bar{x})), \tag{11}$$

$$Y_{e^*}(\bar{x}) = Y_{B_1}(A_1(\bar{x})) - Y_{B_1,B_2}(A_1(\bar{x}), A_2(\bar{x})). \tag{12}$$

It is clear that (11) and (12) provide a correct construction of the $Y_e(\bar{x})$ and $Y_{e^*}(\bar{x})$ sets. However, the *efficiency of these procedures* is more problematic.

The difficulty is essentially due to the fact that the addition and subtraction of aggregate quantities can be done in $O(1)$ time under most models of computation, but the operations of set-union and set-subtraction are much more expensive. For

[3] It is easy to illustrate our Aggregate Join algorithm for the example of the two queries e and e^*. It will simply first build the needed aggregate hash indices for $D^f_e(Y)$ and $D^f_{e^*}(Y)$ in $O(N_y)$ time and then make $O(N_x)$ separate queries into these freshly built data structures (using respectively (formulae (9) and (10)). The total cost of these Aggregate Joins is thus $O(N_x + N_y)$ WH-time and $O(N_x + N_y)$ space.

example, let j, m and n denote the cardinalities of the sets $Y_{e^*}(\bar{x})$, $Y_{B_1}(A_1(\bar{x}))$ and $Y_{B_1,B_2}(A_1(\bar{x}),A_2(\bar{x}))$ from Eq. (12). The set subtraction operation appearing in this equation clearly consumes time $O(1+m+n)$ when it is implemented in a standard manner. It turns out that a more efficient *on-line dynamic algorithm* will be capable of searching an $O(N_y)$ space index structure to construct the $Y_{e^*}(\bar{x})$ set in $O(1+j)$ WH-time.

The latter time is obviously optimal because it is clearly impossible to achieve a worst-case time for computing $Y_{e^*}(\bar{x})$ below an $O(1+j)$ magnitude when the output consists of j tuples. It is also clearly much more efficient than the previous paragraph's $O(1+m+n)$ time because one can easily construct examples where say $j=1$, $m \gg j$ and say $n=m-1$.

Moreover, what will makes our two examples especially interesting is that their $O(1+j)$ WH-times will generalize for all *on-line* E-3 reporting queries.

Some notation will be needed to explain how we can handle E-3 reporting queries efficiently. Let the symbol $Y(\frac{B_1\to c_1 B_2\to c_2 \ldots B_i\to B_i}{P_1\to d_1 P_2\to d_2 \ldots P_k\to d_k})$ denote the set of $y\in Y$ satisfying the condition:

$$\begin{aligned} B_1(y) = c_1 \wedge B_2(y) = c_2 \wedge \ldots B_i(y) \\ = c_i \wedge P_1(y) \neq d_1 \wedge P_2(y) \neq d_2 \wedge \ldots P_i(y) \neq d_k. \end{aligned} \tag{13}$$

Subsets of Y of the form "$Y(\frac{B_1\to c_1 B_2\to c_2 \ldots B_i\to B_i}{P_1\to d_1 P_2\to d_2 \ldots P_k\to d_k})$" will be called *y-sublists*. The symbols $Y_1, Y_2, Y_3 \ldots$ will denote such y-sublists. (The empty set symbol "ϕ" will appear in the numerator or denominator of the y-sublist notation when this relevant portion of the y-condition is empty.)

Let us first consider an admittedly impractical but "idealistic" data structure, called $\overline{D(Y)}$, that stores within itself every possible y-sublist that contains at least one tuple. Such a data structure can obviously trivially answer every on-line E-3 reporting query in $O(1+j)$ time (where j is the size of the output). For instance, Eqs. (14) and (15) illustrate how this formalism can trivially construct the answer to the on-line reporting queries of $Y_{e^*}(\bar{x})$ and $Y_e(\bar{x})$:

$$Y_{e^*}(x) = Y\left(\frac{B_1 \to A_1(x)}{B_2 \to A_2(x)}\right), \tag{14}$$

$$Y_e(x) = Y\left(\frac{B_1 \to A_1(x)}{B_2 \to A_2(x)}\right) \cup Y\left(\frac{B_2 \to A_2(x)}{\phi}\right). \tag{15}$$

The difficulty with using this "idealistic" data structure $\overline{D(Y)}$ is that it clearly uses an unrealistically large amount of memory space to store every possible non-empty y-sublist. Our ability in [67] to reduce the memory space to a more desirable $O(N_y)$ size *while retaining* the "idealistic" retrieval time rested on using a type of Factor-2 relaxation method.

In particular, let N_y again denote the cardinality of Y, and let $e(x,y)$ again denote an E-3 enactment. Section 6 of [67] showed that when such values for Y and e are specified in advance, it is possible to construct a corresponding $O(N_y)$ space data structure, called $D_e(Y)$, which assured that every possible on-line $Y_e(\bar{x})$ query would run in an $O(1+j)$ time. Moreover, the coefficient associated with the O-notation's

j-term will exceed the coefficient of the "idealistic" search time by no more than a factor of 2.

The intuitive idea behind $D_e(Y)$'s data structure was to avoid the temptation to store every possible distinct y-sublist. Instead, we stored only a *select tiny fraction* of all the available y-sublists, called say Z, subject to the following three constraints:

1. For each query element $\bar{x}$, there will exist a finite number L of pairwise disjoint y-sublists $Y_1, Y_2, \ldots Y_L$ in Z such that $Y_e(\bar{x})$ is a subset of the union of these lists, *and such that* the sum of the cardinalities of these lists will be never more than twice the cardinality of $Y_e(\bar{x})$. (This latter fact assures that the cost to construct $Y_e(\bar{x})$ will exceed by no more than roughly a factor of 2 the comparable cost of searching the idealized data structure $\overline{D(Y)}$.)
2. The particular value of L (above) will be no greater than some constant K_e, whose value depends only on the E-8 predicate e. (Typically, K_e's value will be moderately small.)
3. The data structure $D_e(Y)$ will use no more than $O(N_y)$ space. (Since each y-sublist Y_i uses a memory space proportional to its cardinality, this constraint imposes a severe restriction on *both* the number and the sizes of the y-sublists that are allowed in our data structure.)

Section 6 of our article [67] gave a combinatorial proof showing that for every E-3 enactment predicate e, it is possible to build a resulting data structure $D_e(Y)$ that meets the preceding constraints. Moreover, it provided this data structure with a hash-searching scheme that allowed us to find the needed y-sublists in $O(1)$ WH-time and which supported $O(1)$ insert and delete operations.

As an on-line dynamic data structure, the pragmatic implications of Willard's [67] proposal for E-3 search queries are unclear because a separate data structure could possibly be needed for each separate E-3 enactment (in the worst case). However, this methodology is tempting to employ at least when one is doing E-3 *Reporting Joins*. This is because we can use a "Build-and-Throw-Away" strategy for the reporting join. Thus starting initially with raw unprocessed data, Willard's [67] reporting join algorithm can first build the needed data structure, then do a lengthy sequence of $O(N_x)$ queries into it, and finally throw away the data structure at the end of computation. This method is reasonably cost-effective for doing reporting join operations because it consumes $O(N_x + N_y + N_e)$ WH-time, where N_e is the cardinality of the output, and N_x and N_y are the cardinalities of X and Y.

Example 3.3. This example will illustrate how Section 9 of [67] processes tabular atoms. Let $e(x, y)$ denote an E-8 enactment, and define $e_\pi(x, y)$ to be an enactment formed by replacing each of e's tabular atoms with the Boolean constant of FALSE (and then algebraically simplifying the result). The enactment e_π is called a π-reduction of e. Following equations illustrate an example:

$$\begin{aligned} e(x,y) =_{\mathrm{df}} \{&[A_1(x) = B_1(y) \wedge \neg(x,y) \in T_1] \vee A_2(x) = B_2(y) \vee , \\ &[A_3(x) = B_3(y) \wedge (x,y) \in T_2)]\}, \end{aligned} \tag{16}$$

$$e_{\pi}(x,y) =_{\mathrm{df}} \{(A_1(x) = B_1(y) \vee A_2(x) = B_2(y)\}. \qquad (17)$$

Also, let ADD(X, Y, e) and SUBTRACT(X, Y, e) denote the subsets of $X \times Y$ satisfying the conditions "$e(x,y) \wedge \neg e_{\pi}(x,y)$" and "$e_{\pi}(x,y) \wedge \neg e(x,y)$", respectively.

Our algorithm for doing E-8 joins in [67] was essentially a 2-part procedure that first performed the relevant join operation on the π-reduced enactment e_{π} and then incrementally corrected the answer by walking down the ADD(X, Y, e) and SUBTRACT(X, Y, e) subsets. For the example of doing Eq. (16)'s aggregate join, this procedure would consist of the following two steps:

1. First, use Example 3.1's algorithm for calculating e_{π}'s aggregate-join.
2. Let the array $\Phi(x)$ store this result. Increment $\Phi(\bar{x})$ by an amount $f(\bar{y})$ for each $(\bar{x}, \bar{y})$ in the set ADD(X, Y, e), and decrement it by $f(\bar{y})$ for each $(\bar{x}, \bar{y})$ in SUBTRACT(X, y, e). At the end of this computation, $\Phi(x)$ will store the answer to e's aggregate join.

This aggregate join will clearly consume $O(N_x + N_y + N_t)$ WH-time, since its Step 1 was shown by Example 3.1 to use $O(N_x + N_y)$ WH-time, and its Step 2 needs $O(N_t)$ WH-time to walk down the tabular sections (of size $O(N_t)$) for accordingly adjusting the result. A similar computational cost, where the E-8 enactment requires $O(N_t)$ more time than its π-reduced counterpart, generalizes to all other E-8 queries for *both the cases* of an aggregate and reporting join (see Section 9 of [67]).

It is important to explain exactly why the preceding time complexity is interesting. Its complexity is no better than the asymptotic cost of a trivial exhaustive search in the degenerate case when $N_t \cong N_x \cdot N_y$. However, it is far better than an exhaustive search when $N_t \ll N_x \cdot N_y$. The point is that the latter inequality characterizes the cardinality of the tabular sections in *many* database examples. For instance, consider an insurance company's database that lists the family members that are anticipated to drive a particular car. This relationship is neither many-to-one, nor one-to-many, but yet if, say 10^6, denotes the order of magnitude of the number of cars and drivers then the relevant number of ordered pairs will clearly possess an $O(10^6)$ rather than $O(10^{12})$ magnitude. And an analogous sparse tabular relationship will also characterize a database listing adult guardians of a child, residents living in a house, and many more similar examples.

Thus, one reason why our article [67] gave a very formal and complete description of our algorithm for processing tabular atoms is that we anticipated that this construct should be very useful in many database applications (especially when it is hybridized via an E-8 enactment with our other range-query optimization features). A second equally nice feature of the tabular-atom construct will be explained by Example 4.3 (in the next chapter).

EXAMPLE 3.4. Define a *y-range term* to be an expression of the form

$$A_1(x) < B^*(y) < A_2(x). \qquad (18)$$

Define an *orthogonal range query* to be a conjunction of several range terms, similar to say:

$$A_1(x) < B_1(y) < A_2(x) \wedge A_3(x) < B_2(y) < A_4(x). \tag{19}$$

Bentley [4] introduced a 2-part data structure for answering 2-dimensional orthogonal range queries, called a 2-fold tree, and Lueker and Willard [37, 69] developed a more elaborate dynamic version of this data structure. Its first part, called the *base*, was essentially a binary tree, of height $O(\text{Log}\, N_y)$, whose leaves are the elements of Y arranged in the order of increasing $B_1(y)$ value. Henceforth, YSET(v) will denote the subset of Y descending from the tree node v. The 2-fold tree will assign each node v a pointer to an auxiliary data structure AUX(v), consisting of an alternate tree-representation of YSET(v), where these records are instead arranged in the order of increasing $B_2(y)$ value. The 2-fold tree can thus be thought of as a tree indexing a forest of trees that occupy $O(N_y \,\text{Log}\, N_y)$ space.

Suppose that we are given a query element $\bar{x}$ asking to find the $\sum f(y)$ for those y-records satisfying (19). The 2-fold tree suggests a natural divide-and-conquer algorithm for answering this query. It is described below:

1. Define a node v to be *critical* with respect to (19) if every $\bar{y} \in \text{YSET}(v)$ satisfies $A_1(\bar{x}) < B_1(\bar{y}) < A_2(\bar{x})$ and v's parent does not meet this criterion. Use a binary search to find the $O(\text{Log}\, N_y)$ or fewer critical nodes.

2. For each critical node v_i, search its AUX(v_i) field in $O(\text{Log}\, N_y)$ time to find the $\sum f(\bar{y})$ for those $\bar{y} \in \text{YSET}(v)$ satisfying $A_3(\bar{x}) < B_2(\bar{y}) < A_4(\bar{x})$. Let $S(v_i)$ denote this subtotal, and let C denote the set of all critical nodes found by Step 1. Then the answer to Eq. (19)'s aggregation query is begotten by calculating the $\sum S(v)$ for those $v \in C$.

It is clear that the preceding algorithm can perform a 2-dimensional on-line orthogonal range query request in $O(\text{Log}^2 N_y)$ worst-case time, and that its natural d-dimensional generalization consists of a data structure using $O(N_y \,\text{Log}^{d-1} N_y)$ space and having an $O(\text{Log}^d N_y)$ search time.

Lueker and Willard [37, 69] had developed a dynamic version of the d-fold tree structure with an $O(\text{Log}^d N_y)$ worst-case time for insertions and deletions. Fredman [21] established a lower bound showing that this result was time-optimal for dynamic aggregate queries over a semi-group operator. However, the subsequent literature has shown that there exists many more elaborate and detailed generalizations of the d-fold trees that can provide improvements from other perspectives. For instance, we developed a memory compressed version of the Lueker and Willard aggregation data-structures in [63] and a faster version for most types of static on-line queries in [61]. The latter is seemingly very pragmatic: it is based on interconnecting the auxiliary fields of the 2-fold trees with a network of pointers, so that (without increasing the memory space) one can save an $O(\text{Log}\, N_y)$ factor of time by avoiding costly repetitions of a similar binary search into several neighboring auxiliary fields. (This method was called the down-pointer technique by us [61], and Chazelle and Guibas [15] later developed a more general form of it, called Fractional Cascading, that applied to a variety of problems in Computational Geometry.) There are many other useful results in the literature on orthogonal queries, and there is no space in this abbreviated section to survey the full literature.

Of special interest to us is that d-fold trees (and their sundry generalizations) can have their memory spaces compressed to an $O(N_y)$ size when one is doing an *aggregate or reporting* join. This is because one can then treat the Aux(v) fields as representing *virtual* rather than *actual* data structures. That is, a full d-fold tree is never built when doing a Join because its memory size is excessively large. Instead when given as input two initial sets X and Y, whose elements are say $\bar{x}_1, \bar{x}_2, \ldots \bar{x}_n$ and $\bar{y}_1, \bar{y}_2, \ldots \bar{y}_m$, the strategy is to build only a tiny fraction of the AUX fields at one time and to have all elements $\bar{x}_i \in X$ that need to query a particular AUX(v) field to do so during the precise short interim period of time when it is built. (For the example of a 2-fold tree, the implementing algorithm will essentially walk through the tree's base section, *build the AUX*(v) *fields at only one level of the tree at a time*, run all the needed queries $\bar{x}_1, \bar{x}_2, \ldots \bar{x}_n$ against these AUX(v) fields during the interim period when they are available, and then use the prior memory space to construct the AUX(v) fields for the next tree level.)

This type of strategy was first employed by Bentley [4] for the case of ECDF calculation, and Edelsbrunner and Overmars [19] used it for a batched sequence of on-line reporting orthogonal range queries. Our paper [67] showed how its space-saving technique can apply to the reporting and aggregation variants of E-8 join queries (by essentially hybridizing the methodologies of the four examples given in this section with the $O(\text{Log}\, N_y)$ savings in time and memory compression methods, mentioned in the last two paragraphs).

We stated at the beginning of this section that we would not seek to fully describe our algorithms for doing E-8 Reporting and Aggregation Joins because that topic was already discussed by us in [67]. Rather our objective was to give an intuitive summary of Willard's [67] algorithms through four examples. The reader does not need to know more about Willard's [67] algorithm to follow the remainder of this paper, so long as he treats the E-8 algorithms as essentially "*Black Boxes*", whose time and space complexities are summarized by Items (A) and (B) of Section 2.1.

4. NOTATION AND EXAMPLES OF RCS SEARCHES

Our algorithm for decomposing a general RCS query into an efficient block of executing E-8 enactment operations will appear in the next section. The two goals of this section are to provide some useful examples and to introduce notation that will be used in the next section.

LEMMA 4.1. *Consider the two "binary" relational queries operations below*:

$$\{\text{FIND}(x) \in X \ \exists y \in Y : e(x,y)\}, \tag{20}$$

$$\{\text{FIND}(x) \in X \ \forall y \in Y : e(x,y)\}. \tag{21}$$

The E-8 enactment formalism of [67] *provides a method for answering these queries using* $O((N_x + N_y)\text{Log}^{d^*(e)} N_y + N_t)$ *WH-time and* $O(N_x + N_y + N_t)$ *space.*

Proof. In a context where the article [67] supplies us with a library of subroutines for executing every E-8 aggregate join algorithm efficiently, the added formalism needed by Lemma 4.1 is basically trivial. Lemma 4.1's procedure will begin by asking [67]'s aggregate join algorithm to construct the array $\Phi_e(x)$, which indicates how many $\bar{y} \in Y$ satisfy $e(x, \bar{y})$. We will then output those elements $\bar{x} \in X$ satisfying $\Phi_e(\bar{x}) \geqslant 1$ when seeking to find the x-elements satisfying Eq. (20)'s existential quantifier. An analogous algorithm will process Eq. (21) by outputting those $\bar{x} \in X$ satisfying $\Phi_e(\bar{x}) = \mathit{Cardinality}(Y)$. Both universal and existential quantifiers can thus be processed in the claimed quasi-linear time under the E-8 Aggregation formalism. ∎

Remark 4.1. We anticipate that Lemma 4.1's operating exponent should equal zero or one in most settings where it is typically used.[4]

Remark 4.2. Let $Q(y)$ denote any function that returns a Boolean value as a function of the number of elements $y \in Y$ satisfying a specified enactment condition $e(x, y)$. This includes the possibility of a MAJORITY(y) quantifier which returns the value TRUE when over half the tuples $y \in Y$ satisfy $e(x, y)$, an EVEN(y) quantifier which tests to see if an even number of elements $y \in Y$ satisfy this condition, a "2-existence quantifier" which tests to see if at least two distinct elements $y \in Y$ satisfy $e(x, y)$, etc. We can thus think of each quantifier Q as a mapping of an "E-8 enactment array" $\Phi_e^f(\bar{x})$ onto an array of Boolean values. We will use the term *Generalized Quantifier* to refer to such a mapping. Lemma 4.1 obviously generalizes to the case where we replace its existential and universal quantifiers with such "Generalized Quantifiers". We will use this generalization throughout the rest of this paper. Thus, we will permit our k-variable RCS queries to include generalized quantifiers, in addition to universal and existential quantifiers.

Definition 4.1. We will use the term *Binary Procedure* to refer to an algorithm that finds the set of elements $\bar{x} \in X$ satisfying a query, similar to the tests for universal and existential quantifier conditions (given in Lemma 4.1) or its generalization for "Generalized Quantifiers" (given in Remark 4.2). Each such binary algorithm can be thought of as outputting a "subsection" list L enumerating those particular elements $\bar{x} \in X$ satisfying the quantifier concerned. A *QL-Listing Procedure* will be defined to be an algorithm that produces some finite collection of such lists $L_1, L_2, \ldots L_j$ using essentially Lemma 4.1's quasi-linear procedure. We will say that an initial relational calculus query q is *QL-reduced* to a second relational calculus query q^* iff

1. the FIND-clauses for q and q^* produce identical outputs and
2. the query q^* contains distinctly fewer quantifiers than does the query q.

Henceforth, condition 1 (above) will be called *output-equivalence*.

[4]The formal definition of $d^*(e)$ had appeared in Section 2.1. One reason Lemma 4.1's operating exponent will usually (not always) equal zero or one is that $d^*(e)$ was simply defined so that $d^*(e) = d(e) - 1$ whenever $d(e) \geqslant 2$. A further reason why this component is usually quite small was explained by Section 2.1's last paragraph.

EXAMPLE 4.1. The intuitive reason Definition 4.1 allows two queries to be "output-equivalent" despite the fact that one uses fewer quantifiers is due to the use of list atoms. For instance, let e_1, e_2 and e_3 denote three enactment predicates, and consider the query below:

$$\{\text{FIND}(x, y, z) \in X \times Y \times Z\ \forall w \in W : e_1(x, y) \wedge e_2(y, z) \wedge e_3(z, w)\}. \tag{22}$$

Let L^* denote the "subsection" list, itemizing those elements $\bar{z} \in Z$ satisfying

$$L^* = \{\text{FIND}(z) \in Z \forall w \in W : e_3(z, w)\}. \tag{23}$$

The set of ordered pairs satisfying (22) is clearly the same as the set:

$$\{\text{FIND}(x, y, z) \in X \times Y \times Z : e_1(x, y) \wedge e_2(y, z) \wedge z \in L^*\}. \tag{24}$$

Thus Eq. (24) is an example of a QL-reduction of Eq. (22).

EXAMPLE 4.2. Let us now continue the preceding example and ask how to find the set of tuple records satisfying query (22). Since (24) is a QL-reduction of (22), one correct *but extremely inefficient* procedure for resolving this query is the following 3-step procedure:

1. First, use an enactment join to find those $(\bar{x}, \bar{y})$ satisfying $e_1(\bar{x}, \bar{y})$.
2. Next, use an enactment join to find the $(\bar{y}, \bar{z})$ satisfying $e_2(\bar{y}, \bar{z}) \wedge \bar{z} \in L^*$. (This step is permissible because if "$e_2(y, z)$" is an E-8 enactment *then by definition*, the slightly more complicated predicate "$e_2(y, z) \wedge z \in L^*$" is obviously also an E-8 enactment. This fact implies that one of Willard's [67] "Reporting Join Algorithms" can find the set of ordered pairs (y, z) satisfying this predicate.)
3. Let G and H denote the two sets constructed by steps 1 and 2. Then the answer to query (24) is the "natural join" of these two sets, i.e. it is the set of ordered triples $(\bar{x}, \bar{y}, \bar{z})$ satisfying $(\bar{x}, \bar{y}) \in G$ and $(\bar{y}, \bar{z}) \in H$.

An interesting facet is that the procedure (above) is correct but *not efficient enough to satisfy the quasi-linear cost criteria.* To illustrate the difficulty, let us consider an example where:

1. The sets W, X, Y, and Z each have cardinality equal to N.
2. The two sets G and H each have cardinality equal to $N^2/2$.
3. The final output from query (22) is nevertheless empty.

Then in this case, the "input size" $I = 4N$, the "output size" $U = 0$, and the preceding algorithm is certainly *NOT* quasi-linear efficient because its first two steps will require $O(N^2)$ time to construct very large intermediate sets of size $N^2/2$. (This amount of time is obviously too large to be a "quasi-linear" function of our input and output sizes of I and U.)

The curious facet is we can indeed process (22) in quasi-linear WH-time if we use a more subtle form of QL-reduction procedure. This more elaborate procedure will differ from the example above by making three (rather than one) subroutine calls to binary procedures for producing intermediate lists that will assist in producing the final output. One of these lists, L^*, is the same as the list we used in Step 2 of the preceding algorithm. (It was defined by (23).) The other two lists produced by the QL-reduction stage of our algorithm, L_1 and L_2, are new. They are defined below by (25) and (26). (Note that the expression "$e_2(y,z) \wedge z \in L^*$" in (26) is an E-8 enactment simply because $e_2(y,z)$ was. Hence, Lemma 4,1's procedure can construct the two lists below.)

$$L_1 = \{\text{FIND}(y) \in Y \quad \exists x \in X : e_1(x,y)\}, \tag{25}$$

$$L_2 = \{\text{FIND}(y) \in Y \quad \exists z \in Z : [e_2(y,z) \wedge z \in L^*]\}. \tag{26}$$

From the definitions of L^*, L_1 and L_2, it is immediate that query (22) has an output identical to the set of elements satisfying query (27). (In Definition 4.1's terminology, this is the same as simply saying that query (27) is a "*QL-reduction*" of (22).)

$$\begin{aligned}&\{\text{FIND}(x,y,z) \in X \times Y \times Z):\\ &\quad [e_1(x,y) \wedge y \in L_2] \wedge [e_2(y,z) \wedge y \in L_1 \wedge z \in L^*]\}.\end{aligned} \tag{27}$$

Using the fact that (27) is a "QL-Reduction" of (22), we can use (27) as an alternate method for finding the records satisfying (22). This procedure appears below:

1. First, apply Willard's [67] E-8 Reporting-Join algorithm to produce the subset G^* of $X \times Y$ that satisfies the first of (27)'s two square bracket expressions.
2. Next, apply an E-8 Reporting-Join Enactment algorithm to produce the subset H^* of $Y \times Z$ that satisfies (27)'s second square bracket expression.
3. Finally, answer query (27) by taking the natural join of G^* and H^*.

It is easy to prove that unlike G and H, the sets G^* and H^* both satisfy the inequalities Cardinality $(G^*) \leqslant U$ and Cardinality $(H^*) \leqslant U$. These inequalities imply that our second algorithm always runs in quasi-linear WH-time, unlike the first algorithm. This is because the QL-reductions needed to produce the three lists L^*, L_1 and L_2 have an $O(I \operatorname{Log}^d I)$ cost (for some fixed constant d), and the additional costs for steps 1–3 are, respectively, $O(I \operatorname{Log}^d I + \text{Cardinality}(G^*))$, $O(I \operatorname{Log}^d I + \text{Cardinality}(H^*))$ and $O(I \operatorname{Log}^d I + U)$. In this context, the inequalities Cardinality $(G^*) \leqslant U$ and Cardinality $(H^*) \leqslant U$ imply that the sum of all the preceding time-costs is $O(I \operatorname{Log}^d I + U)$.

A question that naturally arises is whether or not the preceding example about the usefulness of QL-reductions always generalizes? In other words, is it true that every RCS query can have a similar quasi-linear complexity-cost, if one uses some form of procedure, using QL-reductions, to simplify them? Theorem 5.3 in the next section will give an affirmative answer to this question.

EXAMPLE 4.3. Finally, we will 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 [55].) Let R_3 be a third relation such that $(r_1, r_2) \in T$ exactly when there exists a corresponding $r_3 \in R_3$ with $A_1(r_3) = A^*(r_1)$ and $A_2(r_3) = A^*(r_2)$. 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

$$\{\exists\ r_3 \in R_3 : A_1(r_3) = A^*(r_1)\ \wedge\ A_2(r_3) = A^*(r_2)\}. \tag{28}$$

For example, consider the queries q_1 and q_2 (in Eqs. (29) and (30)). Since they produce the same output, it is reasonable for a reader to inquire why the tabular atoms is needed? After all, the query q_1 can specify the same set of tuples as q_2 *without the burden of this added notation*:

$$\begin{aligned} q_1 &= \{\text{FIND}(r_1, r_2) \in R_1 \times R_2\ \exists r_3 \in R_3: \\ A_1(r_3) &= A^*(r_1)\ \wedge\ A_2(r_3) = A^*(r_2)\ \wedge\ A_4(r_1) > A_5(r_2)\}, \end{aligned} \tag{29}$$

$$q_2 = \{\text{FIND } (r_1, r_2) \in R_1 \times R_2 : (r_1, r_2) \in T\ \wedge\ A_4(r_1) > A_5(r_2)\}. \tag{30}$$

The answer to this question rests on comparing the relational graphs of q_2 and q_1. The graph of q_2 is a tree, but q_1's graph is not.[5] Formally, this means that q_2 satisfies Section 2.2's definition of the "RCS-requirement," but the "output-equivalent" q_1 technically does not. The point is that Tabular Atoms are a formalism for signaling the fact that some relational calculus queries, such as q_1, can be processed in quasi-linear time *despite the fact* that their graphs technically do not satisfy the RCS-graph requirement. (This is because q_1 is "output-equivalent" to an "*RCS*" query q_2.)

We will return to Tabular atoms in Section 6.2. It will explain that this construct is also useful in modeling the "many-to-one", "one-to-one" and other sparse representations of a database set [55].

Overall Perspective. One obvious partial drawback to all the results mentioned in this section (and elsewhere in this article) is that all our declared runtimes are obviously *at least linear* in the size of the database, in that they are asymptotes of the form: "$O(I \operatorname{Log}^d I + U)$". Several of our prior articles, most notably [23, 59, 61, 67–69], did illustrate search algorithms with sublinear times with magnitudes "$O(\operatorname{Log}^d I + U)$" or better, that were available when one had access to some preprocessed index data structure. There are two reasons that we do not discuss this topic here. The first is simply that in [67] we already summarized our contributions to this subject. The second is that a search algorithm which requires access to a precomputed index data structure is obviously a *very mixed blessing* because of the extremely non-trivial overhead that is often required to maintain these indices.

Thus, there naturally arises a question about "*What types of complicated database queries can run in $O(I \operatorname{Log}^d I + U)$ time when all types of precomputed index data*

[5] The graph of q_1 is not a tree because it has arcs from r_3 to r_1, r_3 to r_2, 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.

structures are made unavailable?" Our study of RCS optimization is intended to provide a partial answer to this question.

5. RCS OPTIMIZATION METHODOLOGY

The previous sections used the notation below to denote a relational query:

$$\{\text{FIND}(r_1 r_2 \ldots r_p) \in R_1 \times R_2 \times \cdots \times R_p$$
$$Q_{p+1}(r_{p+1} \in R_{p+1}) Q_{p+2}(r_{p+2} \in R_{p+2}) \ldots Q_k(r_k \in R_k) : e(r_1 r_2 \ldots r_k)\}. \tag{31}$$

In this section, we will use a more abbreviated notation where the capital letter R_i is always omitted (i.e. it will be implicitly assumed that $r_i \in R_i$). Thus the following equation will be an abbreviation for (31):

$$\{\text{FIND}(r_1 r_2 \ldots r_p) Q_{p+1}(r_{p+1}) Q_{p+2}(r_{p+2}) \ldots Q_k(r_k) : e(r_1 r_2 \ldots r_k)\}. \tag{32}$$

5.1. QL-Reductions

The formal definition of a QL-reduction was given in Definition 4.1. The two main theorems that we will need about QL-reductions are listed below:

THEOREM 5.1. *For each* $0 \leqslant j' < j$, *every RCS query with* j *quantifiers can be* QL-*reduced to a query with* j' *quantifiers. The WH-time needed to perform this* QL-*reduction is* $O(I \operatorname{Log}^d I)$ *for a constant* d *that depends on the particular query.*

THEOREM 5.2. *Every RCS query* q *whose* FIND-*clause has only one or two variables can be executed in time* $O(U + I \operatorname{Log}^d I)$ *and space* $O(U + I)$ *for some constant* d *that depends on* q.

We will need one preliminary lemma to help prove Theorems 5.1 and 5.2.

LEMMA 5.1. *For each* $j > 0$, *every RCS query with* j *quantifiers can be* QL-*reduced to a query with* $j - 1$ *quantifiers.*

Proof. Let Eq. (33) denote the initial RCS query:

$$\{(\text{FIND } (r_1 r_2 \ldots r_i) Q_{i+1}(r_{i+1}) \ldots Q_{i+j}(r_{i+j}) : e(r_1 r_2 \ldots r_{i+j})\}. \tag{33}$$

Let r_f denote the parent of r_{i+j} in the relational graph of Eq. (33) (if r_{i+j} has a parent), and let $r_f = r_1$ otherwise. Let $e_1^A e_2^A \ldots e_k^A$ and $e_1^B e_2^B \ldots e_k^B$ denote a series of predicates such that:

(i) None of the atomic formula in $e_1^A e_2^A \ldots e_k^A$ will employ the variable r_{i+j}. Also, these formulae are "disjoint". For each tuple $(\bar{r}_1, \bar{r}_2 \ldots \bar{r}_{i+j-1}) \in R_1 \times R_2 \times \ldots R_{i+j-1}$, this means that there is no more than one predicate e_n^A where

$e_n^A(\bar{r}_1, \bar{r}_2 \ldots \bar{r}_{i+j-1})$'s Boolean value equals TRUE.

(ii) Only the variables r_{i+j} and r_f appear in the atoms of $e_1^B e_2^B \ldots e_k^B$.

(iii) These predicates satisfy the condition

$$e(r_1, \ldots, r_{i+j}) = \{[e_1^A(r_1, r_2 \ldots r_{i+j-1}) \wedge e_1^B(r_f, r_{i+j})] \vee \cdots \vee [e_k^A(r_1, r_2 \ldots r_{i+j-1}) \wedge e_k^B(r_f, r_{i+j})]\}. \qquad (34)$$

Let L_m denote the subset of R_f produced by the following binary query:

$$L_m = \{\text{FIND } (r_f)\ Q_{i+j}(r_{i+j}) : e_m^B(r_f, r_{i+j})\}. \qquad (35)$$

It is then evident that a QL-reduction which removes one of the quantifiers from (33) but produces an equivalent output is

$$\text{FIND } (r_1 \ldots r_i) Q_{i+1}(r_{i+1}) \ldots Q_{i+j-1}(r_{i+j-1}): \{(e_1^A \wedge r_j \in L_1) \vee \cdots \vee (e_k^A \wedge r_f \in L_k)\}. \quad ▮ \qquad (36)$$

We will now apply Lemma 5.1 to prove Theorems 5.1 and 5.2. Both these proofs are direct consequences of Lemma 5.1, and they are given below:

Proof of Theorem 5.1. Immediate, because if we simply apply Lemma 5.1's algorithm inductively for $j - j'$ iterations then the result will be to QL-reduce the original RCS query into an alternate form with $j - j'$ quantifiers removed. ▮

Remark 5.1. The above 1-sentence proof of Theorem 5.1 is obviously the simplest possible justification for this theorem. However, it is far from the best method for implementing the QL-reductions in a pragmatic environment. Its disadvantage is that each iteration of Lemma 5.1's algorithm will cause the coefficient associated with the QL-reduction procedure's $O(I \operatorname{Log}^d I)$ time-asymptote to increase by at least some constant factor. For many RCS queries q, there will be available more sophisticated methods to execute the QL-reductions so that the constant factor hidden in the O-notation's asymptote $O(I \operatorname{Log}^d I)$ can be better controlled. This is one example of many topics that were omitted from this paper because our intention is to provide only *an intuitive and theoretical feel* for the RCS theory.

Proof of Theorem 5.2. Let k denote the number of variables appearing in the query q, and let us apply Theorem 5.1 with $j' = k - 2$. In this case, Theorem 5.1 implies we can QL-reduce q to a query with two variables. If this final query is of the form "FIND$(x, y)e(x, y)$" then we can resolve it by applying an E-8 Reporting Join operation. On the other hand if for a quantifier Q, it is of the form "FIND(x) $Q(y) : e(x, y)$", then we can resolve it by similarly applying one of Lemma 4.1's binary algorithms (for evaluating a quantifier). ∎

5.2. RCS Queries with K Variables in the FIND-Clause

This section will generalize Theorem 5.2 so that it can also apply to RCS queries having three or more variables in its FIND-clause. In particular, our objective will be to prove the following result:

THEOREM 5.3. *Let e be any predicate expression built out of equality, order, list and tabular atoms concatenated in arbitrary manner by AND, OR and NOT connectives. Let q designate* (37)'*s relational calculus expression. If q satisfies the RCS condition, then for some constant d, it is possible to find the tuples satisfying* (37) *in time $O(U + I \operatorname{Log}^d I)$ and space $O(U + I)$*:

$$\{\text{FIND}(r_1 r_2 \ldots r_i) Q_{i+1}(r_{i+1}) Q_{i+2}(r_{i+2}) \ldots Q_k(r_k) : e(r_1 r_2 \ldots r_k)\}. \tag{37}$$

Some added notation will be needed to help prove Theorem 5.3. Define an atomic literal to be an expression that is either an atomic formula or the negation of an atomic formula. Define a *pure conjunction* to be a conjunction of several atomic literals. An example of a pure conjunction is

$$\{A_1(r_1) \neq A_2(r_2) \wedge A_2(r_2) < A_3(r_3) \wedge (r_2, r_3) \in T\}. \tag{38}$$

We will need the following preliminary lemma to help prove Theorem 5.3.

LEMMA 5.2. *Suppose query* (39) *satisfies the RCS condition, and its predicate e is a pure conjunction. Then for some constant d, it is possible to find the tuples satisfying* (39) *in time $O(U + I \operatorname{Log}^d I)$ and space $O(U + I)$*:

$$\{\text{FIND}(r_1 r_2 \ldots r_k) : e(r_1 r_2 \ldots r_k)\}. \tag{39}$$

Proof. Let S denote the set of tuples satisfying (39). For simplicity, let us assume that all the attributes of the relations $R_1 R_2 \ldots R_k$ are distinct. For any $i<j$, let $S(i,j)$ denote the projection of S onto the attributes of the R_i and R_j relations: In the relational calculus notation, this simply means that $S(i,j)$ is the set of ordered pairs satisfying

$$S(i,j) = \{\text{FIND}(r_i, r_j) \exists r_1 \exists r_2 \ldots \exists r_{i-1}$$
$$\exists r_{i+1} \ldots \exists r_{j-1} \exists r_{j+1} \ldots \exists r_k : e(r_1 r_2 \ldots r_k)\}. \tag{40}$$

The query in (40) does not necessarily satisfy the RCS condition, and therefore we cannot directly use Theorem 5.2 to produce the set of tuples satisfying this equation. One further definition is necessary to remedy this problem.

Let $f(j)$ be that integer i such that r_i is the parent of r_j if r_j indeed has a parent in (39)'s relational graph, and $f(j) = 1$ otherwise. Then the relational calculus expression on the right-hand side of (40) essentially satisfies the RCS condition when $i = f(j)$ (see footnote[6] for the meaning of the phrase "essentially satisfies").

Let U_j denote the cardinality of $S(f(j),j)$, and U denote the cardinality of (39)'s output. Theorem 5.2's algorithm can construct $S(f(j),j)$, in $O(U_j + I \operatorname{Log}^d I)$. WH-time and using $O(I + U_j)$ space. The first step of our algorithm will use this procedure to construct all the sets $S(f(j),j)$ for $2 \leqslant j \leqslant k$. Since (40) implies $U_j < U$, these time and space costs, in fact, reduce to $O(U + I \operatorname{Log}^d I)$ and $O(I + U)$.

The second step of our procedure will take the "natural join" [55] of all these $S(f(j),j)$ relations to construct the relation S. It is immediate from our definitions that S is in fact equal to the natural join of these relations, but it is not immediately obvious that we can calculate their natural join within the quasi-linear time–space claimed by Lemma 5.2. Our proof of the latter fact is partially related to the theory of acyclic queries [2, 8, 9, 25, 26, 49, 52, 55, 70, 72].

The problem here is best understood if one considers the cost of taking the natural join of k relations $S_1 S_2 \ldots S_k$ by using the following 2-step procedure:

```
1. Set $T_2 = S_1 \bowtie S_2$.
2. FOR $i = 3$ TO $k$,    DO $T_i = T_{i-1} \bowtie S_i$.
```

This procedure will be called a chained natural join. It is easy to see that it will consume an amount of time and space proportional to the sum of the cardinalities of all the relations $S_1 S_2 \ldots S_k$ and $T_2 T_3 \ldots T_k$ (if we use hashing to run the joins in essentially a very straightforward manner). However, this chaining procedure *cannot* be presumed to have a quasi-linear cost. The difficulty is that $S_1 S_2 \ldots S_k$ are its input relations, but T_k is its *sole* output relation! Thus, the time/space complexity of a chained natural join will exceed the desired $O(I + U)$ bound if one of the intermediate relations $T_2 T_3 \ldots T_{k-1}$ has a cardinality much larger than the sum of the cardinalities of T_k and $S_1 S_2 \ldots S_k$.

To ascertain that a chained natural join procedure has an $O(I + U)$ complexity, one must therefore verify that each of its intermediately calculated relations $T_2 T_3 \ldots T_{k-1}$ are sets with cardinality no larger than say T_k. For general natural joins it is impossible to obtain this well bounding condition (see for example the discussion of cyclic queries in one of [2, 49, 55]). However, our interests will focus on *the specific problem* of taking the natural join of the $S(f(j),j)$ where $2 \leqslant j \leqslant k$. We will see how the intermediate sets $T_2 T_3 \ldots T_{k-1}$ have *well-managed sizes* in this particular case. Thus, consider the following 3-step procedure:

```
(A) Set $T_2 = S(1,2)$ (essentially by just renaming the latter set).
(B) For $j = 3$ TO $k$, set $T_j = T_{j-1} \bowtie S(f(j),j)$.
(C) Output the relation $T_k$ as the answer to (39)'s query.
```

[6] The relational graphs of some of the $S(f(j),j)$ queries may not technically satisfy the RCS condition because some of their directed edges could be pointing in the wrong direction. We can remedy this problem by rewriting the query $S(f(j),j)$ in an alternate form where the order in which the variables are existentially quantified is permuted. It is well known that permutations of existential quantifiers do not change the set of elements specified in a set-theoretic query similar to Eq. (40). Such permutations can transform Eq. (40)'s possibly non-RCS query into an obviously equivalent RCS expression.

As noted already, this procedure's cost is proportional to the sum of the cardinalities of the relations $T_2T_3 \dots T_k$ and $S(f(2),2), S(f(3),3) \dots S(f(k),k)$. We will use Fact $*$ to determine the sizes of the tables $T_2T_3 \dots T_k$.

FACT $*$. *The combination of the facts that e is a pure conjunction, Eq.* (39) *is an RCS query, and the fact that each* $S(f(j),j)$ *satisfies Eq.* (40) *implies each of the sets* $T_2, T_3, T_4 \dots T_{k-1}$ *have a cardinality no greater than the particular integer "U"* (*which in our notation designates the cardinality of the final output set* T_k).

The proof of Fact $*$ appears in Appendix A: It is related to theory of acyclic queries of Refs. [2, 8, 9, 25, 26, 49, 52, 55, 70, 72]. Let us now explain its significance. It implies that our natural join algorithm will run in a time no worse than $2k \cdot U$, since its running time is proportional to the combined cardinalities of all the sets $T_2, T_3 \dots T_k$ and $S(f(2),2)$, $S(f(3),3) \dots S(f(k),k)$. Moreover, since k is a fixed constant *that depends only on the number of variables* appearing in query (39), our notation allows us to view the quantity $2k \cdot U$ as an asymptote of the form $O(U)$, where $2k$ is a coefficient lying inside the O-notation.

In summary, our algorithm for answering query (39) is a 2-step procedure, whose first step constructs the $S(f(j),j)$ sets and whose second step applies $k-2$ iterations of the natural join algorithm to construct the sets $T_3, T_4 \dots T_k$. The fourth paragraph of this proof showed that the first step ran in time $O(U + I \operatorname{Log}^d I)$ and space $O(I+U)$, and the last paragraph indicated that $O(I+U)$ also bounds the second step's time/space costs. Hence, our algorithm is quasi-linear. ∎

We will next turn our attention to Theorem 5.3, whose formal statement was given at the beginning of this section. Theorem 5.3 is substantially more general than Lemma 5.2 because it allows the relational calculus expression to contain any sequence of quantifiers, and it does not require e to be a pure conjunction. Its only caveat is that it requires that the RCS Graph condition be satisfied.

Proof of Theorem 5.3. The first step of Theorem 5.3's search algorithm will use Theorem 5.1. The latter implies there exists a predicate e^* such that (41) is a QL-reduction of (37):

$$\{\text{FIND}(r_1r_2 \dots r_i) : e^*(r_1r_2 \dots r_i)\}. \tag{41}$$

The expensive part of this step consists of building the new subsection lists $L_1L_2 \dots L_m$ to effect e's translation into the "output-equivalent" form e^*. By Theorem 5.1, this translation process will consume $O(I \operatorname{Log}^d I)$ time and $O(I)$ space.

Our algorithm's second step will use a set of pure conjunction predicates $e_1e_2 \dots e_k$ such that

$$e^*(r_1r_2 \dots r_i) = \{e_1(r_1r_2 \dots r_i) \vee e_2(r_1r_2 \dots r_i) \vee \dots e_k(r_1r_2 \dots r_i)\}, \tag{42}$$

For each such predicate e_j, this step will employ Lemma 5.2's algorithm to find the set of tuples S_j satisfying $\{\text{FIND}(r_1r_2 \dots r_i) : e_j(r_1r_2 \dots r_i)\}$. Each such subroutine call shall consume time $O(U + I \operatorname{Log}^d I)$ and space $O(U+I)$. There are only a fixed number of such subroutine calls (where the constant again depends on q); thus, this second step also has a quasi-linear cost. It provides the answer to the query q because the union of the sets $S_1, S_2 \ldots S_k$ is the query's answer. ∎

We close this section by again reminding the reader of the observations made by Remark 5.1. Our discussion throughout this chapter has deliberately been kept as brief and as simple as possible so that a reader can most easily appreciate the gist. In order to keep the discussion brief, we have deliberately omitted at several junctures examining methods that will often reduce the coefficient belonging to the O-notation's $O(U + I \operatorname{Log}^d I)$ time asymptote, as well as omitted studying possible methods that will often (but not always) reduce the value of the exponent d.

Such issues are clearly important from a pragmatic perspective, but they can be studied by many researchers in the future. Instead, we have sought to give Theorem 5.3 the *simplest possible* proof in this paper, in order to attract the widest possible audience for this general topic.

5.3. RCS Aggregation Queries

During the last few years, several articles about OLAP-like environments [10, 16, 17, 24, 28–32, 40, 41, 45, 50, 51, 53, 54, 73, 75, 76] have studied aggregation queries. We will show in this section how Theorem 5.1's QL-reduction method can assist in such calculations. The notion of an RCS aggregation was defined in Section 2.2. Thus, Eq. (43) is a request to calculate a table that has an entry for each input element $\bar{x} \in X$, indicating the $\sum f(\bar{y})$ (under some aggregate-operator "$\oplus$") of those ordered pairs $(\bar{x}, \bar{y}) \in X \times Y$ that (43) seeks to output:

$$\{\text{ListAgg}^f(x, y) \in X \times Y \;\; Q_1(r_1)Q_2(r_2)\ldots Q_k(r_k) : e(x, y, r_1, r_2, \ldots r_k)\}. \tag{43}$$

THEOREM 5.4. *Let $\oplus$ denote an $O(1)$ time aggregation operator that admits an inverse, and q denote an RCS aggregation query, similar to Eq.* (43). *Each such RCS aggregation query can be executed within the resource parameters of $O(I \operatorname{Log}^d I)$ WH-time and $O(I)$ space* (*where the value of d again depends on the particular RCS query q*).

Proof. The relevant $O(I \operatorname{Log}^d I)$ procedure is an easy consequence of Theorem 5.1's methodology. It consists of the following two steps.

1. First spend $O(I \operatorname{Log}^d I)$ time constructing the needed lists $L_1, L_2, \ldots L_k$ so that after these lists are constructed, we have available an E-8 enactment expression $e^*(x, y)$, relying on $L_1, L_2, \ldots L_k$ as its inputs, such that the set of ordered pairs satisfying $e^*(x, y)$ is the same as the output for the query:

$$\{\text{FIND}(x, y) \in X \times Y \;\; Q_1(r_1)Q_2(r_2)\ldots Q_k(r_k) : e(x, y, r_1, r_2, \ldots, r_k)\}. \tag{44}$$

Theorem 5.1 indicates that such a collection of subsection lists $L_1, L_2, \ldots L_k$ exists and can be constructed in $O(I \operatorname{Log}^d I)$ WH-time. Note that this step *does not physically perform* either $e^*(x, y)$'s reporting join operation or (44)'s formal search (because either of these operations can require in excess of $O(I \operatorname{Log}^d I)$ time). Rather, it *merely constructs* the collection of subsection lists $L_1, L_2, \ldots L_k$.

2. Finish (43)'s query by using an E-8 Aggregation-Join procedure to construct the array $\Phi^{f}_{e*}(x)$. (The existence of an "E-8 Aggregation Join" for executing this task in $O(I \operatorname{Log}^{d} I)$ time was indicated by Item (B) from Section 2.1.) ∎

Remark 5.2. Let AVERAGE denote the conventional notion of an arithmetic mean. This construct is not an "invertible" operator, but we can still apply Theorem 5.4 towards its calculation. Such an application would first use Theorem 5.4's algorithm to calculate the arrays for SUM and COUNT, and it would then divide these arrays to construct the AVERAGE array.

Remark 5.3. Define $\oplus$ to be a *SemiGroup* operator iff it satisfies the associative principle. (Semigroup operators do not necessarily admit an inverse, and they have been studied extensively in computational geometry [6, 14, 21, 22, 44, 56, 63, 69].) Theorem 5.4's algorithm generalizes to semigroup operators automatically *whenever* the E-8 enactment $e^{*}(x,y)$ produced by its Step 1 contains no tabular atoms. (If tabular atoms are present in $e^{*}(x,y)$, this generalization applies *only when* these atoms do not lie within the range of a NOT connective symbol.)

Let us next consider a query seeking to find the list of Median, Minimal or Maximal elements requested by an RCS formula. The first query type falls under neither the paradigms of Theorem 5.4 nor Remark 5.3, since Median is neither an invertible nor a semigroup aggregate operator. The operations of MIN and MAX are semigroup operators; however, one would ideally desire them not to rely on Remark 5.3's procedure because it is much more inherently restrictive than Theorem 5.4's procedure. It turns out that we can handle these three aggregation operations with the full desired level of generality. The remainder of this section will discuss this topic and other similar forms of aggregation queries.

Notation. Let F denote a sorted list itemizing the values of $f(\bar{y})$ for $\bar{y} \in Y$. Let A, I and P denote functions (or equivalently arrays) that map each $\bar{x} \in X$ onto numbers, which we shall denote as $A(\bar{x})$, $I(\bar{x})$ and $P(\bar{x})$. We will often replace the header element "$\text{ListAgg}^{f}(x,y)$" from Eq. (43)'s RCS query with alternate headers such as "List $\text{Count}(x,y)$", "$\text{SubCount}^{A}(x,y)$", "$\text{Check}^{A,P}(x,y)$" or "$\text{Pick}^{P}(x,y)$". Their definitions are given below:

1. An RCS query that begins with the header $\text{ListCount}(x,y)$ will denote a "$\text{ListAgg}(x,y)$" query, where the aggregation operation is "Count". Thus, such a query will construct an array $I(x)$, where for each $\bar{x} \in X$ the array-element $I(\bar{x})$ will indicate how many $\bar{y} \in Y$ satisfy the condition to the right of Eq. (45)'s ListCount header:

$$\{\text{ListCount}(x,y) \quad Q_1(r_1)Q_2(r_2)\ldots Q_k(r_k) : e(x,y,r_1,r_2,\ldots,r_k)\}. \tag{45}$$

2. $\text{SubCount}^{A}(x,y)$ is defined so that Equation (46) is just an abbreviation for Equation (47).

$$\{\text{SubCount}^{A}(x,y) \quad Q_1(r_1)Q_2(r_2)\ldots Q_k(r_k) : e(x,y,r_1,r_2,\ldots r_k)\}, \tag{46}$$

$$\{\text{ListCount}(x,y) \quad Q_1(r_1)Q_2(r_2)\ldots Q_k(r_k) : e(x,y,r_1,r_2,\ldots r_k) \wedge f(y) \leqslant A(x)\}. \tag{47}$$

Theorem 5.4 implies that the above queries can be executed with quasi-linear efficiency (because if we view the tuples $\bar{x}$ and $\bar{y}$ as containing one extra field each, denoted as, respectively, "$A(\bar{x})$" and "$f(\bar{y})$", then query (47) falls within Theorem 5.4's paradigm).

3. Let $I(x)$ denote the "SubCount^A Array" that is produced by Item 2's procedure (above). The header Check$^A(x,y)$ will indicate the presence of a 2-step procedure that first constructs this subcount array $I(x)$, and then for each $\bar{x} \in X$ compares the values of $I(\bar{x})$ with $P(\bar{x})$. This procedure will store the results of these multiple comparisons in an output array called "$T(x)$". Thus, $T(\bar{x})$ will correspond to one of the three key-words of "less-than," "equals" or "greater-than," depending on how $I(\bar{x})$ compares to $P(\bar{x})$.

4. In order to define (48)'s Pick$^P(x,y)$ command, let us assume P denotes an array of integers and each element $y \in Y$ has *an unique* $f(y)$ value:

$$\{\text{Pick}^P(x,y) \quad Q_1(r_1)Q_2(r_2)\ldots Q_k(r_k) : e(x,y,r_1,r_2,\ldots r_k)\}. \tag{48}$$

This command will be defined as an operation that constructs an array A where $A(\bar{x})$ is the particular element in the sorted list F, such that precisely $P(\bar{x})$ distinct $\bar{y} \in Y$ satisfy (49). Footnote[7] explains the significance of this definition through three examples, illustrating how we can make the array $A(x)$ correspond to a list of minimum, maximum or median elements, by applying various different forms of Pick$^P(x,y)$ operations:

$$Q_1(r_1)Q_2(r_2)\ldots Q_k(r_k) : e(\bar{x},\bar{y},r_1,r_2,\ldots r_k) \wedge f(\bar{y}) \leqslant A(\bar{x}). \tag{49}$$

THEOREM 5.5. *Suppose the* Pick$^P(x,y)$ *query in* (48) *satisfies Section* 2.2*'s "RCS graph condition." Then there will exist some constant d such that this query can be processed in* $O(I \operatorname{Log}^d I)$ *WH-time and using* $O(I)$ *space.*

Proof. Let N_y again denote the cardinality of table Y. The heart of our Pick$^P(x,y)$ search algorithm will consist of making $\text{Log}_2(N_y)$ subroutine calls to Item 3's Check$^{A,P}(x,y)$ procedure. In essence, these $\text{Log}_2(N_y)$ subroutine calls will enable us to formulate a straightforward generalization of a conventional binary search, where each iteration allows us to more closely approximate the final form of the particular array A that our Pick$^P(x,y)$ query seeks to construct.

More precisely, let m denote F's median $f(y)$-value. The array A will be initially set so that $A(\bar{x}) = m$ for each $\bar{x} \in X$. Our first invocation of Check$^{A,P}(x,y)$ will use this initial state for A to generate an output array T, where $T(\bar{x})$ specifies one of the three states of "less-than", "equals" or "greater-than" (depending on how $I(\bar{x})$ compares to $P(\bar{x})$). Then if m_1 and m_2 denote the respective elements of F that have 25% and 75% of members of F lying below them, our algorithm will rewrite the array A so that $A(\bar{x})$ will now equal one of the three values of m_1, m or m_2, depending on whether $T(\bar{x})$ had stored a state of "less-than", "equals" or "greater-than". Our

[7] If P is an array *whose every element* is the integer 1 then $A(\bar{x})$ will represent the minimal value $f(\bar{y})$ for the set of ordered pairs $(\bar{x},\bar{y})$ satisfying the condition *to the right* of Eq. (48)'s "Pick$^P(x,y)$" header. It will likewise be the corresponding maximal value if P represents the array which is the output of Eq. (45)'s ListCount query. On the other hand, if we let P denote the arithmetic mean of these two arrays, then "Pick$^P(x,y)$" will cause the output array A to be Eq. (48)'s list of implied median elements.

algorithm's second and further invocations of the subroutine $\text{Check}^{A,P}(x, y)$ will be analogous to the first, except that they will use the array A's successively revised states.

In other words, our algorithm for constructing A will be roughly the same as the conventional binary search, except that it runs all the binary searches *simultaneously* for every $\bar{x} \in X$; thus, at the conclusion of its $\text{Log}_2(N_y)$ subroutine calls to $\text{Check}^{A,P}(x, y)$, our algorithm will have constructed all the values for $A(\bar{x})$. The formal description of this analog of binary search will not appear here because it is an extremely straightforward consequence of the definitions of $\text{Check}^{A,P}(x, y)$ and of $\text{Pick}^{P}(x, y)$. Since $N_y \leqslant I$ and since Theorem 5.4 implied $\text{Check}^{A,P}(x, y)$ had a quasi-linear efficiency, it follows that $\text{Pick}^{P}(x, y)$ must also have a quasi-linear complexity, with its exponent d differing from $\text{Check}^{A,P}(x, y)$'s exponent by exactly an increment of 1. ∎

6. SIGNIFICANCE OF RESULTS

The preceding discussion was deliberately written in a style to make our proofs as short and simple as possible. To shorten the proofs considerably, we have often produced versions of our algorithm that had a needlessly large coefficient. Any other style of presentation would have been inappropriate for an article attempting to present briefly the simplest possible overview for this subject.

However, because, for the sake of brevity, our presentation sacrificed the coefficient inside the $O(I \,\text{Log}^d I + U)$ and $O(I)$ asymptotic magnitudes, we request that the reader not prejudge the RCS algorithm based on the particular version of the procedure presented in these pages. We do not claim that the coefficient hidden inside the O-notation will always be small, even when one does try to minimize the coefficient. However, it should have an adequately small magnitude for our algorithm to be worthy of consideration in several settings.

The growth in main memory size is the main reason the RCS formalism is tempting. The first sentence of this article did not exaggerate in noting that the size of the main memory has grown by a factor of more than 10,000 since the time, 20 years ago, when the potential of RCS was mentioned in our dissertation [59]. One way to illustrate this change is simply to take Moore's Law and count the number of doublings that would take place in 20 years at a rate of one doubling per 18 months (i.e., $2^{13.5} > 10{,}000$). A second method to gather a roughly similar estimate is to observe that since their inception 23 years ago, the memory spaces of Personal Computers have actually grown by a somewhat larger 32,000-to-1 ratio.[8] Moreover, the recent "1998 Ansilamar Report" seems to agree with our interest in databases resident in main memory. It [7] predicts that:

"*Within ten years, it will be common to have a terabyte of main memory serving as a buffer pool for a hundred terabyte database. All but the largest database tables will be resident in main memory.*"

[8] The original Altair machine had a 4K memory, which differs by a 32,000-to-1 ratio from the 128-Meg memories becoming the standard size for Personal Computers in several stores we visited while preparing the final draft of this paper. In particular, on 16 December, 2000, one store manager informed us that 80% of his Gateway computer sales involved at least 128-Meg size machines, and no computers were now available below a 64-Meg size.

In all these contexts put together, it would seem safe to suggest that some forms of $O(I \operatorname{Log}^d I + U)$ algorithms will be cost-effective for certainly some databases resident in the main memory.

Our work related to RCS had thus influenced Goyal and Paige, who were generous enough to mention our name in the title of one of their articles [27]. It described an implementation of a portion of Theorem 5.3, based essentially on the combination of our prior work [59, 65, 67], private communications from Willard and some of Paige *et al.*'s earlier work [11, 27, 33, 46–48].

One implementation of at least a portion of the RCS method is thus already available at an experimental level; moreover, some aspects of the RCS formalism are likely to have implications for database design, even if *the full desired level of a commercial-grade* software product never emerges. It is, after all, reasonable to view database theory from *roughly a RISC-like perspective*. That is, suppose one implemented only the E-8 reporting and aggregate JOIN procedures from our earlier 1996 article [67], using a RISC-type philosophy, where a *small number of primitive* operations should be the focus of an *extremely intense* and dedicated effort to implement them with *maximum efficiency*. (In the SQL language, such an implementation could correspond to a strongly optimized procedure for executing those SELECT-FROM-WHERE queries that have only two tables appearing in the FROM-clause and whose WHERE-clause corresponds to essentially an E-8 enactment (see footnote[9] for how one can model an E-8 enactment's tabular atoms in the SQL language). It would then be plausible to ask the database user to perform more complicated k-variable relational-calculus-like operations by having the computer programmer literally manually chain together several subroutine calls to a library of very efficient E-8 enactment operations.

In other words, we are suggesting that one can interpret Theorem 5.3 as having either of two uses. One possibility is to see it as describing formal operations available to a computer optimizer. An alternate interpretation is to view it as summarizing how a human computer programmer can hand-optimize his code when using a library of computer programs for doing E-8 operations efficiently.

6.1. *What Actually is an RCS Query?*

There are also other issues, pertaining to the potential implications of RCS, that we should vent at least briefly. Although Section 2.2 may have appeared to have given a fully succinct and explicit 1-paragraph definition of the RCS language, it actually contained some non-trivial levels of ambiguity hidden within it. After all, while many relational calculus queries may technically violate the RCS acyclic query property, they are often still "output-equivalent" to other queries that are acyclic. For instance, Example 4.3 *illustrated* how—when one uses a tabular atom to replace Eq. (29)'s existential quantifiers—the non-RCS query in (29) is output-equivalent to (30)'s RCS query.

[9] The "EXISTS" or "IN" primitives, when embedded inside a WHERE-clause, can be used by SQL to signal the presence of a Tabular atom. An alternative for the SQL language would obviously be to introduce a new primitive into the language for explicitly representing Tabular atoms. We suspect that the second approach is preferable, but either could be used.

There are also many other types of examples of pairs of output-equivalent relational calculus queries with a similar property. For instance, let X and Y denote two relations. Consider the following query:

Find all $\bar{x} \in X$ where there are at least two different elements $\bar{y}_1$ and $\bar{y}_2$ in Y satisfying $e(x, y)$.

Both Remark 4.2 of this present paper and Remark 4 of our earlier paper [65] anticipated this type of query by including the notion of a "generalized quantifier" in the RCS language. We defined a "generalized quantifier" as any function that mapped values from the count-array $\Phi_e(x)$ onto Boolean values. Thus, a valid example of a generalized quantifier could be a quantifier Q that returns the value TRUE when $\Phi_e(x) \geqslant 2$. Note that this means that (50) and (51) are two alternate formalisms for representing the italicized sentence as a relational query:

$$\text{FIND } x \in X \quad \text{COUNT}(y \in Y) \geqslant 2 : e(x, y), \tag{50}$$

$$\text{FIND } x \in X \quad \exists y_1 \in Y \quad \exists y_2 \in Y : y_1 \neq y_2 \wedge e(x, y_1) \wedge e(x, y_2). \tag{51}$$

The point is that these two queries have identical output, although technically only the former actually belonged to the RCS class.

A third example of a pair of output-equivalent queries appears below:

$$\text{FIND } x \in X \quad \exists y \in Y \quad \exists z \in Z : [A_1(x) < B_1(y) \wedge B_2(y) < C(z)] \vee$$
$$[A_1(x) \geqslant B_1(y) \wedge A_2(x) < C(z)], \tag{52}$$

$$\text{FIND } x \in X \quad \exists y_1 \in Y \quad \exists z_1 \in Z \quad \exists y_2 \in Y \quad \exists z_2 \in Z :$$
$$[A_1(x) < B_1(y_1) \wedge B_2(y_1) < C(z_1)] \vee [A_1(x) \geqslant B_1(y_2) \wedge A_2(x) < C(z_2)]. \tag{53}$$

This example has almost the precise opposite quality to the example from the previous paragraph. Their difference is that the former example had the RCS query possess fewer quantifiers than its non-RCS counterpart, while the latter example has the RCS query containing more quantifiers.

A fourth class of similar examples arises because one can permute the order of two consecutive existential (or universal) quantifiers without changing the meaning of the relational calculus query. (It is also possible to permute the order of two variables in the FIND clause.) In some cases, such permutations will transform a non-RCS query into an RCS query (because such permutations reverse the directions of some edges in our query's corresponding graph). Moreover, one can often apply several algebraic identities to transform non-RCS queries into output-equivalent RCS operations.

In closing this subsection, we wish to point out that it is not picayune to examine methods for transforming non-RCS queries into output-equivalent RCS forms. We do so because we believe that database researchers are most likely to ask themselves the following question:

What are the linguistic limitations of an RCS-like database query language? That is, what types of natural database queries cannot be expressed in this formalism?

Our point is that the answer to this question is complicated and partly ambiguous because each of Eqs. (29), (51) and (52) illustrate examples of non-RCS queries, which, after a *trivial transformation*, have the same output as an RCS search.

How many more examples of this type are there? We do not know: The most general version of the problem of translating non-RCS queries into output-equivalent RCS operations is clearly NP-hard. Moreover, there certainly exists several database queries that *lie properly outside* the RCS class, but can be executed efficiently. For instance, the Papadimitriou–Yannakakis article [49] showed how certain types of cyclic inequality join-project-selections can be processed efficiently, and we will illustrate a different type of such example in Section 6.3. The future will certainly discover many more such examples.

6.2. More About Tabular Predicates

Our earlier Example 4.3 explained that one reason we introduced the Tabular Predicate notion into the RCS language was to broaden considerably this database language. Thus, Example 4.3 showed how Eq. (29) technically was not an RCS query, but that the tabular atoms allowed us to rewrite it in an alternate form that was RCS.

This broadening of the RCS language is obviously a nice feature. However, there is also a second nice aspect of the tabular predicates that might be overlooked if we were not to mention it explicitly. It will require some additional notation.

Let N_x and N_y again denote the cardinality of the relations X and Y. Let N_t denote the cardinality of a tabular section T which itemizes some ordered pairs (x, y) from the cross-product space $X \times Y$. In theory, N_t could obviously be as large as the quantity $N_x \cdot N_y$. However, it is well known that in many database settings, it will satisfy either $N_t < O(N_x + N_y)$, or at least

$$N_t \ll N_x \cdot N_y. \tag{54}$$

We will use the term *Sparse Table* to describe a tabular section T whose set of ordered pairs satisfies an identity similar to Eq. (54).

One can appreciate the very central nature of sparse tables in database applications by looking as far back as the early Codasyl literature. The notions of a 1-1 and many-to-1 relationship stem from the original Codasyl Set-Ownership model: it is well known that if T either represents a 1-1 or many-to-1 relationship then the sparsity inequality below will be satisfied:

$$N_t \leqslant N_x + N_y. \tag{55}$$

The 1-1 and many-to-1 relationships clearly occur very frequently in database applications, since most of the database textbooks give this topic quite prominent mention. Thus because Eq. (54) is often satisfied, it would be prudent for a database optimizer to attempt to use this equation to optimize performance (whenever such an opportunity arises). Moreover, (54)'s inequality is especially significant because there are *many instances* when (54) is satisfied *without* the tabular sections formalizing a 1-1 or many-to-1 relationship (see the second-to-last paragraph of Example 3.3 for several such examples in a typical database environment). Hence, one reason we introduced the Tabular atom formalism into the RCS and E-8 languages, starting in our dissertation [59], was because the widespread implicit usage of sparse tables in database applications made it desirable for an optimizer to take advantage of the presence of sparsity for improving performance, whenever such an opportunity[10] occurs.

Before closing this section, we should also mention that it is possible to view tabular atoms as not exactly representing explicitly formed tables T, *available at the onset* of a database search. Instead, if one so desires (?), one can view the tabular sections T as representing *intermediately constructed objects* that are built midway during a sequence of several serially performed RCS operations.

6.3. More Issues About Database Language Expressibility

For the sake of keeping our mathematical discussion in this article as crisp and abbreviated as possible, we had assumed that the tuple variable r would span over only a *single relation R*. From a mathematical perspective, this assumption was very minor because it is obvious that all our algorithms (and their quasi-linear performance characteristics) will trivially generalize to the broader case where r can range over the union of several relations, such as for example $R_1 \cup R_2 \cup \cdots \cup R_j$. Let us therefore use the acronym E-RCS for a query that is the same as an RCS operation, except that its tuple variables r are allowed to range over the union of several relations, such as for example $R_1 \cup R_2 \cup \cdots \cup R_j$. Also, let the acronym UE-RCS denote a query of the form $q_1 \cup q_2 \cup \cdots \cup q_j$, where each q_i is E-RCS.

It often happens that a distinction may be trivial from a mathematical perspective, but still possess some significance from a systems-programming perspective. This type of issue seems to arise when one compares the RCS, E-RCS and UE-RCS languages. From the mathematical perspective of Algorithm Design, the distinction between these three languages is trivial because they all have the same quasi-linear time complexities (for both the cases of doing a Find or Aggregation search). In contrast, we will see that this distinction is quite important from the viewpoint of database expressibility.

In particular, one reason that the E-RCS primitive is needed is that the unmodified-RCS formalism is unable to express the relational algebraic notion of "Finite Set Union" without E-RCS's added flexibility. One would certainly like a database language to have a capacity to formulate as working operations each of Codd's eight original relational algebra commands (i.e. Union, Intersection, Set-Subtraction, Projection, Division, Selection, Cross-Product and Join). It turns out that E-RCS formalism is fully adequate for this purpose.

[10] Our article [67] explained that the E-8 Reporting and Aggregate Joins had complexities of $O((N_x + N_y)\mathrm{Log}^{d^*(e)} N_y + N_e + N_t)$ and $O((N_x + N_y)\mathrm{Log}^{d^*(e)} N_y + N_t)$. Whenever $N_t \ll N_x \cdot N_y$, these runtimes are clearly much better than the $O(N_x \cdot N_y)$ cost of a brute force exhaustive search.

It also turns out that there are several examples of UE-RCS queries that *actually cannot be written* in an E-RCS form. This curiosity occurs because some UE-RCS queries $q_1 \cup q_2 \cup \cdots \cup q_j$ have the property that if one attempts to compress q_1, q_2 , $\ldots$, q_j into a *single relational calculus* expression, then the resulting graph $G(q)$ will contain such a large number of edges that it would violate Section 2.2's RCS graph condition. Thus, one needs the UE-RCS primitive to signal the fact that these operations can be performed efficiently by breaking them into their subcomponents $q_1, q_2, .., q_j$ and then taking the union of the resultant queries.

To further appreciate the potential as well as inherent limitations of RCS-like languages, it is helpful to return to some of the comments that Papadimitriou and Yannakakis [49] made about database optimization. They noted that the commonly believed conjectures about NP-hardness suggest that (1) a deterioration in performance for some cyclic database queries will be unavoidable, and (2) it will also be impossible to devise a decision procedure that takes a relational calculus query q of length L as input, and determines in time polynomial in L the amount of resources that the query q will optimally require. Thus, the study of quasi-linear database search algorithms is highly likely to be a never-ending quest that never fully reaches a perfect conclusion. At best, it will probably only devise broadly general languages that can process a reasonably large fraction of potential queries with quasi-linear efficiency.

In this context, we can now more clearly explain our basic objectives. Our Example 4.3, Sections 6.1 and 6.2, and our 3-way distinction between RCS, E-RCS and UE-RCS queries had collectively documented that there are a quite large number of likely database queries that have RCS-like quasi-linear complexities. By showing that a database optimizer can perform a very broad class of relational calculus queries with quasi-linear efficiency, we have sought to stimulate further research into this area. For instance, one would like ideally to lower the exponent d and the coefficient hidden inside the asymptote $O(I \operatorname{Log}^d I + U)$ as much as possible, as well as to further extend the class of queries q that can be processed with quasi-linear efficiency. We hope that our research into RCS will thus stimulate other researchers to join our investigation into the many remaining open questions.

APPENDIX A: PROOF OF FACT $*$

This appendix will prove Fact $*$. Since results roughly similar to this claim have analogs in the literature on acyclic joins [2, 8, 9, 25, 26, 49, 52, 55, 70, 72], mentioned in Section 2.3, our proof of Fact $*$ will be short and abbreviated. We will need one lemma to help prove Fact $*$.

LEMMA A.1. *Once again, let us assume that query* (A.1) *satisfies the RCS graph condition and that its predicate* $e(r_1 r_2 \ldots r_k)$ *is a pure conjunction*:

$$\{\text{FIND}(r_1 r_2 \ldots r_k) : e(r_1 r_2 \ldots r_k)\}. \tag{A.1}$$

Let T_k denote the set of k-tuples satisfying (A.1). *Suppose that the sets T_{j-1} and $S(f(j),j)$ (used during the jth iteration in Step B of Lemma* 5.2*'s algorithm) satisfy the two conditions*:

$$(\bar{r}_1, \bar{r}_2, \ldots \bar{r}_{j-1}) \in T_{j-1} \Leftrightarrow \exists r_j, \exists r_{j+1}, \ldots \exists r_k : e(\bar{r}_1, \bar{r}_2, \ldots \bar{r}_{j-1}, r_j, r_{j+1}, \ldots r_k), \qquad \text{(A.2)}$$

$$\begin{aligned}(\bar{r}_{f(j)}, \bar{r}_j) \in S(f(j),j) \Leftrightarrow \exists r_1, \ldots \exists r_{f(j)-1} \exists r_{f(j)+1}, \ldots \exists f_{j-1} \exists r_{j+1}, \ldots \exists r_k :\\ e(r_1 \ldots r_{f(j)-1}, \bar{r}_{f(j)}, r_{f(j)+1} \ldots r_{j-1}, \bar{r}_j, r_{j+1} \ldots r_k). \qquad \text{(A.3)}\end{aligned}$$

Then the set $T_j = T_{j-1} \bowtie S(f(j),j)$ will also satisfy:

$$(\bar{r}_1, \bar{r}_2, \ldots \bar{r}_j) \in T_j \Leftrightarrow \exists r_{j+1}, \exists r_{j+2}, \ldots \exists r_k : e(\bar{r}_1, \bar{r}_2, \ldots \bar{r}_j, r_{j+1}, r_{j+2}, \ldots r_k). \qquad \text{(A.4)}$$

Proof. Let t be a $(j-1)$-tuple $(a_1, a_2, \ldots a_{j-1}) \in T_{j-1}$. Then from (A.2), we can infer the existence of a more elongated k-tuple $(a_1, a_2, \ldots a_k) \in T_k$. Similarly, if (x, y) is an ordered pair in $S(f(j),j)$, we can infer from (A.3) the existence of a k-tuple $(b_1, b_2, \ldots b_k) \in T_k$ with $b_{f(j)} = x$ and $b_j = y$.

By definition, the preceding tuple t and ordered pair (x, y) will generate an element in the "join-set" $T_j = T_{j-1} \bowtie S(f(j),j)$ precisely when $x = a_{f(j)}$. This element will simply be a j-tuple $(c_1, c_2, \ldots c_j)$ where $c_j = y$ and all the other $c_i = a_i$. To prove that (A.4) is satisfied, it suffices to show that this j-tuple $(c_1, c_2, \ldots c_j)$ can be associated with a longer k-tuple $(c_1, c_2, \ldots, c_k) \in T_k$.

To do so, we will use the fact that Eq. (A.1)'s query q satisfies the RCS graph condition. Let $G(q)$ denote the graph associated with q. Let $G^*(q)$ be a graph identical to $G(q)$ except that:

1. If $G(q)$ is a forest-graph containing distinctly more than one tree, then $G^*(q)$ will be a *tree* where for each $i > 1$, there will be a new edge in $G^*(q)$ connecting r_i to r_1 when such an edge is necessary to connect what would *otherwise be two disjoint trees* into *one single* tree.
2. We will view $G^*(q)$ as an undirected graph in the current discussion.[11]

Our objective will be to finish Lemma 6.1's proof by using the graph $G^*(q)$ to construct the tuple k-tuple $(c_1, c_2, \ldots c_k) \in T_k$ which the previous paragraph needed.

This construction is quite simple. Let us say the node r_i *lies on the left-hand side* of the tree-graph $G^*(q)$ if the path from r_i to r_j goes through $r_{f(j)}$. Say it *lies on $G^*(q)$'s right-hand side* otherwise. Let the k-tuple $(c_1, c_2, \ldots c_k)$ have its components c_i defined by the two rules that $c_i = a_i$ if r_i is a "left" node and $c_i = b_i$ if it is "right-sided". Since Lemma 6.1's hypothesis indicated $e(r_1 r_2 \ldots r_k)$ was a pure conjunction, this fact and $(a_1, a_2, \ldots a_k) \in T_k$ and $(b_1, b_2, \ldots b_k) \in T_k$ immediately imply $(c_1, c_2, \ldots, c_k) \in T_k$. Hence, Eq. (A.4) is certainly valid. ∎

[11] In Theorems 5.1, 5.2 and 5.3, we needed the definition of $G(q)$ to view this structure as a *directed graph* (basically to preclude some difficulties that could otherwise be posed by universal quantifiers and generalized quantifiers). These difficulties cannot exist for Eq. (A.2)–(A.4) because they contain only existential quantifiers. Therefore without difficulty, $G^*(q)$ can be viewed as an undirected graph in our present discussion.

We will now use Lemma 6.1 to complete the proof of Fact $*$ (from Lemma 5.2's proof). The formal statement of this claim appeared in Lemma 5.2's proof, and we have duplicated it below:

FACT $*$. *The combination of the facts that e is a pure conjunction,* (A.1) *is an RCS query, and each* $S(f(j),j)$ *set satisfies* (A.5) *implies that each of the sets* $T_2, T_3, T_4 \ldots T_{k-1}$ *have a cardinality no greater than the cardinality of* T_k:

$$S(f(j),j) = \{\text{FIND}(r_{f(j)}, r_j)\exists r_1 \ldots \exists r_{f(j)-1} \\ \exists r_{f(j)+1} \ldots \exists r_{j-1}\exists r_{j+1} \ldots \exists r_k : e(r_1 r_2 \ldots r_k)\}. \tag{A.5}$$

Proof. It will be immediately apparent that the claim is true if we can establish that each of the sets $T_2, T_3, T_4 \ldots T_{k-1}$ satisfies

$$(\bar{r}_1, \bar{r}_2, \ldots \bar{r}_j) \in T_j \Leftrightarrow \exists r_{j+1}, \exists r_{j+2}, \ldots \exists r_k : e(\bar{r}_1, \bar{r}_2, \ldots \bar{r}_j, \bar{r}_{j+1}, r_{j+2}, \ldots r_k). \tag{A.6}$$

We can easily verify this fact by induction. In particular, T_2 must satisfy condition (A.6) simply because step A of Lemma 5.2's join algorithm defined $T_2 = S(1,2)$ and the hypothesis of Fact $*$ had indicated that $S(1,2)$ satisfied Eq. (A.5). The further verification that $T_3 T_4 \ldots T_{k-1}$ satisfy Eq. (A.6) follows by an easy inductive argument that uses Lemma 6.1 and the fact that T_j satisfies (A.6) to conclude that so does T_{j+1} satisfy (A.6)

Hence, all of $T_2, T_3, T_4 \ldots T_{k-1}$ satisfy Eq. (A.6). This implies that they all have cardinalities no greater than the cardinality of T_k. ∎

REFERENCES

1. C. Beeri, R. Fagin, D. Maier, A. Mendelzon, J. Ullman, and M. Yannakakis, Properties of acyclic database schemes, *in* "STOC-1981," Milwaukee, WI, pp. 355–362, 1981.
2. C. Beeri, R. Fagin, D. Maier, and M. Yannakakis, On the desirability of acyclic database schemes, *JACM* **30** (1983), 479–513.
3. J. Bentley, Multidimensional binary search trees used for associative searching, *CACM* **18** (1975), 509–517.
4. J. Bentley, Multidimensional divide-and-conquer, *CACM* **23** (1980), 214–229.
5. J. Bentley and H. Mauer, Efficient worst-case data structures for range searching, *Acta Inform.* **13** (1980), 155–168.
6. J. Bentley and J. Saxe, Decomposable searching problems: Static to dynamic transformations, *J. Algorithms* **1** (1980), 301–358.
7. P. Bernstein, M. Brodie, S. Ceri, D. DeWitt, M. Franklin, H. Garcia-Molina, J. Gray, J. Held, J. Helterstein, H. Jagadish, M. Lesk, D. Maier, J. Naughton, H. Pirahesh, M. Stonebraker, and J. Ullman, The Asilomar Report on Database Research, *AMC Sigmod Rec.* **27**, 4 (1998), 74–80.
8. P. Bernstein and D. Chiu, Using semijoins to solve relational queries, *JACM* **21** (1981), 25–40.
9. P. Bernstein and N. Goodman, The power of natural semijoins, *SIAM J. Comput.* **10** (1981), 751–771.
10. K. Beyer and R. Ramakrishnan, Bottom-up computation of sparse iceberg cubes and maintenance of data cubes and summary data in a warehouse, *in* "SIGMOD-1999," Philadelphia, PA, pp. 359–370, 1999.
11. J. Cai and R. Paige, Binding performance of language design, *in* "POPL-1987," Montreal, Quebec, pp. 85–97.
12. B. Chazelle, Filter search: A new approach to query processing, *SIAM J. Comput.* **15** (1986), 703–724.

13. B. Chazelle, A functional approach to data structures and its use in multidimensional searching, *SIAM J. Comput.* **17** (1988), 427–462.

14. B. Chazelle, Lower bounds for orthogonal range searching, *JACM* **37** (1990), 200–212; and also *JACM* **37** (1990), 439–463.

15. B. Chazelle and L. Guibas, Fractional cascading: A data structuring technique, *Algorithmica* **1** (1986), 133–162 and 163–191 (a 2-part article).

16. P. Deshpande and J. Naghton, Aggregate aware caching for multi-dimensional queries, *in* "EBDIT-2000," Lecture Notes in Computer Science, Vol. 1777, pp. 168–182, Springer-Verlag, Berlin, 2000.

17. P. Deshpande, K. Ramasamy, A. Shukla, and J. Naghton, Caching multidimensional queries using chunks, "SIGMOD-1998," Seattle, WA, pp. 259–270, 1998.

18. H. Edelsbrunner, A note on dynamic range searching, *Bull. EATCS* **15** (1981), 34–40.

19. H. Edelsbrunner and M. Overmars, Batch solutions to decomposable search problems, *J. Algebra* **5** (1985), 515–542.

20. R. Fagin, Degrees of acyclicity for hypergraphs and relational database schemes, *JACM* **30** (1983), 514–550.

21. M. Fredman, Lower bounds on some optimal data structures, *SIAM J. Comput.* **10** (198), 1–10.

22. M. Fredman, A lower bound on the complexity of range queries, *JACM* **28** (1981), 696–706.

23. M. Fredman and D. Willard, Surpassing the information theoretic barrier with fusion trees, *J. Comput. System Sci.* **47** (1993), 424–436.

24. S. Geffner, D. Agrawal, and A. Abbadi, The dynamic data cube, *in* "EBDIT-2000," Konstanz, Germany, pp. 237–253, 2000.

25. N. Goodman and O. Shmueli, Tree queries: A simple class of relational queries, *ACM TODS* **7** (1982), 653–677.

26. N. Goodman and O. Shmueli, Syntactic characterization of tree schemes, *JACM* **30** (1983), 767–786.

27. D. Goyal and R. Paige, The formal speedup of the linear time fragment of Willard's relational calculus subset, *in* "Algorithmic Languages and Calculi" (Bird and Meertens, Eds.), pp. 382–414, Chapman & Hall, 1997.

28. S. Grumbach, M. Rafnelli, and L. Tinini, Querying aggregate data, *in* "PODS-1999," Philadelphia, PA, pp. 174–184, 1999.

29. A. Gupta, V. Harinarayan, and D. Quass, Aggregate query processing in data warehousing applications, *in* "VLDB-1995," Zurich, Switzerland, pp. 358–369, 1995.

30. V. Harinarayan, A. Rajaraman, and J. Ullman, Implementing data cube efficiently, *in* "SIGMOD 1996," Montreal, Quebec, pp. 205–216, 1996.

31. J. Hellerstein, P. Haas, and H. Wang, On-line aggregation, *in* "SIGMOD-1997," Tucson, AZ, pp. 171–182, 1997.

32. C. Ho. R. Agrawal, N. Migiddo, and R. Srikant, Range queries in OLAP data cubes, *in* "SIGMOD-1997," Tucson, AZ, pp. 73–88, 1997.

33. S. Koenig and R. Paige, A transformational framework for the automatic control of derived data, *in* "VLDB-1981," Cannes, France, pp. 306–318, 1981.

34. D. Lee and F. Preparata, Computational geometry: A survey, *IEEE Trans. Comput.* **33** (1984), 1072–1101.

35. D. Lee and C. Wong, Worst-case analysis of region and partial region searches in multi-dimensional binary search trees and balanced quad trees, *Acta Inform.* **9** (1977), 23–29.

36. Y. Lien, On the equivalence of database models, *JACM* **29** (1982), 333–362.

37. G. Lueker and D. Willard, A data structure for dynamic range queries, *Inform. Process. Lett.* **15** (1982), 209–213.

38. K. Mehlhorn, "Data Structures and Algorithms," Vol. 3, Multidimensional Searching and Computational Geometry, Springer-Verlag, Berlin, 1984.

39. K. Mehlhorn and S. Naher, Dynamic fractional cascading, *Algorithmica* **5** (1990), 215–241.

40. I. Mumick, D. Quass, and B. Mumick, Maintenance of data cubes and summary data in a warehouse, *in* "SIGMOD-1997," Tucson, AZ, pp. 100–111, 1997.

41. W. Nutt, Y. Sagiv, and S. Shuring, Deciding equivalence among aggregate queries, *in* "PODS-1998," Seattle, WA, pp. 214–223, 1998.

42. M. Overmars, "The Design of Dynamic Data Structure," Lecture Notes in Computer Science, Vol. 156, Springer-Verlag, Berlin, 1983.

43. M. Overmars, Range searching on a grid, *J. Algebra* **9** (1988), 254–275.

44. M. Overmars and J. v. Leeuwen, Two general methods for dynamizing decomposable searching problems, *Computing* **26** (1981), 155–166.

45. G. Ozsoyoglu, Z. Ozsoyoglu, and V. Matos, Extending relational algebra and calculus with set-valued and aggregate functions, *ACM TODS* **25**(4) (1996), 8–13.

46. R. Paige, "Formal Differentiation—A Program Synthesis Technique," UMI Research Press, 1981 (277pp); revised Ph.D. thesis, NYU, June 1979, which appeared in Courant CS Rep 15, pp. 269–658, September 1979.

47. R. Paige, Applications of finite differencing to database integrity control and query/transaction optimization, *in* "Advances in Database Theory," (H. Gallaire, J. Minker, and J. M. Nicolas, Eds.), Vol. 2, pp. 171–210, Plenum Press, New York, 1984.

48. R. Paige and F. Henglein, Mechanical translation of set theoretic problem specifications into efficient RAM code—A case study, *J. Symbolic Comput.* **4**(2) (1987), 207–232.

49. C. Papadimitriou and M. Yannakakis, On the complexity of database queries, *J. Comput. System Sci.* **58** (1999), 407–427.

50. S. Rao, A. Badia, and D. Van Gucht, Providing better support for a class of decision support queries, *in* "SIGMOD-1996," Montreal, Quebec, pp. 217–227, 1996.

51. N. Rousspoulos, Y. Kotidis, and P. Rousspoulos, Cubetree: Organization of an bulk incremental updates on the data cube, *in* "SIGMOD-1997," Tucson, AZ, pp. 89–99, 1997.

52. Y. Sagiv and O. Shmeuli, Solving queries by tree projections, *ACM TODS* **18** (1993), 487–511.

53. S. Tsur, J. Ullman, S. Abiteboul, C. Clifton, R. Motwani, S. Nestorov, and A. Rosenthal, Query flocks, a generalization of association-rule mining, *in* "SIGMOD-1998," Seattle, WA, pp. 1–12, 1998.

54. S. Tsur and C. Zaniolo, LDL: A logic based data language, *in* "VLDB-1986," Kyoto, Japan, pp. 75–90, 1986.

55. J. Ullman, "Database and Knowledge Base Systems," Vols. I and II, Computer Science Press, Rockville, MD, 1989.

56. P. Vaidya, Space time tradeoffs for orthogonal range queries, *in* "STOC-1985," Providence, RI, pp. 169–174, 1985.

57. M. Vardi, The complexity of relational query languages, *in* "STOC-1982," San Francisco, CA, pp. 137–146, 1982.

58. M. Vardi, On the complexity of bounded-variable queries, *in* "STOC-1995," Las Vegas, NV, pp. 266–276, 1995.

59. D. Willard, "Predicate-Oriented Database Search Algorithms," Harvard Ph.D. dissertation, May 1978, published in the Garland Series of Outstanding Dissertations in Computer Science.

60. D. Willard, Efficient processing of relational calculus expressions using range query theory, *in* "SIGMOD-1984," Boston, MA, pp. 160–172.

61. D. Willard, New data structures for orthogonal queries, *SIAM J. Comput.* **14** (1985), 233–253.

62. D. Willard, On the application of sheared retrieval to orthogonal range queries, *in* "1986 Computational Geometry Conference," pp. 80–90, 1986.

63. D. Willard, Multidimensional search trees that provide new types of memory reductions, *JACM* **34** (1987), 846–858.

64. D. Willard, Lower bounds for the addition-subtraction operations in orthogonal range queries, *Inform. Comput.* **82** (1989), 45–64.

65. D. Willard, Quasi-linear algorithm for processing relational calculus expressions, *in* "PODS-1990," Nashville, TN, pp. 243–257, 1990.

66. D. Willard, Optimal sampling residues for differentiable database problems, *JACM* **38** (1991), 104–119.

67. D. Willard, Applications of range query theory to relational database selection and join operations, *J. Comput. System Sci.* **52** (1996), 157–169.

68. D. Willard, Examining computational geometry, Van Emde Boas trees and hashing from the perspective of the fusion tree, *SIAM J. Comput.* **29** (2000), 1030–1049.

69. D. Willard and G. Lueker, Adding range restriction capability to dynamic data structures, *JACM* **32** (1985), 597–617.

70. M. Yannakakis, Algorithms for acyclic database schemes, *in* "VLDB-1981," Cannes, France, pp. 82–94, 1981.

71. A. Yao, On the complexity of maintaining partial sums, *SIAM J. Comput.* **14** (1985), 277–288.

72. C. Yu, M. Ozsoyoglu, and K. Lam, Optimization of distributed tree distributed queries, *J. Comput. System Sci.* **29** (1984), 409–445; *in* "IEEE-Compsac-1979," pp. 306–312, 1979.

73. M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirajesh, and M. Urata, Answering complex SQL queries using automatic summary tables, *in* "SIGMOD-2000," Dallas, TX, pp. 105–116, 2000.

74. C. Zaniolo, "Analysis and Design of Relational Schemata for Database Schemes," Ph.D. dissertation, 1976. Available as Tech Report UCLA-Eng-7669.

75. C. Zaniolo, Design and implementation of a logic based language for data intensive applications, *in* "Proceedings of the 5th Symposium on Logic Programming," Chicago, IL, pp. 1666–1687, 1988.

76. Y. Zhao, P. Deshpande and J. Naughton, An array-based algorithm for simultaneous multi-dimensional aggregates, *in* "SIGMOD-1997," Tucson, AZ, pp. 159–170, 1997.