Incremental Recomputation of Recursive Queries with Nested Sets and Aggregate Functions*

Leonid Libkin
Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974, USA
Email: libkin@bell-labs.com

Limsoon Wong
BioInformatics Centre &
Institute of Systems Science
Singapore 119597
Email: limsoon@iss.nus.sg

Abstract

We examine the power of incremental evaluation systems that use an SQL-like language for maintaining recursively-defined views. We show that recursive queries such as transitive closure, and "alternating paths" can be incrementally maintained in a nested relational language, when some auxiliary relations are allowed. In the presence of aggregate functions, even more queries can be maintained, for example, the "same generation" query. In contrast, it is still an open problem whether such queries are maintainable in relational calculus. We then restrict the language so that no nested relations are involved (but we keep the aggregate functions). Such a language captures the capability of most practical relational database systems. We prove that this restriction does not reduce the incremental computational power; that is, any query that can be maintained in a *nested* language with aggregates, is still maintainable using only *flat* relations. We also show that one does not need auxiliary relations of arity more than 2. In particular, this implies that the recursive queries maintainable in the nested language with aggregates, can be also maintained in a practical relational database systems using auxiliary tables of arity at most 2. This is again in sharp contrast to maintenance in relational calculus, which admits a strict arity-based hierarchy.

1 Introduction

It is common knowledge that the expressiveness of relational calculus is limited. For example, recursive queries such as the transitive closure cannot be defined [3]. However, in a real database system, one can try to overcome this problem by storing both the relation and its transitive closure and updating the latter whenever edges are added to or removed from the former. In other words, the recursive queries are evaluated and maintained incrementally. One can think of the result of such a recursive query as a view of the database and the incremental evaluation of the query as view maintenance.

The above leads us to the concept of an *incremental evaluation system*, or IES. An IES($\mathcal{L}$) is a system consisting of a finite set of "update" functions expressible in the language $\mathcal{L}$, where each of these functions takes as input the old database, the old answer, the old auxiliary database, and the update. We require the update to be *permissible* according to certain criteria specified for the IES($\mathcal{L}$). In this report, the criteria for permissible update is restricted to insertion and deletion of a single tuple.

*Part of this work was done when Wong was visiting Bell Labs and when Libkin was visiting Institute of Systems Science.

For each permissible update that is coming in, the system uses its update functions to compute the new answer to the query and the new auxiliary database. A restriction is also imposed so that the constants that appear in auxiliary database must also appear in the database or in the answer or in some fixed set. In this report, this fixed set is $\mathbb{Q}$, the set of rational numbers.

We use the first-order incremental evaluation system, $\mathsf{IES}(\mathcal{FO})$(called FOIES in [10]), to illustrate the concept. $\mathsf{IES}(\mathcal{FO})$ uses first-order logic to express update functions [9, 12]. The permissible updates are tuples to be inserted or deleted from the input relations. For each relation symbol R, we use R^o to refer to the instance of R *before* an update, and R^n the instance of R *after* the update (here 'o' stands for old and 'n' for new). Consider the view $EVEN$ that is defined to be $\{1\}$ if the relation R has even cardinality and $\{\}$ if R has odd cardinality. While $EVEN$ is well known to be inexpressible in first-order logic [1], it can be expressed in $\mathsf{IES}(\mathcal{FO})$. The update function when a tuple o is deleted from R is given by

$$EVEN^n(1) \quad \text{iff} \quad (R(o) \wedge \neg EVEN^o(1)) \vee (\neg R(o) \wedge EVEN^o(1)).$$

The update function when a tuple o is inserted into R is given by

$$EVEN^n(1) \quad \text{iff} \quad (R(o) \wedge EVEN^o(1)) \vee (\neg R(o) \wedge \neg EVEN^o(1)).$$

The $\mathsf{IES}(\mathcal{FO})$ that we used to maintain $EVEN$ as above is also called a space-free $\mathsf{IES}(\mathcal{FO})$, because it does not make use of any auxiliary relations. The transitive closure of acyclic graphs is another view that can be maintained by a space-free $\mathsf{IES}(\mathcal{FO})$ [9]. However, the transitive closure of general graphs cannot be maintained using space-free $\mathsf{IES}(\mathcal{FO})$[10, 7]. Thus, it is sometimes necessary to use auxiliary relations. We write $\mathsf{IES}(\mathcal{FO})_k$ to mean the subclass of $\mathsf{IES}(\mathcal{FO})$ where auxiliary relations of arities up to k can be used. Observe that, with maximal arity k, the auxiliary relations can hold at most $O(n^k)$ tuples, where n is the number of constants in the input database.

There are some interesting queries that can be maintained by $\mathsf{IES}(\mathcal{FO})$ with some auxiliary relations. For the transitive closure of undirected graphs, it can be maintained in $\mathsf{IES}(\mathcal{FO})_3$ [20] and even in $\mathsf{IES}(\mathcal{FO})_2$ [10]. But it is open if there is a $\mathsf{IES}(\mathcal{FO})$ for transitive closure of general directed graphs. Also, Dong and Su [10] showed that the $\mathsf{IES}(\mathcal{FO})_k$ hierarchy is strict for $k \leq 2$. More recently, using a result of Cai [6], Dong and Su showed in the journal version of their paper [10] that the $\mathsf{IES}(\mathcal{FO})_k$ hierarchy is strict for every k. However, their example query that proved the strict inclusion of $\mathsf{IES}(\mathcal{FO})_k$ in $\mathsf{IES}(\mathcal{FO})_{k+1}$ had input arity much greater than k. It is open whether the $\mathsf{IES}(\mathcal{FO})_k$ hierarchy remains strict if we restrict to queries having fixed input arity.

The two open questions above render $\mathsf{IES}(\mathcal{FO})$ a little unsatisfactory. $\mathsf{IES}(\mathcal{FO})$ uses first-order logic as its ambient language. First-order logic or relational algebra does not properly reflect the power of practical relational systems. One wonders if transitive closure can be maintained in a *real* relational database. Such databases use SQL as their query language. May be transitive closure can be maintained using SQL after all? One also wonders if strictness of the $\mathsf{IES}(\mathcal{FO})_k$ hierarchy is natural. Maybe such a hierarchy will collapse if the ambient language is SQL?

So we study the incremental evaluation system whose ambient language is $\mathcal{NRC}^{\text{aggr}}$, our reconstruction of SQL based on a nested relational calculus. We use the notation $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ to denote the incremental evaluation system where both the input database and answer are flat relations, but the auxiliary database can be nested relations. We use the notation $\mathsf{IES}(\mathcal{SQL})$ when the auxiliary database is restricted to flat relations. The rationale for the $\mathsf{IES}(\mathcal{SQL})$ is that it more closely approximates what could be done in a relational database, which can store only flat tables. With features such as nesting of intermediate data (as in GROUPBY) and aggregates, the ambient language has essentially the power of SQL, hence the notation. As $\mathcal{NRC}^{\text{aggr}}$ is more expressive than first-order logic, both $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ and $\mathsf{IES}(\mathcal{SQL})$ are more powerful than $\mathsf{IES}(\mathcal{FO})$. So we will concentrate on queries that are not known to be expressible in $\mathsf{IES}(\mathcal{FO})$.

Our results are organized as follows. In Section 2, we describe $\mathcal{NRC}^{\text{aggr}}$, our reconstruction of SQL. In Section 3, we give a formal definition of $\mathsf{IES}(\mathcal{L})$. In Section 4, we show that transitive closure of arbitrary graphs, "alternating paths" of arbitrary (and-or) graphs, and "same generation" of acyclic graphs can be maintained in $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ using some auxiliary space. In addition, we also show that any query that can be implemented using a single application of structural recursion [5] can be maintained in $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ using some auxiliary space. In fact, some of these do not even require the power of counting.

In Section 5, we consider $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ where the auxiliary relations are restricted to be flat; that is, we consider $\mathsf{IES}(\mathcal{SQL})$. We show that a linear order on data can be generated in the context of $\mathsf{IES}(\mathcal{SQL})$. We then use this order to encode $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ into $\mathsf{IES}(\mathcal{SQL})$, demonstrating that storing flat tables is sufficient. This result says in essence that queries that can be maintained by $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ are precisely those that can be maintained by a practical relational system.

In Section 6, we consider the effect that restriction on the arity of auxiliary relations can have on $\mathsf{IES}(\mathcal{SQL})$. We prove that a hierarchy does not form under the arity restriction on auxiliary relations. In fact, it does not form even when an arity restriction is imposed on input data as well. This result points out that a hierarchy based on arity restriction (such as the $\mathsf{IES}(\mathcal{FO})_k$ hierarchy) is not robust.

In Section 7, we discuss the physical costs of maintaining queries in $\mathsf{IES}(\mathcal{SQL})$. The total overall cost may be unacceptably high, but each incremental step is always guaranteed to be tractable.

All proofs are given in the full report[18].

2 Nested Relational Calculus with Aggregates

Let us start by describing our ambient query language. We want this language to be more powerful than the relational calculus in two ways: it will deal with nested relations, and will use aggregate functions. There are many choices for such a language. We use the language similar to those considered in [5, 15, 17, 8]. These languages have been extensively studied and they are easier to work with than most other nested formalisms. However, we would like to emphasize the the choice of a particular language is not central to our problems. In particular, our results extend to any language with the same power as the language $\mathcal{NRC}^{\text{aggr}}$ presented below.

The language $\mathcal{NRC}^{\text{aggr}}$ is obtained by extending the nested relational calculus $\mathcal{NRC}(=)$ of [5, 23] by arithmetics and aggregate functions. The motivation for considering $\mathcal{NRC}^{\text{aggr}}$ is that it is a much more realistic query language than relational algebra. Indeed, as explained later, one can consider $\mathcal{NRC}^{\text{aggr}}$ to be a theoretical reconstruction of SQL, the de facto language of the commercial world.

We present the language incrementally. We start from $\mathcal{NRC}(=)$, which is equivalent to the usual nested relational algebra [2, 5]. The data types that can be manipulated are:

$$s ::= b \mid s_1 \times \cdots \times s_n \mid \{s\}$$

The symbol b ranges over base types like Booleans $\mathbb{B}$, rational numbers $\mathbb{Q}$, etc. The type $s_1 \times \cdots \times s_n$

contains n-ary tuples whose components have types s_1, ..., s_n respectively. The objects of type $\{s\}$ are sets of finite cardinality whose elements are objects of type s.

As can be seen from the data types, $\mathcal{NRC}(=)$ is a language for arbitrarily nested relations. The syntax and typing rules of $\mathcal{NRC}$ are given below.

$$\frac{}{x^s : s} \qquad \frac{}{c : b} \qquad \frac{e : s_1 \times \cdots \times s_n}{\pi_i\ e : s_i} \qquad \frac{e_1 : s_1 \quad \cdots \quad e_n : s_n}{(e_1, \ldots, e_n) : s_1 \times \cdots \times s_n}$$

$$\frac{}{\{\}^s : \{s\}} \qquad \frac{e : s}{\{e\} : \{s\}} \qquad \frac{e_1 : \{s\} \quad e_2 : \{s\}}{e_1 \cup e_2 : \{s\}} \qquad \frac{e_1 : \{t\} \quad e_2 : \{s\}}{\bigcup\{e_1 \mid x^s \in e_2\} : \{t\}}$$

$$\frac{e_1 : s \quad e_2 : s}{e_1 = e_2 : \mathbb{B}} \qquad \frac{}{\mathit{true} : \mathbb{B}} \qquad \frac{}{\mathit{false} : \mathbb{B}} \qquad \frac{e_1 : \mathbb{B} \quad e_2 : s \quad e_3 : s}{\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 : s}$$

We often omit the type superscripts as they can be inferred. An expression e having free variables $\vec{x}$ is interpreted as a function $f(\vec{x}) = e$, which given input $\vec{O}$, of the same arity as $\vec{x}$, produces $e[\vec{O}/\vec{x}]$ as its output. Here $[\vec{O}/\vec{x}]$ is the substitution replacing the ith component of $\vec{x}$ by the ith component of $\vec{O}$. An expression e with no free variable can be regarded as a constant function $f \equiv e$.

Let us briefly recall the semantics; see also [5]. Variables x^s are available for each type s. Every constant c of base type b is available. The operations for tuples are standard. Namely, $(e_1, \ldots, e_n)$ forms an n-tuple whose i component is e_i and $\pi_i\ e$ returns the i component of the n-tuple e.

$\{\}$ forms the empty set. $\{e\}$ forms the singleton set containing e. $e_1 \cup e_2$ unions the two sets e_1 and e_2. $\bigcup\{e_1 \mid x \in e_2\}$ maps the function $f(x) = e_1$ over all elements in e_2 and then returns their union; thus if e_2 is the set $\{o_1, \ldots, o_n\}$, the result of this operation would be $f(o_1) \cup \cdots \cup f(o_n)$. For example, $\bigcup\{\{(x, x)\} \mid x \in \{1, 2\}\}$ evaluates to $\{(1, 1), (2, 2)\}$.

The operations for Booleans are also quite typical, with *true* and *false* denoting the two Boolean values. $e_1 = e_2$ returns *true* if e_1 and e_2 have the same value and returns *false* otherwise. Finally, $\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3$ evaluates to e_2 if e_1 is *true* and evaluates to e_3 if e_1 is *false*. We provided equality test on every type s. However, this is equivalent to having equality test restricted to base types together with emptiness test for set of base types [22].

$\mathcal{NRC}$ possesses the so-called conservative extension property [23]: if a function $f : s_1 \to s_2$ is expressible in $\mathcal{NRC}$, then it can be expressed using an expression of height no more than that of s_1 and s_2. The height of a type is defined as its depth of nesting of set brackets. The height of an expression is defined as the maximum height of all types that appear in its typing derivation. More specifically, if $f : s_1 \to s_2$ takes flat relations to flat relations and is expressible in $\mathcal{NRC}$, then it is also expressible in the standard flat relational algebra [19, 5].

It is a common misconception that the relational algebra is the same as SQL. The truth is that all versions of SQL come with three features that have no equivalence in relational algebra: SQL extends the relational calculus by having arithmetic operations, a group-by operation, and various aggregate functions such as `AVG`, `COUNT`, `SUM`, `MIN`, and `MAX`.

It is known [5] that the group-by operator can already be simulated in $\mathcal{NRC}(=)$. The others need to be added. The arithmetic operators are the standard ones: $+$, $-$, $\cdot$, and $\div$ of type $\mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$. We also add the order on the rationals: $\leq_{\mathbb{Q}} : \mathbb{Q} \times \mathbb{Q} \to \mathbb{B}$. As to aggregate functions, we add just the following construct

$$\frac{e_1 : \mathbb{Q} \quad e_2 : \{s\}}{\sum\{\!|e_1 \mid x^s \in e_2|\!\} : \mathbb{Q}}$$

The semantics is this: map the function $f(x) = e_1$ over all elements of e_2 and then add up the results. Thus, if e_2 is the set $\{o_1, \ldots, o_n\}$, it returns $f(o_1) + \cdots + f(o_n)$. For example, $\sum\{\!|1 \mid x \in X|\!\}$ returns the cardinality of X. Note that this is different from adding up the values in $\{f(o_1), \ldots, f(o_n)\}$; in the example above, doing so yields 1 as no duplicates are kept. To emphasize that duplicate values of f are being added up, we use bag (multiset) brackets $\{\!|\ |\!\}$ in this construct.

We denote this theoretical reconstruction of SQL by $\mathcal{NRC}^{\text{aggr}}$. That is, $\mathcal{NRC}^{\text{aggr}}$ has all the constructs of $\mathcal{NRC}(=)$, the arithmetic operations $+, -, \cdot$ and $\div$, the summation construct $\sum$ and the linear order on the rationals. It was shown in [15, 17] that all SQL aggregate functions mentioned above can be implemented in $\mathcal{NRC}^{\text{aggr}}$. It is also known [15, 17] that $\mathcal{NRC}^{\text{aggr}}$ has the conservative extension property and thus its expressive power depends only on the height of input and output and is independent of the height of intermediate data. So to conform to SQL, it suffices to restrict our input and output to height at most one, that is, to the usual flat relational databases.

Before we begin studying $\mathcal{NRC}^{\text{aggr}}$ in the setting of an incremental evaluation system, let us briefly introduce a nice shorthand, based on the comprehension notation [21, 4], for writing $\mathcal{NRC}^{\text{aggr}}$ queries. Recall from [4, 5, 23] that the comprehension $\{e \mid A_1, \ldots, A_n\}$, where each A_i either has the form $x_i \in e_i$ or is an expression e_i of type $\mathbb{B}$, has a direct correspondent in $\mathcal{NRC}$ that is given by recursively applying the following equations:

- $\{e \mid x_i \in e_i, \ldots\} = \bigcup\{\{e \mid \ \ldots\} \mid x_i \in e_i\}$
- $\{e \mid e_i, \ldots\} = \textit{if } e_i \textit{ then } \{e \mid \ \ldots\} \textit{ else } \{\}$

The comprehension notation is more user-friendly than the syntax of $\mathcal{NRC}^{\text{aggr}}$. For example, it allows us to write $\{(x, y) \mid x \in e_1,\ y \in e_2\}$ for the cartesian product of e_1 and e_2 instead of the clumsier $\bigcup\{\bigcup\{\{(x, y)\} \mid y \in e_2\} \mid x \in e_1\}$.

In addition to comprehension, we also find it convenient to use a little bit of pattern matching, which can be removed in a straightforward manner. For example, we write $\{(x, z) \mid (x, y) \in e_1,\ (y', z) \in e_2,\ y = y'\}$ for relational composition instead of the more official $\{(\pi_1\ X, \pi_2\ Y) \mid X \in e_1,\ Y \in e_2,\ \pi_2\ X = \pi_1\ Y\}$ or the much clumsier $\bigcup\{\bigcup\{\textit{if } \pi_2\ X = \pi_1\ Y \textit{ then } \{(\pi_1\ X, \pi_2\ Y)\} \textit{ else } \{\} \mid Y \in e_2\} \mid X \in e_1\}$. Here X and Y denote edges ((x, y) and (y, z) respectively), whose components, x, y and z, are obtained by applying projections π_1 and π_2.

3 Formal Definition of $\mathsf{IES}(\mathcal{L})$

The definition of $\mathsf{IES}(\mathcal{L})$ is very similar to the definitions of Dong-Su's FOIES [10] and Immerman-Patnaik's Dyn-$\mathcal{C}$ [20]. The idea is that, in order to incrementally maintain a query Q, we do the following. At the first step, we initialize auxiliary data and compute Q assuming that the input is empty. Then we provide functions that, upon each insertion or deletion, correctly update both the answer to Q and the auxiliary data. If the initializing and the updating functions are definable in $\mathcal{L}$, we say that Q is expressible in $\mathsf{IES}(\mathcal{L})$. If all auxiliary data are flat relations of arity not exceeding k, we say that Q is expressible in $\mathsf{IES}(\mathcal{L})_k$.

While this informal definition is sufficient for understanding the results of the paper, we give a formal definition of $\mathsf{IES}(\mathcal{L})$ for the sake of completeness. Suppose we are given a type $S = \{s_1\} \times \ldots \times \{s_m\}$, where $s_1, \ldots, s_m$ are record types. We consider elementary updates of the form $ins_i(x)$ and $del_i(x)$, where x is of type s_i. Given an object X of type S, applying such an update results in inserting x into or deleting x from the ith set in X, that is, the set of type $\{s_i\}$. Given a sequence $\mathcal{U}$ of updates, $\mathcal{U}(X)$ denotes the result of applying the sequence $\mathcal{U}$ to an object X of type S.

Given a query Q of type $S \to T$, and a type T_{aux} (of auxiliary data), consider a collection of $\mathcal{F}_Q$ functions:

$$\begin{array}{ll} f_{\text{init}} : S \to T & f_{\text{init}}^{\text{aux}} : S \to T_{\text{aux}} \\ f_{\text{del}}^{i} : s_i \times S \times T \times T_{\text{aux}} \to T & f_{\text{del}}^{\text{aux}} : s_i \times S \times T \times T_{\text{aux}} \to T_{\text{aux}} \\ f_{\text{ins}}^{i} : s_i \times S \times T \times T_{\text{aux}} \to T & f_{\text{ins}}^{\text{aux}} : s_i \times S \times T \times T_{\text{aux}} \to T_{\text{aux}} \end{array}$$

Given an elementary update u, we associate two functions with it. The function $f_u : S \times T \times T_{\text{aux}} \to T$ is defined as $\lambda(X,Y,Z).f_{\text{del}}^{i}(a,X,Y,Z)$ if u is $del_i(a)$, and as $\lambda(X,Y,Z).f_{\text{ins}}^{i}(a,X,Y,Z)$ if u is $ins_i(a)$. We similarly define $f_u^{\text{aux}} : S \times T \times T_{\text{aux}} \to T_{\text{aux}}$.

Given a sequence of updates $\mathcal{U} = \{u_1, \ldots, u_l\}$, define inductively the collection of objects: $X_0 = \emptyset : S, RES_0 = f_{\text{init}}(X_0),\ AUX_0 = f_{\text{init}}^{\text{aux}}(X_0)$ (where $\emptyset$ of type S is a product of m empty sets), and

$$X_{i+1} = u_{i+1}(X_i), \quad RES_{i+1} = f_{u_{i+1}}(X_i, RES_i, AUX_i), \quad AUX_{i+1} = f_{u_{i+1}}^{\text{aux}}(X_i, RES_i, AUX_i)$$

Finally, we define $\mathcal{F}_Q(\mathcal{U})$ as RES_l.

We now say that there exists an *incremental evaluation system* for Q in $\mathcal{L}$ if there is a type T_{aux} and a collection $\mathcal{F}_Q$ of functions, typed as above, such that, for any sequence $\mathcal{U}$ of updates, $\mathcal{F}_Q(\mathcal{U}) = Q(\mathcal{U}(\emptyset))$. We also say then that Q is expressible in $\mathsf{IES}(\mathcal{L})$. If T_{aux} is a product of flat relational types, none of arity more than k, we say that Q is in $\mathsf{IES}(\mathcal{L})_k$.

Since every expression in $\mathcal{NRC}$ or $\mathcal{NRC}^{\text{aggr}}$ has a well-typed function associated with it, the definition above applies to these languages.

4 Power of $\mathsf{IES}(\mathcal{NRC})$ and $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$

It is known [7] that recursive queries such as transitive closure cannot be expressed in space-free $\mathsf{IES}(\mathcal{NRC})$ and space-free $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$. In this section, we focus on the power of $\mathsf{IES}(\mathcal{NRC})$ and $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ in the presence of auxiliary data. We prove four expressibility results. We first show that transitive closure and the "alternating path" query are expressible in $\mathsf{IES}(\mathcal{NRC})$ (and hence in $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$). Furthermore, any query expressed using one application of structural recursion (with parameter functions defined in $\mathcal{NRC}$) can be expressed in $\mathsf{IES}(\mathcal{NRC})$. If the parameter functions are defined in $\mathcal{NRC}^{\text{aggr}}$, then such a query is expressible in $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$. Finally, we show that the "same-generation" query is expressible in $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$, although it is not expressible in space-free $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$.

Proposition 4.1 *Transitive closure of arbitrary graphs is expressible in* $\mathsf{IES}(\mathcal{NRC})$. □

We also consider a generalization of the transitive closure query, namely the "alternating paths" query, cf. [14]. This query is complete with respect to first-order reductions for PTIME. Since transitive closure is complete for NLOGSPACE, it is likely that the "alternating paths" is harder than the transitive closure.

Suppose we are given a graph G and a subset U of the nodes in G (U is for "universal"). Nodes not in the set U are "existential." Intuitively, an alternating path between two nodes must go through every descendant of a universal node, and through just one descendant of an existential node; thus, one can think of the transitive closure query as a special case of this one when $U = \emptyset$. More formally, we define $apath(x, y)$ to be the smallest relation such that: (1) $apath(x, x)$ holds for each node x, and (2) if $x \in U$ and there is an edge leaving x, and for all edges (x, z) it is the case that $apath(z, y)$, then $apath(x, y)$, and (3) if $x \notin U$, and for some edge (x, z), $apath(z, y)$ holds, then $apath(x, y)$. The "alternating paths" query is simply this: given a graph, compute the *apath* relation.

Proposition 4.2 *"Alternating paths" of arbitrary graphs can be expressed in* $\mathsf{IES}(\mathcal{NRC})$. $\square$

Corollary 4.3 *Transitive closure and "alternating path" are expressible in* $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$. $\square$

The "same generation" query is another recursive query that often serves as one of canonical examples of queries definable in datalog but not in relational calculus. Two nodes x and y of a graph G are in the same generation if and only if there is a node z in G such that there is a walk (an edge sequence, possibly repeated) that goes from z to x and a walk of the same length that goes from z to y. It is known from [7] that this query cannot be maintained in $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ without using auxiliary space. It turns out that it can be maintained in $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ with some auxiliary space, if the graph is acyclic. Note that we do need the counting power of $\mathcal{NRC}^{\text{aggr}}$. However, the case of arbitrary graphs remains open.

Proposition 4.4 *"Same generation" of acyclic graphs can be expressed in* $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$. $\square$

In general, $\mathsf{IES}(\mathcal{NRC})$ can express queries specified by a single application of the structural recursion operator of [5]. Let us first define this operator. Let $f : s \times t \to t$ be a function expressible in $\mathcal{NRC}$. Let $i : t$ be an object expressible in $\mathcal{NRC}$ (that is, the constant function returning this object is definable). Furthermore, we assume that $f(x, f(x, y)) = f(x, y)$ and $f(x, f(y, z)) = f(y, f(x, z))$ hold. Then the structural recursion operator $sri(f, i) : \{s\} \to t$ is given by the equations: $sri(f, i)(\{\}) = i$, $sri(f, i)(\{x\} \cup Y) = f(x, sri(f, i)(Y))$. This operator is very powerful. It can generate powersets. It can also produce all three example queries considered above: transitive closure, "alternating paths," and "same generation." (Thus these previous results are really corollaries of the next theorem!)

Theorem 4.5 *Any query expressible as $sri(f, i) : \{s\} \to \{t\}$, where $\{s\}$ and $\{t\}$ are flat relation types, and f and i are definable in $\mathcal{NRC}$, can be maintained in* $\mathsf{IES}(\mathcal{NRC})$. $\square$

The same argument applies to $\mathcal{NRC}^{\text{aggr}}$:

Corollary 4.6 *Any query expressible as $sri(f, i) : \{s\} \to \{t\}$, where $\{s\}$ and $\{t\}$ are flat relation types, and f and i are definable in $\mathcal{NRC}^{\text{aggr}}$, can be maintained in* $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$. $\square$

An $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ or $\mathsf{IES}(\mathcal{NRC})$ having input relations $\vec{I}$, answer relation A, and auxiliary relations $\vec{R}$ is said to be *deterministic* if there is a function f such that $f(\vec{I}) = (\vec{R}, A)$. (Note that f needs not be expressible within $\mathcal{NRC}^{\text{aggr}}$.) That is, the values of the auxiliary relations in a deterministic $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ do not depend on the history of updates. Deterministic incremental evaluation systems are interesting in their own right [11]. While we do not examine them further in this paper, it is worth pointing out the following result, follows from the proofs of other results in this section.

Corollary 4.7 *Transitive closure of arbitrary graphs, "alternating paths" of arbitrary graphs, "same generation" of acyclic graphs, as well as any query expressible as $sri(f,i) : \{s\} \to \{t\}$, where $\{s\}$ and $\{t\}$ are flat relation types, can be expressed in deterministic* $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$. $\square$

5 Power of $\mathsf{IES}(\mathcal{SQL})$

We now focus on the power of $\mathsf{IES}(\mathcal{SQL})$, the restriction of $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ to use only flat auxiliary relations. We first show that $\mathsf{IES}(\mathcal{SQL})$ can generate a linear order on all its data. This result is then used to encode $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ into $\mathsf{IES}(\mathcal{SQL})$, showing that the two systems are equivalent. Thus the power of $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ is undiminished even when it is restricted to flat auxiliary relations. Of course this also means that $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ can be fully implemented using any real relational database.

5.1 Ordering in $\mathsf{IES}(\mathcal{SQL})$

Recall that $\mathcal{NRC}^{\text{aggr}}$ is only equipped with a linear order on $\mathbb{Q}$. Linear orders on any other infinite base types are not expressible in $\mathcal{NRC}^{\text{aggr}}$[15]. In this section, we show that in the context of $\mathsf{IES}(\mathcal{SQL})$, a linear order on any base type b can be expressed, when restricted to its "active domain." By active domain, we mean those constants that currently appear in the input database.

Proposition 5.1 *For any base type b,* $\mathsf{IES}(\mathcal{SQL})$ *is always able to maintain an auxiliary relation that defines a linear ordering on all the objects of type b in the active domain of a database.* $\square$

Thus, for each type b, a linear order $<^b: b \times b \to \mathbb{B}$ can always be simulated in $\mathsf{IES}(\mathcal{SQL})$. It is known [15, 16] that if a linear order is available on each base types, then there is enough power in $\mathcal{NRC}^{\text{aggr}}$ to compute a linear order $<: s \times s \to \mathbb{B}$ on every type s. Thus from now on, we assume $<$ is available whenever we are talking about $\mathsf{IES}(\mathcal{SQL})$ when auxiliary relations of arity at least 2 are allowed.

Thus within an $\mathsf{IES}(\mathcal{SQL})$, we can implement a ranking function $rank : \{s\} \to \{s \times \mathbb{Q}\}$ on any set O built up from the active domain of the $\mathsf{IES}(\mathcal{SQL})$: $rank(O) = \{(x, \sum\{\!|\mathit{if}\ y < x\ \mathit{then}\ 1\ \mathit{else}\ 0 \mid y \in O|\!\}) \mid x \in O\}$. Then we can define $rankof : \{s\} \times s \to \mathbb{Q}$ to be a function that given any set O built up from the active domain of the $\mathsf{IES}(\mathcal{SQL})$ and an o in O, produces the rank of o in O: $rankof(O,o) = \sum\{\!|\mathit{if}\ x = o\ \mathit{then}\ r\ \mathit{else}\ 0 \mid (x,r) \in rank(O)|\!\}$. This result is used in the next section to encode nested relations into flat relations.

5.2 $\mathsf{IES}(\mathcal{SQL})$ Equals $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$

Let us begin by comparing the power of $\mathsf{IES}(\mathcal{SQL})$ and $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$. It is clear that $\mathsf{IES}(\mathcal{SQL}) \subseteq \mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$. We prove that the converse holds and hence the two systems are equivalent.

Theorem 5.2 $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}}) = \mathsf{IES}(\mathcal{SQL})$. $\square$

The idea of the proof is that, using *rank*, objects of any type can be encoded with natural numbers. Using this, we manage to encode any auxiliary database in $\mathsf{IES}(\mathcal{NRC}^{\mathrm{aggr}})$ into a product of flat relations, so that it can be used by $\mathsf{IES}(\mathcal{SQL})$. Using *rankof*, we can decode the result, and simulate $\mathsf{IES}(\mathcal{NRC}^{\mathrm{aggr}})$ in $\mathsf{IES}(\mathcal{SQL})$. All the details can be found in the full report[18].

Corollary 5.3 $\mathsf{IES}(\mathcal{NRC}) \subseteq \mathsf{IES}(\mathcal{SQL})$. □

We can conclude that $\mathsf{IES}(\mathcal{SQL})$ can maintain transitive closure of arbitrary graphs and can maintain queries expressed using a single application of *sri*. In the rest of this report, we concentrate on $\mathsf{IES}(\mathcal{SQL})$, because it precisely models real relational databases. However, for convenience and clarity, we give proofs using $\mathsf{IES}(\mathcal{NRC}^{\mathrm{aggr}})$.

It is worth pointing out that previous results on the conservative extension of $\mathcal{NRC}$[19, 23] and $\mathcal{NRC}^{\mathrm{aggr}}$[22, 15] do not imply the collapse of $\mathsf{IES}(\mathcal{NRC}^{\mathrm{aggr}})$ to $\mathsf{IES}(\mathcal{SQL})$. The conservative extension property[23] implies that if the input and output of a (update) function are flat, then the function can be implemented using only flat intermediate data. In a $\mathsf{IES}(\mathcal{NRC}^{\mathrm{aggr}})$ having a non-flat auxiliary relation of nesting depth k, its update functions necessarily have non-flat input of nesting depth k. Thus the conservative extension result on $\mathcal{NRC}$ only guarantees that these update functions will not use intermediate data of nesting depth exceeding k. In other words, in order to guarantee that update functions uses only flat data, it is necessary to guarantee that their input auxiliary relations are also flat. This must be accomplished using means other than the conservative extension property of $\mathcal{NRC}$. This is the significance of the equivalence result above.

6 Arity in $\mathsf{IES}(\mathcal{SQL})$

We write $\mathsf{IES}(\mathcal{SQL})_k$ to mean the subclass of $\mathsf{IES}(\mathcal{SQL})$ that uses auxiliary relations up to arity k. As mentioned earlier, $\mathsf{IES}(\mathcal{FO})_k \subset \mathsf{IES}(\mathcal{FO})_{k+1}$ for all $k > 1$, forming a noncollapsing hierarchy for $\mathsf{IES}(\mathcal{FO})$ based on arity of auxiliary relations. We consider the analogous question on $\mathsf{IES}(\mathcal{SQL})_k$ and show that the hierarchy collapses for $k > 1$. The proof uses a coding method that could also be used to prove that it is possible to maintain the equi-cardinality view of two k-ary relations in $\mathsf{IES}(\mathcal{FO})_2$ [13]. After that, we prove that the two levels below $\mathsf{IES}(\mathcal{SQL})_2$ are strict; thus the $\mathsf{IES}(\mathcal{SQL})_k$ hierarchy has only three levels.

Proposition 6.1 $\mathsf{IES}(\mathcal{SQL})_2 = \mathsf{IES}(\mathcal{SQL})_k$ *for all* $k > 1$. □

Proposition 6.2 $\mathsf{IES}(\mathcal{SQL})_1$ *is strictly less powerful than* $\mathsf{IES}(\mathcal{SQL})_2$. □

Proposition 6.3 *Space-free* $\mathsf{IES}(\mathcal{SQL})$ *is strictly less powerful than* $\mathsf{IES}(\mathcal{SQL})_1$. □

Putting all three propositions above together, we conclude that

Theorem 6.4 *Space-free* $\mathsf{IES}(\mathcal{SQL}) \subset \mathsf{IES}(\mathcal{SQL})_1 \subset \mathsf{IES}(\mathcal{SQL})_2 = \mathsf{IES}(\mathcal{SQL})_{k>2}$. □

This result contrasts sharply with the situation of $\mathsf{IES}(\mathcal{FO})_k$, which is a strict hierarchy. The strictness of the $\mathsf{IES}(\mathcal{FO})_k$ hierarchy were obtained using a result of Cai [6]; it uses queries with input relations of greater and greater arities to separate higher and higher layers of the $\mathsf{IES}(\mathcal{FO})_k$ hierarchy. It is not known if $\mathsf{IES}(\mathcal{FO})_k$ remains strict if we further impose a restriction on arities of input relations. Since the arity hierarchy collapses in the presence of simple extensions such as aggregate functions as in $\mathsf{IES}(\mathcal{SQL})_k$, we feel that a hierarchy based on arities is not robust and not natural for incremental evaluation systems. However, it is still an interesting problem to work out a general hierarchy for incremental evaluation systems.

7 Conclusion

We focused on incremental evaluation systems that use the SQL-like language $\mathcal{NRC}^{\text{aggr}}$. In particular, we examined their power in the presence of auxiliary (nested) relations. With respect to $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$, we proved that they can maintain transitive closure, "alternating paths," and "same generation." These results are in contrast to earlier ones [9, etc.] on $\mathsf{IES}(\mathcal{FO})$, where expressibility of these queries remains unsolved (and the negative results are conjectured). They are also in contrast to earlier results [7] on space-free $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$, where these queries were shown to be inexpressible.

Then we considered the restriction of $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ to $\mathsf{IES}(\mathcal{SQL})$, which are allowed to use only flat auxiliary relations. $\mathsf{IES}(\mathcal{SQL})$ is an interesting and important subclass because it naturally reflects the capability of commercial relational database systems which use SQL and store flat tables. We showed that $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ and $\mathsf{IES}(\mathcal{SQL})$ have the same power. Thus all queries that can be expressed in $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ can also be maintained using a standard relational database system. We further proved that every $\mathsf{IES}(\mathcal{SQL})$ can be replaced by one that uses auxiliary relations of arity at most 2. That means arity restriction on auxiliary relations does not lead to a hierarchy in $\mathsf{IES}(\mathcal{SQL})$. This contrasts with [10] showing that arity restriction on auxiliary relations leads to a strict hierarchy in $\mathsf{IES}(\mathcal{FO})$.

In some of our proofs, it can be observed that the amount of auxiliary data involved could be exponential with respect to the size of the history of updates. (The size of the history of updates to a $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ is defined as the sum of the size of all the tuples that were inserted to or deleted from the $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ up to that point in time.) However, at each update, the size of auxiliary data is changed only a polynomial amount from its current size. Nevertheless, we do not know of a method for maintaining recursive views such as transitive closure of arbitrary graphs in $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ that uses only a polynomial amount of space. We leave the search for such a method or the disprove of its existence for future work.

Acknowledgements. We thank Michael Benedikt, Ke Wang, and especially Guozhu Dong for numerous discussions and valuable inputs. We are also grateful to one of our referees who suggested that "same generation" and "alternating paths" could be expressed using a single application of *sri*.

References

[1] S. Abiteboul, R. Hull and V. Vianu. *Foundations of Databases.* Addison Wesley, 1995.

[2] S. Abiteboul and P. Kanellakis. Query languages for complex object databases. *SIGACT News*, 21(3):9–18, 1990.

[3] A. Aho and J. Ullman. Universality of data retrieval languages. In *Proceedings 6th Symposium on Principles of Programming Languages, Texas, January 1979*, pages 110–120, 1979.

[4] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[5] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, September 1995.

[6] J.-Y. Cai. Lower bound for constant-depth circuits in the presence of help bits. *Information Processing Letters*, 36:79–83, 1990.

[7] G. Dong, L. Libkin, and L. Wong. On impossibility of decremental recomputation of recursive queries in relational calculus and SQL. In *Proceedings of 5th International Workshop on Database Programming Languages, Gubbio, Italy, September 1995*, Springer Electronic Workshops in Computing, 1996. Available at http: //www.springer.co.uk /eWiC /Workshops /DBPL5.html.

[8] G. Dong, L. Libkin, and L. Wong. Local properties of query languages. In *Proceedings of 6th International Conference on Database Theory*, pages 140–154, Delphi, Greece, January 1997.

[9] G. Dong and J. Su. Incremental and decremental evaluation of transitive closure by first-order queries. *Information and Computation*, 120(1):101–106, July 1995.

[10] G. Dong and J. Su. Space-bounded FOIES. In *Proceedings of 14th ACM Symposium on Principles of Database Systems, San Jose, California*, pages 139–150, May 1995.

[11] G. Dong and J. Su. Deterministic FOIES are strictly weaker. Technical Report 96/27, Department of Computer Science, University of Melbourne, 1996.

[12] G. Dong, J. Su, and R. Topor. Nonrecursive incremental evaluation of Datalog queries. *Annals of Mathematics and Artificial Intelligence*, 14:187–223, 1995.

[13] G. Dong and L. Wong. Some relationships between FOIES and Σ_1^1 arity hierarchies. *Bulletin of EATCS*, 61:72–79, 1997.

[14] N. Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16:760–778, 1987.

[15] L. Libkin and L. Wong. Aggregate functions, conservative extension, and linear orders. In C. Beeri, A. Ohori, and D. Shasha, editors, *Proceedings of 4th International Workshop on Database Programming Languages, New York, August 1993*, pages 282–294. Springer-Verlag, January 1994.

[16] L. Libkin and L. Wong. Conservativity of nested relational calculi with internal generic functions. *Information Processing Letters*, 49(6):273–280, March 1994.

[17] L. Libkin and L. Wong. New techniques for studying set languages, bag languages, and aggregate functions. In *Proceedings of 13th ACM Symposium on Principles of Database Systems*, pages 155–166, Minneapolis, Minnesota, May 1994. Full version to appear in *JCSS*.

[18] L. Libkin and L. Wong. Incremental recomputation of recursive queries with nested sets and aggregate functions. Technical Report 97-224-0, Institute of Systems Science, Heng Mui Keng Terrace, Singapore 119597, April 1997.

[19] J. Paredaens and D. Van Gucht. Converting nested relational algebra expressions into flat algebra expressions. *ACM Transaction on Database Systems*, 17(1):65–93, March 1992.

[20] S. Patnaik and N. Immerman. Dyn-FO: A parallel dynamic complexity class. In *Proceedings of 13th ACM Symposium on Principles of Database Systems*, pages 210–221, Minneapolis, Minnesota, May 1994.

[21] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[22] L. Wong. *Querying Nested Collections*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, August 1994. Available as University of Pennsylvania IRCS Report 94-09.

[23] L. Wong. Normal forms and conservative extension properties for query languages over collection types. *Journal of Computer and System Sciences*, 52(3):495–505, June 1996.