On the Power of Incremental Evaluation in SQL-like Languages

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

Limsoon Wong†
Kent Ridge Digital Labs
21 Heng Mui Keng Terrace
Singapore 119613
Email: `limsoon@krdl.org.sg`

1 Summary

There are two kinds of incremental query evaluation in general. The first kind is where a query is definable in the ambient language. In this case, incremental evaluation is possible and the main problem is to find efficient algorithms to perform it [12, 13, etc.] The second kind is where a query is *not* definable in the ambient language, and it is the main interest of this paper. The main questions addressed in this setting deal with conditions under which it is possible to evaluate queries incrementally.

Let us motivate this second kind of incremental query evaluation by a very simple example using the relational calculus (first-order logic) as the ambient language. Let PARITY be the query that returns true iff the cardinality of a set X is even. This query cannot be expressed in relational calculus, but it can be incrementally evaluated. Indeed, on the insertion of an x into X, one replaces the current answer to PARITY by its negation if $x \notin X$, and keeps it intact if $x \in X$. On the deletion of an x from X, one negates the current answer if $x \in X$, and keeps the answer unchanged if $x \notin X$. Clearly, this algorithm is first-order definable.

We denote the class of queries that can be incrementally evaluated in a language $\mathcal{L}$, using auxiliary relations of arity up to k, $k > 0$, by $\mathsf{IES}(\mathcal{L})_k$. We let $\mathsf{IES}(\mathcal{L})_\epsilon$ be the class of queries incrementally evaluated in $\mathcal{L}$ without using any auxiliary data (like the PARITY example above). Finally, $\mathsf{IES}(\mathcal{L})$ is the union of all $\mathsf{IES}(\mathcal{L})_k$.

The most frequently considered class is $\mathsf{IES}(\mathcal{FO})$, which uses the relational calculus as its ambient language. There are several examples of queries belonging to $\mathsf{IES}(\mathcal{FO})$ that are not definable in $\mathcal{FO}$ [21, 7]. The most complex example is probably that of [9], which is a query that is in $\mathsf{IES}(\mathcal{FO})$ but cannot be expressed even in first-order logic enhanced with counting and transitive closure operators. It is known [7] that the arity hierarchy is strict: $\mathsf{IES}(\mathcal{FO})_k \subset \mathsf{IES}(\mathcal{FO})_{k+1}$, and that $\mathsf{IES}(\mathcal{FO}) \subseteq$ PTIME. Still, for most queries of interest, such as the transitive closure of a relation, it remains open whether they belong to $\mathsf{IES}(\mathcal{FO})$. It also appears [9] that proving lower bounds for $\mathsf{IES}(\mathcal{FO})$ is as difficult as proving some circuit lower bounds.

*Part of this work was done while visiting INRIA and Kent Ridge Digital Labs.
†Part of this work was done while visiting Bell Labs.

Most commercial database systems speak SQL and most practical implementations of SQL are more expressive than the relational algebra because they have aggregate functions (e.g., `AVG`, `TOTAL`) and grouping constructs (`GROUPBY`, `HAVING`). This motivated us [19] to look at incremental evaluation systems based on the "core" of SQL, which comprises relational calculus plus grouping and aggregation. Somewhat surprisingly, we discovered the following. First, queries such as the transitive closure and even some PTIME-complete queries, can be incrementally evaluated by core SQL queries (although the algorithms presented in [19] were quite ad hoc). Second, the arity hierarchy for core SQL collapses at the second level.

Our goal here is to investigate deeper into the incremental evaluation capabilities of SQL-like languages. In particular, we want to find nice descriptions of classes of queries that can be incrementally evaluated. The first set of results shows that the classes are indeed much larger than we suspected before. We define a language $\mathcal{SQL}$ that extends relational algebra with grouping and aggregation, and show that:

1. Every query whose data complexity is in the polynomial hierarchy (equivalently: every second-order definable query) is in $\mathsf{IES}(\mathcal{SQL})$.
2. There exists PSPACE-complete queries in $\mathsf{IES}(\mathcal{SQL})$.
3. Adding deterministic transitive closure to $\mathcal{SQL}$ (a DLOGSPACE operator) results in a language that can incrementally evaluate every query of PSPACE data complexity.

In the second part of the paper, we compare the IES hierarchy in the cases of ordered and unordered types. We show that the $\mathsf{IES}(\mathcal{SQL})_k$ hierarchy collapses at level 1 in the case of ordered types. We further paint the complete picture of the relationship between the classes of the ordered and the unordered hierarchies; see Figure 2.

As one might expect, the reason for the enormous power of SQL-like languages in terms of incremental evaluation is that one can create and maintain rather large structures on *numbers* and use them for coding queries. In some cases, this can be quite inefficient. However, we have demonstrated elsewhere [6] that coding an algorithm for incremental evaluation of transitive closure in SQL is reasonably simple. Moreover, it has also been shown [22] that the performance is adequate for a large class of graphs. Thus, while the proofs here in general do not lend themselves to efficient algorithms (nor can they, as we show how to evaluate presumably intractable queries), the incremental techniques can well be used in practice. However, proving that certain queries cannot be incrementally evaluated in SQL within some complexity bounds appears beyond reach, as doing so would separate some complexity classes, cf. [15].

Organization In the next section, we give preliminary material, such as a theoretical language $\mathcal{SQL}$ capturing the grouping and aggregation features of SQL, the definition of incremental evaluation system IES, a nested relational language, and the relationship between the incremental evaluation systems based on the nested language and aggregation.

In Section 3, we prove that $\mathsf{IES}(\mathcal{SQL})$, the incremental evaluation system based on core SQL, includes every query whose data complexity is in the polynomial hierarchy. We also give an example of a PSPACE-complete query which belongs to $\mathsf{IES}(\mathcal{SQL})$, and show that $\mathcal{SQL}$ augmented with the deterministic transitive closure operator can incrementally evaluate every query of PSPACE data complexity.

In Section 4, we consider a slightly different version of $\mathcal{SQL}$, denoted by $\mathcal{SQL}^<$. In this language, base types come equipped with an order relation. We show that the $\mathsf{IES}(\mathcal{SQL}^<)_k$ hierarchy collapses at the first level, and explain the relationship between the classes in both $\mathsf{IES}(\mathcal{SQL})_k$ and $\mathsf{IES}(\mathcal{SQL}^<)_k$ hierarchies.

2 Preliminaries

Languages $\mathcal{SQL}$ and $\mathcal{NRC}$

A functional-style language that captures the essential features of SQL (grouping and aggregation) has been studied in a number of papers [18, 5, 15]. While the syntax slightly varies, choosing any particular one will not affect our results, as the expressive power is the same. Here we work with the version presented in [15].

The language is defined as a suitable restriction of a *nested* language. The type system is given by

$$\begin{array}{rcl} \textsc{Base} & := & b \mid \mathbb{Q} \\ rt & := & \textsc{Base} \times \ldots \times \textsc{Base} \\ t & := & \mathbb{B} \mid rt \mid \{rt\} \mid t \times \ldots \times t \end{array}$$

The base types are b and $\mathbb{Q}$, with the domain of b being an infinite set $\mathcal{U}$, disjoint from $\mathbb{Q}$. We use $\times$ for product types; the semantics of $t_1 \times \ldots \times t_n$ is the cartesian product of domains of types $t_1, \ldots, t_n$. The semantics of $\{t\}$ is the finite powerset of elements of type t. We use the notation rt for record types, and let $\mathbb{B}$ be the Boolean type.

A database schema σ is a collection of relation names and their types of the form $\{rt\}$. For a relation $R \in \sigma$, we denote its type by $\mathrm{tp}_\sigma(R)$. Expressions of the language over a fixed relational schema σ are shown in Figure 1. We adopt the convention of omitting the explicit type superscripts in these expressions whenever they can be inferred from the context. We briefly explain the semantics here. The set of free variables of an expression e is defined in a standard way by induction on the structure of e and we often write $e(x_1, \ldots, x_n)$ to explicitly indicate that x_1, ..., x_n are free variables of e. Expressions $\bigcup\{e_1 \mid x \in e_2\}$ and $\sum\{e_1 \mid x \in e_2\}$ bind the variable x (furthermore, x is not allowed to be free in e_2 for this expression to be well-formed).

For each fixed schema σ and an expression $e(x_1, \ldots, x_n)$, the value of $e(x_1, \ldots, x_n)$ is defined by induction on the structure of e and with respect to a σ-database D and a substitution $[x_1 := a_1, \ldots, x_n := a_n]$ that assigns to each variable x_i a value a_i of the appropriate type. We write $e[x_1 := a_1, \ldots, x_n := a_n](D)$ to denote this value; if the context is understood, we shorten this to $e[x_1 := a_1, \ldots, x_n := a_n]$ or just e. We have equality test on both base types. On the rationals, we have the order and the usual arithmetic operations. There is the tupling operation $(e_1, \ldots, e_n)$ and projections $\pi_{i,n}$ on tuples. The value of $\{e\}$ is the singleton set containing the value of e; $e_1 \cup e_2$ computes the union of two sets, and $\emptyset$ is the empty set.

To define the semantics of $\bigcup$ and $\sum$, assume that the value of e_2 is the set $\{b_1, \ldots, b_m\}$. Then the value of $\bigcup\{e_1 \mid x \in e_2\}$ is defined to be $\bigcup_{i=1}^m e_1[x_1{:=}a_1, \ldots, x_n{:=}a_n, x{:=}b_i](D)$. The value of $\sum\{e_1 \mid x \in e_2\}$

is $c_1 + \ldots + c_m$, each c_i is the value of $e_1[x_1{:=}a_1, \ldots, x_n{:=}a_n, x{:=}b_i]$, $i = 1, \ldots, m$.

$$\frac{}{x^t : t} \qquad \frac{R \in \sigma}{R : \mathrm{tp}_\sigma(R)} \qquad \frac{}{0, 1 : \mathbb{Q}} \qquad \frac{e_1, e_2 : \mathbb{Q}}{e_1 + e_2,\ e_1 - e_2,\ e_1 * e_2,\ e_1 \div e_2 : \mathbb{Q}}$$

$$\frac{e_1, e_2 : b}{= (e_1, e_2) : \mathbb{B}} \qquad \frac{e_1, e_2 : \mathbb{Q}}{= (e_1, e_2) : \mathbb{B}} \qquad \frac{e_1, e_2 : \mathbb{Q}}{< (e_1, e_2) : \mathbb{B}} \qquad \frac{e : \mathbb{B} \quad e_1 : t \quad e_2 : t}{\mathit{if}\ e\ \mathit{then}\ e_1\ \mathit{else}\ e_2 : t}$$

$$\frac{e_1 : t_1 \quad \ldots \quad e_n : t_n}{(e_1, \ldots, e_n) : t_1 \times \ldots \times t_n} \qquad \frac{i \leq n \quad e : t_1 \times \ldots \times t_n}{\pi_{i,n}\ e : t_i}$$

$$\frac{e : rt}{\{e\} : \{rt\}} \qquad \frac{e_1 : \{rt\} \quad e_2 : \{rt\}}{e_1 \cup e_2 : \{rt\}} \qquad \frac{}{\emptyset^{rt} : \{rt\}}$$

$$\frac{e_1 : \{rt_1\} \quad e_2 : \{rt_2\}}{\bigcup\{e_1 \mid x^{rt_2} \in e_2\} : \{rt_1\}} \qquad \frac{e_1 : \mathbb{Q} \quad e_2 : \{rt\}}{\sum\{e_1 \mid x^{rt} \in e_2\} : \mathbb{Q}}$$

Figure 1: Expressions of $\mathcal{SQL}$ over schema σ

Properties of $\mathcal{SQL}$ The relational part of the language (without arithmetic and aggregation) is known [18, 3] to have essentially the power of the *relational algebra*. When the standard arithmetic and the $\sum$ aggregate are added, the language becomes [18] powerful enough to code standard SQL aggregation features such as the `GROUPBY` and `HAVING` clauses, and aggregate functions such as `TOTAL`, `COUNT`, `AVG`, `MIN`, `MAX`, which are present in all commercial versions of SQL [1].

Another language that we frequently use is the nested relational calculus $\mathcal{NRC}$. Its type system is given by

$$t \ := \ b \mid \ \mathbb{B} \mid \ t \times \ldots \times t \mid \ \{t\}$$

That is, sets nested arbitrarily deep are allowed. The expressions of $\mathcal{NRC}$ are exactly the expressions of $\mathcal{SQL}$ that do not involve arithmetic, except that there is no restriction to flat types in the set operations.

Incremental evaluation systems

The idea of an incremental evaluation system, or IES, is as follows. Suppose we have a query Q and a language $\mathcal{L}$. An $\mathsf{IES}(\mathcal{L})$ for incrementally evaluating Q is a system consisting of an input database, an answer database, an optional auxiliary database, and a finite set of "update" functions that correspond to different kinds of permissible updates to the input database. These update functions take as input the corresponding update, the input database, the answer database, and the auxiliary database; and collectively produce as output the updated input database, the updated answer database, and the updated auxiliary database. There are two main requirements: the condition $O = Q(I)$ must be maintained, where I is the input database, and O is the output database; and that the update functions must be expressible in the language $\mathcal{L}$. For example, in the previous section we gave an incremental evaluation system for the PARITY query in relational calculus. That system did not use any auxiliary relations.

Following [21, 7, 8, 19], we consider here only queries that operate on relational databases storing elements of the base type b. These queries are those whose inputs are of types of the form $\{b \times \ldots \times b\}$. Queries whose incremental evaluation we study have to be generic, that is, invariant under permutations of the domain $\mathcal{U}$ of type b. Examples include all queries definable in a variety of classical query languages, such as relational calculus, datalog, and the while-loop language. The criteria for permissible update are restricted to the insertion and deletion of a single tuple into an input relation.

While the informal definition given above is sufficient for understanding the results of the paper, we give a formal definition of $\mathsf{IES}(\mathcal{L})$, as in [19], which is very similar to the definitions of FOIES [7] and Dyn-$\mathcal{C}$ [21]. Suppose the types of relations of the input database are $\{rt_1\}, \ldots, \{rt_m\}$, where $rt_1, \ldots, rt_m$ are record types of the form $b \times \ldots \times b$. We consider elementary updates of the form $ins_i(x)$ and $del_i(x)$, where x is of type rt_i. Given an object X of type $S = \{rt_1\} \times \ldots \times \{rt_m\}$, applying such an update results in inserting x into or deleting x from the ith set in X, that is, the set of type $\{rt_i\}$. 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 $S \to T$ (that is, an expression of type T with free variables of types $\{rt_1\}, \ldots, \{rt_m\}$), and a type T_{aux} (of auxiliary data), consider a collection of functions $\mathcal{F}_Q$:

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

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

$$\begin{array}{rcl} X_{i+1} & = & u_{i+1}(X_i) \\ RES_{i+1} & = & f_{u_{i+1}}(X_i, RES_i, AUX_i) \\ AUX_{i+1} & = & f_{u_{i+1}}^{\text{aux}}(X_i, RES_i, AUX_i) \end{array}$$

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* $\mathsf{IES}(\mathcal{L})$ or *maintainable in* $\mathcal{L}$. If T_{aux} is a product of flat types $\{rt\}$, with rts having at most k components, then we say that Q is in $\mathsf{IES}(\mathcal{L})_k$.

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

Properties of IES Clearly, every query expressible in $\mathcal{L}$ belongs to $\mathsf{IES}(\mathcal{L})_\epsilon$. What makes IES interesting is that many queries that are not expressible in $\mathcal{L}$ can still be incrementally evaluated in $\mathcal{L}$. For example, the transitive closure of undirected graphs belongs to $\mathsf{IES}(\mathcal{FO})_2$ [21, 7]. One of the more remarkable facts about $\mathsf{IES}(\mathcal{FO})$, mentioned already in the introduction, is that the arity hierarchy is strict: $\mathsf{IES}(\mathcal{FO})_k \subsetneq \mathsf{IES}(\mathcal{FO})_{k+1}$ [7]. Also, every query in $\mathsf{IES}(\mathcal{FO})$ has PTIME data complexity.

A number of results about $\mathsf{IES}(\mathcal{SQL})$ exist in the literature. We know [4] that $\mathcal{SQL}$ is unable to maintain transitive closure of arbitrary graphs without using auxiliary relations. We also know that transitive closure of arbitrary graphs remains unmaintainable in $\mathcal{SQL}$ even in the presence of auxiliary data whose degrees are bounded by a constant [5]. On the positive side, we know that if the bounded degree constraint on auxiliary data is removed, transitive closure of arbitrary graphs becomes maintainable in $\mathcal{SQL}$. In fact, this query and even the alternating path query belong to $\mathsf{IES}(\mathcal{SQL})_2$. Finally, we also know [19] that the $\mathsf{IES}(\mathcal{SQL})_k$ hierarchy collapses to $\mathsf{IES}(\mathcal{SQL})_2$. We shall use the following result [19] several times in this paper.

Fact 2.1 $\mathsf{IES}(\mathcal{NRC}) \subseteq \mathsf{IES}(\mathcal{SQL})$. □

3 Maintainability of Second Order Queries

We prove in this section that we can incrementally evaluate all queries whose data complexity is in the polynomial hierarchy PHIER (equivalently, all queries expressible in second order logic). The proof, sketched at the end of the section, is based on the ability to maintain very large sets using arithmetic, which suffices to model second-order expressible queries.

Theorem 3.1 *$\mathcal{SQL}$ can incrementally evaluate all queries whose data complexity is in the polynomial hierarchy. That is, PHIER $\subseteq$ $\mathsf{IES}(\mathcal{SQL})$.* □

The best previously known [19] positive result on the limit of incremental evaluation in $\mathcal{SQL}$ was for a PTIME-complete query. Theorem 3.1 shows that the class of queries that can be incrementally evaluated in $\mathcal{SQL}$ is presumably much larger than the class of tractable queries. In particular, every NP-complete problem is in $\mathsf{IES}(\mathcal{SQL})$.

The next question is whether the containment can be replaced by equality. This appears unlikely in view of the following.

Proposition 3.2 *There exists a problem complete for PSPACE which belongs to $\mathsf{IES}(\mathcal{SQL})$.*

Note that this is *not* sufficient to conclude the containment of PSPACE in $\mathsf{IES}(\mathcal{SQL})$, as the notion of reduction for dynamic complexity classes is more restrictive than the usual reduction notions in complexity theory, see [21]. In fact, we do not know if PSPACE is contained in $\mathsf{IES}(\mathcal{SQL})$. We can show, however, that a mild extension of $\mathcal{SQL}$ gives us a language powerful enough to incrementally evaluate all PSPACE queries. Namely, consider the following addition to the language:

$$\frac{e : \{rt \times rt\}}{dtc(e) : \{rt \times rt\}}$$

Here *dtc* is the deterministic transitive closure operator [16]. Given a graph with the set of edges E, there is an edge (a, b) in its deterministic transitive closure iff there is a deterministic path (a, a_1), (a_1, a_2), ..., (a_{n-1}, a_n), (a_n, b) in E; that is, a path in which every node a_i, $i < n$, and a have outdegree 1. It is known [16] that *dtc* is complete for DLOGSPACE. We prove the following new result.

Proposition 3.3 *$\mathcal{SQL} + dtc$ can incrementally evaluate all queries of PSPACE data complexity. That is, PSPACE $\subseteq$ $\mathsf{IES}(\mathcal{SQL} + dtc)$.* □

We now sketch the proofs of these results. We use the notation $\wp(B^k)$ to mean the powerset of the k-fold cartesian product of the set $B : \{b\}$ of atomic objects. The proof of Theorem 3.1 involves two steps. In the first step, we show that $\wp(B^k)$ can be maintained in $\mathcal{NRC}$ for every k, when B is updated. In the second step, we show that if the domain of each second order quantifier is made available to $\mathcal{NRC}$, then any second order logic formula can be translated to $\mathcal{NRC}$. The first of these two steps is also needed for the proof of Propositions 3.3 and 3.2, so we abstract it out in the following lemma.

Lemma 3.4 *$\mathcal{NRC}$ can incrementally evaluate $\wp(B^k)$ for every k when $B : \{b\}$ is updated.*

Proof sketch. Let PB_k^o and PB_k^n be the symbols naming the nested relation $\wp(B^k)$ immediately before and after the update. We proceed by induction on k. The simple base case of $k = 1$ (maintaining the powerset of a unary relation) is omitted. For the induction case of $k > 1$, we consider two cases.

Suppose the update is the insertion of a new element x into the set B. By the induction hypothesis, $\mathcal{NRC}$ can maintain $\wp(B^{k-1})$. So we can create the following nested sets: $Y_0 = \{\{(x, \ldots, x)\}\}$ and $Y_i = \{\{(z_1, \ldots, z_i, x, z_{i+1}, \ldots, z_{k-1}) \mid (z_1, \ldots, z_{k-1}) \in X\} \mid X \in PB_{k-1}^n\}$, for $i = 1$, ..., $k-1$. Let *cartprod* be the function that forms the cartesian product of two sets; this function is easily definable in $\mathcal{NRC}$. Let *allunion* be the function that takes a tuple $(S_1, \ldots, S_k)$ of sets and returns a set of sets containing all possible unions of S_1, ..., S_k; this function is also definable in $\mathcal{NRC}$ because the number of combinations is fixed once k is given. Then it is not difficult to see that $PB_k^n = \{X \mid Y \in (PB_k^o\ cartprod\ Y_0\ cartprod\ Y_1\ cartprod \cdots\ cartprod Y_{k-1}), X \in allunion(Y)\}$.

Suppose the update is the deletion of an existing element x from the set B. Then all we need is to delete from each of PB_1, ..., PB_k all the sets that have x as a component of one of their elements, which is definable in $\mathcal{NRC}$. □

Proof sketch of Theorem 3.1. Let $Q : \{rt\}$ be a query in PHIER, with input relations $R_1, \ldots, R_m$ of types $\{rt_i\}$. Then Q is definable by a second-order formula with n free first-order variables, where n is the arity of rt. Suppose this formula is $\phi(\vec{x}) = \mathbf{Q}_1 S_1 \ldots \mathbf{Q}_p S_p \alpha(\vec{x}, S_1, \ldots, S_p)$; where α is a first-order formula in the language of R_is, S_is, and equality; $\mathbf{Q}$s are the quantifiers $\forall$ and $\exists$; and each S_i has arity k_i. Then, to maintain Q in $\mathcal{NRC}$, we have to maintain: (a) the active domain B of the database $R_1, \ldots, R_m$, and (b) all $\wp(B^{k_i})$. Note that the definition of $\mathsf{IES}(\mathcal{NRC})$ puts no restriction on types of auxiliary relations. Since a single insertion into or deletion from a relation R_i results in a fixed number of insertions and deletions in B that is bounded by the maximal arity of a relation, we conclude from Lemma 3.4 that all $\wp(B^{k_i})$ can be incrementally evaluated. Since $\mathcal{NRC}$ has all the power of first-order logic [3], we conclude that it can incrementally evaluate Q by maintaining all the powersets and then evaluating a first-order query on them. □

Proof sketch of Proposition 3.2. It is not hard to show that with $\wp(B^k)$, one can incrementally evaluate the REACHABLE DEADLOCK problem, which is known to be PSPACE-complete [20].

Proof sketch of Proposition 3.3. Let Q be a PSPACE query. It is known then that Q is expressible in partial-fixpoint logic, if the underlying structure is ordered. We know [19] that an order relation on the active domain can be maintained in $\mathcal{SQL}$. We also know [2] that Q is of the form $\mathrm{PFP}_{\vec{y},S}\phi(\vec{x},\vec{y},S)$, where ϕ is a first-order formula. To show that Q is in $\mathsf{IES}(\mathcal{SQL} + dtc)$ we do the following. We maintain the active domain B, an order relation on it, and $\wp(B^k)$ where $k = |\vec{y}|$. We maintain it, however, as a *flat* relation of type $\{\mathbb{Q} \times b \times \ldots \times b\}$ where subsets are coded; that is, a tuple $(c, \vec{a})$ indicates that $\vec{a}$ belongs to a subset of B^k coded by c. That this can be done, follows from the proof of $\mathsf{IES}(\mathcal{NRC}) \subseteq \mathsf{IES}(\mathcal{SQL})$ in [19]. We next define a binary relation R_0 of type $\{\mathbb{Q} \times \mathbb{Q}\}$ such that a pair (c_1, c_2) is in it if applying the operator defined by ϕ to the subset of B^k coded by c_1 yields c_2. It is routine to verify that this is definable. Next, we note that the outdegree of every node of R_0 is at most 1; hence, $dtc(R_0)$ is its transitive closure. Using this, we can determine the value of the partial fixpoint operator. □

Limitations of Incremental Evaluation in $\mathcal{SQL}$

Having captured the whole of the polynomial hierarchy inside $\mathsf{IES}(\mathcal{SQL})$, can we do more? Proving lower bounds in the area of dynamic complexity is very hard [21, 9] and $\mathcal{SQL}$ is apparently no exception. Still, we can establish some easy limitations. More precisely, we address the following question. We saw that the powerset of B^k can be incrementally evaluated in $\mathcal{NRC}$. Does this continue to hold for *iterated* powerset constructions? For example, can we maintain sets like $\wp(\wp(B^k))$, $\wp(\wp(B)\ \mathit{cartprod}\ \wp(B))$, etc.? If we could maintain $\wp(\wp(B^k))$ in $\mathcal{NRC}$, it would have shown that PSPACE is contained in $\mathsf{IES}(\mathcal{SQL})$. However, it turns out the Lemma 3.4 is close to the limit. First, we note the 2-DEXPSPACE data complexity of $\mathsf{IES}(\mathcal{SQL})$.

Proposition 3.5 *For every query in $\mathsf{IES}(\mathcal{SQL})$ (even without restriction to flat types) there exist numbers $c, d > 0$ such that the total size of the input database, answer database, and auxiliary database after n updates is at most c^{d^n}.*

Proof. It is known that $\mathcal{SQL}$ queries have PTIME data complexity [18]. Thus, if $f(n)$ is the size of the input, output and auxiliary databases after n updates, we obtain $f(n+1) \leq Cf(n)^m$ for appropriately chosen $C, m > 0$. The claim now follows by induction on n. □

We use $\wp^j(B^k)$ to mean taking the powerset j times on the k-fold cartesian product of the set B of atomic objects. We know that $\wp(B^k)$ can be maintained by $\mathcal{NRC}$. For the iterated case, not much can be done.

Corollary 3.6 *Let $j > 1$. $\wp^j(B^k)$ can be maintained by $\mathcal{NRC}$ when B is updated iff $j = 2$ and $k = 1$.*

Proof sketch. First, we show that $\wp^2(B)$ can be maintained. Let $B : \{b\}$ denote the input database. Let $PPB = \wp(\wp(B)) : \{\{\{b\}\}\}$ denote the answer database. B is initially empty. PPB is initially $\{\{\}, \{\{\}\}\}$. Suppose the update is the insertion of a new atomic object x into B. Let $\Delta = \{U \cup \{\{x\} \cup v \mid v \in V\} \mid U \in PPB^o,\ V \in PPB^o\}$. Then $PPB^n = PPB^o \cup \Delta$ is the desired double powerset.

$$\begin{tikzcd}
\mathcal{SQL}^{<} \arrow[r, hook, "2"] \arrow[dr, dotted, leftrightarrow, "9"] & \mathsf{IES}(\mathcal{SQL}^{<})_{\epsilon} \arrow[r, hook, "3"] \arrow[dr, dotted, leftrightarrow, "11"] & \mathsf{IES}(\mathcal{SQL}^{<})_{1} \arrow[r, equal, "4"] & \mathsf{IES}(\mathcal{SQL}^{<})_{k\geq 1} \arrow[r, phantom, "\overset{13}{\supseteq}"] \arrow[d, equal, "14"'] & \mathrm{PHIER} \\
\mathcal{SQL} \arrow[r, hook, "5"] \arrow[u, hook, "1"] & \mathsf{IES}(\mathcal{SQL})_{\epsilon} \arrow[r, hook, "6"] \arrow[u, hook, "10"] & \mathsf{IES}(\mathcal{SQL})_{1} \arrow[r, hook, "7"] \arrow[u, hook, "12"] & \mathsf{IES}(\mathcal{SQL})_{2} \arrow[r, equal, "8"] & \mathsf{IES}(\mathcal{SQL})_{k\geq 2}
\end{tikzcd}$$

Figure 2: $\mathsf{IES}(\mathcal{SQL})_k$ and $\mathsf{IES}(\mathcal{SQL}^{<})_k$ hierarchies

Suppose the update is the deletion of an old object x from B. Then we simply delete from PPB all those sets that mention x. Both operations are definable in $\mathcal{NRC}$.

That $\wp^j(B^k)$ cannot be maintained for $(j,k) \neq (2,1)$, easily follows from the bounds above, as $2^{2^{n^2}}$ is not majorized by c^{d^n} for any constants c, d. $\square$

4 Low Levels of the IES hierarchy

We know that the class of queries that can be evaluated incrementally in $\mathcal{SQL}$ is very large. We also know from earlier work [4, 19] that with restrictions on the class of auxiliary relations, even many PTIME queries cannot be maintained. Thus, we would like to investigate the low levels of the $\mathsf{IES}(\mathcal{SQL})$ hierarchy. This was partly done in [19], under a severe restriction that only elements of base types be used in auxiliary relations. Now, using recent results on the expressive power of SQL-like languages and locality tools from finite-model theory [14, 15], we paint the complete picture of the relationship between the levels of the hierarchy.

In many incremental algorithms, the presence of an order is essential. While having an order on the base type b makes no difference if binary auxiliary relations are allowed (since one can maintain an order as an auxiliary relation), there is a difference for the case when restrictions on the arity of auxiliary relations are imposed. We thus consider an extension of $\mathcal{SQL}$ denoted by $\mathcal{SQL}^{<}$ which is obtained by a adding a new rule

$$\frac{e_1, e_2 : b}{<_b (e_1, e_2) : \mathbb{B}}$$

where $<_b$ is interpreted as an order on the domain of the base type b. The main result now relates the levels of the $\mathsf{IES}(\mathcal{SQL})_k$ and $\mathsf{IES}(\mathcal{SQL}^{<})_k$ hierarchies.

Theorem 4.1 *The relationships shown in the diagram in Figure 2 hold. Here $A \hookrightarrow B$ means that A is a proper subset of B, and $A \leftarrow\cdots\rightarrow B$ means that $A \not\subseteq B$ and $B \not\subseteq A$.*

Proof sketch. The containment 13 was shown in this paper (Theorem 3.1). The hierarchy collapse 8, as well as the inclusion 6 and the maintenance of order 14 are from [19]. We also note that in $\mathcal{SQL}$, one can incrementally evaluate a query q_0 such that $q_0(D) = 2^n$, where n is the size of the active domain of D. However, it is known that the maximal number $\mathcal{SQL}$ or $\mathcal{SQL}^<$ can produce is at most polynomial in the size of the active domain and the maximal number stored in the database. This shows inclusions 2, 5 and half of 9: $\mathsf{IES}(\mathcal{SQL})_\epsilon \not\subset \mathcal{SQL}^<$.

Next, consider an input of type $\{b\}$, and a query

$$q_1(X) \;=\; \begin{cases} 2^{|X|} & \text{if } |X| \text{ is a power of 2} \\ 0 & \text{otherwise} \end{cases}$$

This query belongs to $\mathsf{IES}(\mathcal{SQL})_1$, as we can maintain the set $\{0, 1, 2, \ldots, 2^{|X|}\}$ and then use standard techniques to test for the powers of 2. However, $q_1 \notin \mathsf{IES}(\mathcal{SQL}^<)_\epsilon$. Indeed, if $|X| = 2^m - 1$, then $q_1(X) = 0$ and thus on an insert into X, the maintenance query would have to produce an integer exponential in the size of the input. This shows 3, 6, and half of 11: $\mathsf{IES}(\mathcal{SQL})_1 \not\subseteq \mathsf{IES}(\mathcal{SQL}^<)_\epsilon$.

The proof of collapse 4 proceeds similarly to the proof of 8 in [19]. To reduce arity 2 to arity 1, we maintain a large enough initial segment of natural numbers (but still polynomial) which we use to code tuples by numbers, where an element of base type b is coded by its relative position in the ordering of the active domain, and tuples are coded using the standard pairing function. Then 4 and 7 imply 12.

For the remaining relationship, we use locality techniques from finite-model theory [10, 11, 14]. We shall now consider queries on tuples of flat relations of types $\{b \times \ldots \times b\}$ into a relation of type of the same form. Given an input database D, which is a tuple of relations $R_1, \ldots, R_k$, we define the Gaifman graph $\mathcal{G}(D)$ on its active domain as an undirected graph with (a, b) being an edge in it if one of R_is has a tuple that contains both a and b. By a distance in D, we mean the distance in its Gaifman graph. Given a tuple $\vec{t}$, by $S_r^D(\vec{t})$ we mean the set of all elements of the active domain of D at a distance at most r of some element of $\vec{t}$. These are *neighborhoods* of tuples, which can be considered as databases of the same schema as D, by restricting the relations of D onto them. Two tuples are said to have the same r-type if their r-neighborhoods are isomorphic. That is, there is a bijection $f : S_r^D(\vec{t}_1) \to S_r^D(\vec{t}_2)$ such that $f(\vec{t}_1) = \vec{t}_2$ and for every tuple $\vec{u}$ of elements of $S_r^D(\vec{t}_1)$, $\vec{u} \in R_i$ implies $f(\vec{u}) \in R_i$, and for every $\vec{v}$ in $S_r^D(\vec{t}_2)$, $\vec{v} \in R_i$ implies $f^{-1}(\vec{v}) \in R_i$.

We now say (see [14], where connection with Gaifman's theorem [11] is explained) that a query Q is *local* if there exists an integer r such that, if $\vec{t}_1$ and $\vec{t}_2$ have the same r-type in D, then $\vec{t}_1 \in Q(D)$ iff $\vec{t}_2 \in Q(D)$. We shall use the fact [15] that every query of pure relational type (no rationals) in $\mathcal{SQL}$ is local.

Now 1 follows from locality of $\mathcal{SQL}$, and the fact that $\mathcal{SQL}^<$ expresses all queries definable in first-order logic with counting over ordered structures (see [15]), which is known to violate locality [14]. For other relationships, consider the following query. Its input type is $\{b \times b\} \times \{b\}$; its output is of type $\{b\}$. We shall refer to the graph part of the input as G and to the set part as P; that is, the input is a pair (G, P). A pair is *good* if G is the graph of a successor relation, and P is its initial segment. A query q is good if it has the following properties whenever its input is good: (1) If $n = 2^{|P|}$, where n is the number of nodes in G, then $q(G, P)$ is the transitive closure of the initial segment defined by P; (2) If $n \neq 2^{|P|}$, then $q(G, P) = \emptyset$. It can be shown that there is a good query q in $\mathcal{SQL}^<$—this is because with counting power we can encode fragments of monadic second-order on small portions of the input [14].

As the next step, we show that no such good q can belong to $\mathsf{IES}(\mathcal{SQL})_1$. This shows the second half of 11 (that $\mathsf{IES}(\mathcal{SQL}^<)_\epsilon \not\subseteq \mathsf{IES}(\mathcal{SQL})_1$), 10, 12, and second half of 9. It also shows 7, because we know $\mathcal{SQL}^< \subseteq \mathsf{IES}(\mathcal{SQL})_2$. To prove this, we first reduce the problem to inexpressibility of a good query in $\mathcal{SQL}$ in the presence of additional unary relations. This is because we can consider an input in which $2^{|P|-1} = n$. For such an input, the answer to q is $\emptyset$, but on an insert into P it becomes the transitive closure of the segment defined by P. As the next step, we show that locality of $\mathcal{SQL}$ withstands adding *numerical* relations, those of type $\{\mathbb{Q} \times \ldots \times \mathbb{Q}\}$, as long as there is no ordering on b. To prove this, we first code $\mathcal{SQL}$ into an infinitary logic with counting, as was done in [15], and then modify the induction argument from [17] to prove locality in the presence of extra numerical relations. Finally, a finite number, say m, of unary relations of type $\{b\}$, amounts to coloring nodes of a graph with 2^m colors. If we assume that q is definable with auxiliary unary relations, we fix a number r witnessing its locality, and choose n big enough so that there would be two identically colored disjoint neighborhoods of points a and b in P. This would mean that the r-types of (a, b) and (b, a) are the same, but these tuples can clearly be distinguished by q. This completes the proof. $\square$

5 Open Problems

We have shown that PHIER $\subseteq \mathsf{IES}(\mathcal{SQL})$, but it remains open whether a larger complexity class can be subsumed. One possibility is that all PSPACE queries are maintainable in $\mathcal{SQL}$. While we showed that there is a PSPACE-complete problem in $\mathsf{IES}(\mathcal{SQL})$, this does not mean that all PSPACE queries are maintainable, as IES in general is not closed under the usual reductions (polynomial or first-order), and we do not yet know of any problem complete for PSPACE under stronger reductions, defined in [21], that would belong to $\mathsf{IES}(\mathcal{SQL})$.

The proof of PHIER $\subseteq \mathsf{IES}(\mathcal{SQL})$ does not lend itself to an efficient algorithm for queries of lower complexity. In fact, it is not clear if such algorithms exist in general, and proving, or disproving their existence, is closely tied to deep unresolved problems in complexity. However, coding the maintenance algorithms for some useful queries (e.g., the transitive closure) in SQL is quite easy [6] and in fact the maintenance is quite efficient for graphs of special form [22]. Thus, while general results in this area are probably beyond reach, one could consider restrictions on classes of inputs that would lead to efficient maintenance algorithms.

References

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

[2] S. Abiteboul, V. Vianu. Computing with first-order logic. *JCSS* 50 (1995), 309–335.

[3] 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.

[4] G. Dong, L. Libkin, and L. Wong. On impossibility of decremental recomputation of recursive queries in relational calculus and SQL. In *DBPL'95.*

[5] G. Dong, L. Libkin, and L. Wong. Local properties of query languages. In *Theoretical Computer Science*, to appear. Extended abstract in *ICDT'97.*

[6] G. Dong, L. Libkin, J. Su and L. Wong. Maintaining the transitive closure of graphs in SQL. In *Int. J. Information Technology*, 1999, to appear.

[7] G. Dong and J. Su. Arity bounds in first-order incremental evaluation and definition of polynomial time database queries. *Journal of Computer and System Sciences* 57 (1998), 289–308.

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

[9] K. Etessami. Dynamic tree isomorphism via first-order updates to a relational database. In *PODS'98*, pages 235–243.

[10] R. Fagin, L. Stockmeyer, M. Vardi, On monadic NP vs monadic co-NP, *Information and Computation*, 120 (1994), 78–92.

[11] H. Gaifman, On local and non-local properties, *in* "Proceedings of the Herbrand Symposium, Logic Colloquium '81," North Holland, 1982.

[12] A. Gupta, I. S. Mumick and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD'93*, pages 157–166.

[13] A. Gupta and I.S. Mumick. Maintenance of materialized views: problems, techniques, and applications. *Data Engineering Bulletin* 18 (1995), 3–18.

[14] L. Hella, L. Libkin and J. Nurmonen. Notions of locality and their logical characterizations over finite models. *J. Symb. Logic*, to appear.

[15] L. Hella, L. Libkin, J. Nurmonen and L. Wong. Logics with aggregate operators. In *LICS'99*, to appear.

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

[17] L. Libkin. On counting logics and local properties. In *LICS'98*, pages 501-512.

[18] L. Libkin and L. Wong. Query languages for bags and aggregate functions. *Journal of Computer and System Sciences* 55 (1997), 241–272.

[19] L. Libkin and L. Wong. Incremental recomputation of recursive queries with nested sets and aggregate functions. *Proceedings of Database Programming Languages 97*, Estes Park, Colorado, (S. Cluet and R. Hull, eds.), Springer LNCS volume 1369, pages 222–238.

[20] C. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.

[21] S. Patnaik and N. Immerman. Dyn-FO: A parallel dynamic complexity class. *Journal of Computer and System Sciences* 55 (1997), 199–209.

[22] T.A. Schultz. ADEPT – The advanced database environment for planning and tracking. *Bell Labs Technical Journal*, 3(3):3–9, 1998.