Incremental Maintenance of Recursive Views Using Relational Calculus/SQL[*]

Guozhu Dong[†]
Department of Computer Science and Engineering
Wright State University
Dayton, Ohio 45435
Email: `gdong@cs.wright.edu`

Jianwen Su[‡]
Department of Computer Science
University of California
Santa Barbara, CA 93106
Email: `su@cs.ucsb.edu`

Abstract

Views are a central component of both traditional database systems and new applications such as data warehouses. Very often the desired views (e.g. the transitive closure) cannot be defined in the standard language of the underlying database system. Fortunately, it is often possible to incrementally maintain these views using the standard language. For example, transitive closure of acyclic graphs, and of undirected graphs, can be maintained in relational calculus after both single edge insertions and deletions. Many such results have been published in the theoretical database community. The purpose of this survey is to make these useful results known to the wider database research and development community.

There are many interesting issues involved in the maintenance of recursive views. A maintenance algorithm may be applicable to just one view, or to a class of views specified by a view definition language such as Datalog. The maintenance algorithm can be specified in a maintenance language of different expressiveness, such as the conjunctive queries, the relational calculus or SQL. Ideally, this maintenance language should be less expensive than the view definition language. The maintenance algorithm may allow updates of different kinds, such as just single tuple insertions, just single tuple deletions, special set-based insertions and/or deletions, or combinations thereof. The view maintenance algorithms may also need to maintain auxiliary relations to help maintain the views of interest. It is of interest to know the minimal arity necessary for these auxiliary relations and whether the auxiliary relations are deterministic. While many results are known about these issues for several settings, many further challenging research problems still remain to be solved.

1 Introduction

In many database applications there is a need to pose queries which cannot be defined by relational calculus and even SQL. For example, the popular transitive closure query is frequently needed and cannot be defined in these languages [2, 27]. Fortunately, in a real database system, one can try to overcome this problem by storing the answer to the queries as materialized views and maintaining the views whenever updates (e.g. the insertion or deletion of tuples) to the former occur. This idea has been investigated quite extensively under the name of an *incremental evaluation system*, or IES.

For the simplest setting, an IES maintains a view Q over a single base relation R. There are two maintenance algorithms, Q^+ and Q^-, where Q^+ maintains Q after insertion to the base relation R and Q^- maintains Q after deletion from R. Throughout this paper, we will use R^{old} to refer to the instance of R in the database *before* an update, and R^{new} the instance of R *after* the update. For each old instance R^{old} and each insertion update Δ that the IES is designed to handle, $Q(R^{\text{old}} \cup \Delta)$ can be derived from the old base relation, the old view contents, and the update using the maintenance query Q^+: $Q(R^{\text{old}} \cup \Delta) = Q^+(R^{\text{old}}, Q(R^{\text{old}}), \Delta)$. The IES handles deletions similarly.

When there are multiple views and/or multiple base relations, there will be more maintenance algorithms, one for each allowed update type and each view-base relation combination.

If an IES maintains just the given views, it is called a *space-free* IES. Sometimes, it is also necessary to

[*] Database Principles Column.
Column editor: Leonid Libkin, Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 08974. E-mail: libkin@research.bell-labs.com.

[†] Supported in part by a Research Challenge Award from Ohio Board of Regents and Wright State University.

[‡] Supported in part by NSF grants IRI-9700370 and IIS-9817432.

maintain auxiliary relations, which are maintained similarly to the way the given views are maintained. With the help of auxiliary relations, we can maintain views that cannot be maintained otherwise. Furthermore, we will see later that the arity of the auxiliary relations also makes a difference to what can be maintained.

The maintenance algorithms can be specified in different maintenance languages, such as relational calculus, SQL, nested relational algebra, Datalog, or even a host programming language. The choice of a maintenance language can be influenced by the practical constraints imposed by real systems, and it can also be influenced by efficiency issues, as some languages are more efficient or more optimizable than others. In the literature, FOIES refers to the type of IES that uses the relational calculus (first-order) language as the maintenance language. A closely related formalism is dynamic first-order, DynFO, of [30]. While DynFO is similar to FOIES in many aspects, there are some important differences between the two, see [30, 10] for comparison. Also in the literature, SQLIES refers to the type of IES that uses SQL[1] as the maintenance language. We will also use these names in this article.

Generally, an IES may be able to deal with different types of updates. Example types of updates include single-tuple insertions, single-tuple deletions, insertions/deletions of sets satisfying certain conditions, or combinations of these.

Incremental evaluation is often seen as merely a means to avoid expensive re-computation. However, from what we know of the transitive closure query (discussed in Sections 2 and 3), one can see that there is much more to the idea of incremental evaluation than just a simple view of avoiding re-computation. In particular, we see incremental evaluation also as a way to do things that could not be done otherwise. Coming back to the transitive closure, it cannot be expressed in relational databases using SQL *without incremental evaluation* but can be expressed in relational databases using SQL *in the setting of an incremental evaluation system.* In other words, avoidance of the cost of re-computation is not even the issue here, for the query is not even do-able in SQL without incremental evaluation in the first place!

After the first study [13] on the idea of an IES for maintaining a view defined by a more powerful language using maintenance algorithms written in a less powerful language, much is already known about the theory and algorithms of IES [7, 14, 12, 9, 11, 5, 26, 30, 29, etc.]. The objective of this paper is to provide a tutorial on IES and an overview of some of the results on IES. We hope that this will speed up the process of implementing these results in real systems.

This survey is organized as follows. Section 2 discusses the transitive closure of acyclic graphs. Section 3 considers the transitive closure of undirected graphs. Section 4 is about regular chain Datalog views. Section 5 is concerned with set-based updates. Section 6 considers space complexity. Section 7 reviews work on IES using SQL as the maintenance language. Section 8 sketches some other related work, including those using a host programming language as the maintenance language, either from a database perspective or from a more general algorithmic perspective. Section 9 offers some concluding remarks.

[1] By which we mean an extension of relational calculus with aggregation; that is, essentially select-from-where-groupby-having clauses plus Boolean operations.

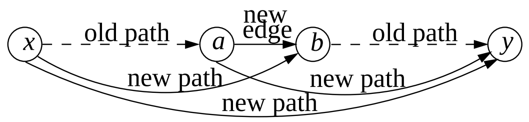
2 Transitive closure of acyclic graphs

The FOIES for maintaining the transitive closure of acyclic graphs [9, 8] is a simple illustration of the power of the FOIES model. It uses the relational calculus as the maintenance language, it handles both tuple insertions and tuple deletions, and it does not use auxiliary relations.

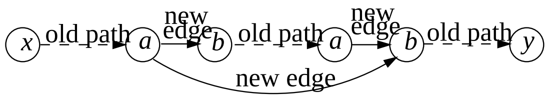
Let G represent the input graph (directed) and TC the transitive closure of G. So a tuple (x, y) is in the relation G if and only if there is a directed edge from the node x to the node y in the input graph, and a tuple (x, y) is in the relation TC if and only if there is a directed path from the node x to the node y in the input graph. An edge insertion is allowed only if this insertion does not lead to cycles in the new graph.

Maintenance after insertions

Suppose an edge (a, b) is inserted. We maintain TC as follows. Essentially, the new transitive closure is obtained by adding to the old transitive closure the following: (1) all new paths constructed by adding the new edge (a, b) to the back of an existing path ending at a, (2) all new paths constructed by adding the new edge (a, b) to the front of an existing path starting at b, (3) all new paths constructed by inserting the new edge (a, b) between an existing path ending at a and an existing path starting at b, and (4) the new edge itself. New paths added by rules (1), (2), and (3) correspond to paths of type $x \to a$, $b \to y$, and $x \to y$, respectively, shown in Figure 1(a). This covers all new paths because only one occurrence of the new edge is necessary in every new path (Fig-



(a) New paths after inserting (a, b)



(b) One occurrence of the new edge is sufficient

Figure 1: Transitive closure after an edge insertion

ure 1(b)).

Maintenance after deletions

Suppose an existing edge (a, b) is deleted. TC can be maintained as follows.

Let $S_{ab} = \{(x, y) \mid TC^{\text{old}}(x, a) \wedge TC^{\text{old}}(b, y)\}$ be the set of all paths (x, y) in the old TC which go through (a, b). It is doubtful whether these paths should belong to the new TC. (The letter S is for *suspicious.*) Let $G^{\text{new}} = G^{\text{old}} - \{(a, b)\}$ and $T_{ab} = (TC^{\text{old}} - S_{ab}) \cup G^{\text{new}}$. Each pair in T_{ab} is definitely in the new TC. (The letter T is for *trusty.*) Surprisingly, the new TC can be completely reconstructed from T_{ab} using several joins and projections given by the following formula:

$$T_{ab} \cup (T_{ab} \circ T_{ab}) \cup (T_{ab} \circ T_{ab} \circ T_{ab})$$

where $R_1 \circ R_2$ is defined as $\{(x, y) \mid \exists u(R_1(x, u) \wedge R_2(u, y)\}$ for any pair of relations R_1 and R_2.

So the new transitive closure contains (1) all *trusty* paths, (2) all paths constructed by concatenating two consecutive *trusty* paths, and (3) all paths constructed by concatenating three consecutive *trusty* paths.

The correctness of the above formula for constructing the new TC is shown in [9, 8], and the correctness relies on the following property. Suppose (x_1, x_k) is in S_{ab} and there is a path $x_1, x_2, ..., x_k$ in the new graph that does not use the edge (a, b). Then there must exist $i < k$ such that no path of the new graph from x_1 to x_i uses the edge (a, b) and no path of the new graph from x_{i+1} to x_k uses the edge (a, b).

We illustrate the maintenance of TC by considering deleting the edge (a, b) from the acyclic graph given in Figure 2.

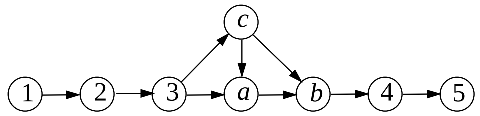


Figure 2: TC of acyclic graph after deleting (a, b)

- The contents of S_{ab} and T_{ab} are as follows:

S_{ab}

1	$b/4/5$
2	$b/4/5$
3	$b/4/5$
a	$b/4/5$
c	$b/4/5$

T_{ab}

1	$2/3/a/c$
2	$3/a/c$
3	a/c
c	a/b
b	$4/5$
4	5

  Here a pair of form $(x, u/v/w)$ stands for pairs $(x, u), (x, v), (x, w)$.

- All the paths of the new graph G^{new} are now derived from T_{ab} through zero, e.g. for the case of $(1, c)$, one, e.g. $(1, b)$, or two joins, e.g. $(1, 5)$, followed by projections. It is more instructive to visualize all the edges in the above figure other than (a, b) as a sequence of edges.

3 Transitive closure of undirected graphs

We now show how to maintain the transitive closure of undirected graphs in the relational calculus. An undirected graph contains the edge from a node y to a node x whenever it contains the edge from x to y. While the TC of acyclic graphs can be maintained without maintaining additional views, the TC of undirected graphs can only be maintained if we also maintain some *auxiliary* relations (views).

The first FOIES that maintains the transitive closure of undirected graph using nothing more than pure relational calculus was given in [30] and an improved (space-wise optimal) FOIES was subsequently developed in [10]. The queries sketched below are mainly derived from the former; the maintenance of the required total order was given in [10]. The FOIES given below uses the relational calculus as the maintenance language, it handles both tuple insertions and tuple deletions, but it uses auxiliary relations.

We again assume these schemas: G for the input undirected graph and TC for the transitive closure.

A (near) total order LT on the nodes is needed, for choosing a node among a set of nodes that are indistinguishable in relational calculus. In practical systems, LT can be any total ordering available for the application and need not be maintained. In the theoretical consideration, LT must and can be maintained, and the details of this maintenance can be found in [10].

To maintain TC, we need to maintain two additional auxiliary relations: FOREST(A, B) for a spanning forest of the graph (FOREST is also symmetric), THRU(A, V, B) for indicating that V is on the unique path from A to B in FOREST if the nodes A and B are connected. The contents of the auxiliary relations are dependent on the order of the updates to the graph, i.e. the update history leading to the current graph.

Clearly we can derive the transitive closure of an undirected graph as a relational-calculus view of THRU easily. We need to demonstrate how to maintain the auxiliary relations FOREST and THRU.

To simplify the presentation, let $Eq(x, y, c, d)$ denote the formula $(x = c \wedge y = d) \vee (x = d \wedge y = c)$.

Maintaining FOREST and THRU after insertions

Suppose a new edge (a, b) is inserted. We need to change FOREST only if the inserted edge connects two previously disconnected trees (or equivalently a and b were not previously connected). Therefore we let FOREST$^{\text{new}}$ be FOREST$^{\text{old}} \cup \{(a,b),(b,a) \mid \neg$THRU$^{\text{old}}(a,a,b)\}$.

The queries for maintaining THRU resembles in a way the maintenance of TC of directed graphs after an edge insertion. Specifically, THRU$^{\text{new}}$ is given by:

$$
\begin{aligned}
\{(x,z,y) \mid & \mathsf{THRU}^{\text{old}}(x,z,y) \\
& \vee(Eq(x,y,a,b) \wedge (z = a \vee z = b)) \\
& \vee\neg\mathsf{THRU}^{\text{old}}(x,x,y)\wedge \\
& \quad \exists u \exists v Eq(u,v,a,b) \wedge \mathsf{THRU}^{\text{old}}(x,x,u) \\
& \quad\ \wedge \mathsf{THRU}^{\text{old}}(v,v,y) \wedge (z = a \vee z = b \\
& \quad\quad \vee \mathsf{THRU}^{\text{old}}(x,u,z) \vee \mathsf{THRU}^{\text{old}}(v,y,z))\}
\end{aligned}
$$

Roughly, the formula above states that the new THRU contains the old THRU, the new edge (a, b), and paths formed by paths in the old THRU and the new edge (a, b).

Maintaining FOREST and THRU after deletions

Suppose an existing edge (a, b) is deleted. We first update LT$^{\text{old}}$ to LT$^{\text{new}}$. If (a, b) is in FOREST$^{\text{old}}$, we remove it. Deleting (a, b) from FOREST$^{\text{old}}$ may cause one tree to split into two. When this happens, there can be either none or several edges in G^{new} which connect these two trees. For the former, we only need to eliminate relevant tuples in THRU$^{\text{old}}$ to complete the maintenance. For the latter, we first delete relevant tuples from THRU$^{\text{old}}$; then we pick a replacement edge and insert it into the spanning forest; finally we insert tuples into THRU that are relevant to the replacement edge. The procedure of inserting the replacement edge is identical to the maintenance of FOREST and THRU upon an insertion, and hence will not be repeated here.

We describe the deletion and replacement edge selection steps in the following.

Suppose (a, b) belongs to FOREST$^{\text{old}}$. Let FOREST^1 be a temporary relation which denotes FOREST$^{\text{old}} - \{(a,b)\}$. Let THRU^1 be the corresponding version of THRU for FOREST^1; this can be derived by $\{(x,z,y) \mid \mathsf{THRU}^{\text{old}}(x,z,y) \wedge \neg\mathsf{THRU}^{\text{old}}(x,a,y) \wedge \neg\mathsf{THRU}^{\text{old}}(x,b,y)\}$.

The candidate replacement edges are those edges (x, y) of the graph such that x and a are in one tree of FOREST^1 and y and b are in another tree of FOREST^1: $G^{\text{new}}(x,y) \wedge \mathsf{THRU}^1(x,a,a) \wedge \mathsf{THRU}^1(b,b,y)$. We use the LT$^{\text{new}}$ relation to pick the smallest of these edges, and then add it to FOREST. As mentioned above, the addition is done by essentially the same steps of maintenance as the insertion case.

We now use an example to illustrate the maintenance algorithms. Because G, FOREST, and the first and last columns of THRU are symmetric, we will only show half of the edges for clarity; furthermore, LT is not symmetric and only the chain part is shown.

Suppose our initial graph G is as given below.

G

a	b
c	d
c	e
d	e

LT

a	b
b	c
c	d
d	e
…	

FOREST

a	b
c	e
d	e

THRU

a	a/b	b
c	c/e	e
d	d/e	e
c	$c/d/e$	d

Then the corresponding LT, FOREST, and THRU relations can be as above. The notation $(a, a/b, b)$ is a shorthand for the two tuples of (a, a, b) and (a, b, b).

Suppose we insert the edge (b, c). Since both nodes are already in G, LT is not modified. Our maintenance algorithm will add the following tuples into the remaining two relations:

+FOREST

b	c

+THRU

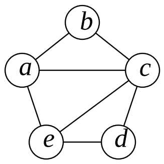
a	$a/b/c$	c
a	$a/b/c/e$	e
a	$a/b/c/e/d$	d
b	b/c	c
b	$b/c/e/$	e
b	$b/c/e/d$	d

The edge (b, c) is inserted into FOREST because it connects two previously disconnected trees.

The contents of LT, FOREST, and THRU will remain the same when edges (a, c) and (a, e) are subsequently inserted to G. The new graph G is shown below.

G

a	b
a	c
a	e
b	c
c	d
c	e
d	e



Now suppose (a, b) is deleted from the current G (shown above). There is no need to change LT. The actions to FOREST and THRU are as follows.

- The edge (a, b) is deleted from FOREST.
- The following tuples and their symmetries are deleted from THRU:

a	a/b	b
a	$a/b/c$	c
a	$a/b/c/e$	e
a	$a/b/c/e/d$	d

- (a, c) and (a, e) are two candidate replacement edges for (a, b), and (a, c) is chosen as c is less than e according to LT.
- The replacement edge is inserted into FOREST, and THRU is updated accordingly using the insertion algorithm. The resulting relations are:

G

a	c
a	e
b	c
c	d
c	e
d	e

LT

a	b
b	c
c	d
d	e
$\cdots$	

FOREST

a	c
b	c
c	e
d	e

THRU

a	a/c	c
a	$a/c/b$	b
a	$a/c/e$	e
a	$a/c/e/d$	d
b	b/c	c
b	$b/c/e/$	e
b	$b/c/e/d$	d
c	c/e	e
c	$c/d/e$	d
d	d/e	e

4 Regular chain Datalog views

In [13, 12], algorithms are given for constructing FOIES for all views defined by regular chain Datalog programs, for single-tuple insertions. Observe that these handle a class of views, whereas the constructions of the previous two sections were only designed for individual views.

A chain Datalog program is a set of rules of the form $p(x_1, x_{n+1}) \leftarrow q_1(x_1, x_2), \cdots, q_n(x_n, x_{n+1})$, and it is regular if, for each rule, all but the rightmost relation symbol are base relations. For example, the following is a regular chain Datalog program.

$$
\begin{aligned}
p_1(x, y) &\leftarrow p(x, y) \\
p_1(x, y) &\leftarrow q(x, y) \\
p_1(x, y) &\leftarrow q(x, z), p_1(z, y) \\
p(x, y) &\leftarrow s(x, z), p_1(z, y) \\
p(x, y) &\leftarrow t(x, z), p(z, y)
\end{aligned}
$$

To construct an FOIES, roughly, we first rewrite the regular expression for the program into a form which does not use the empty word, the empty set, and the $*$ operator (but it can use the $+$ operator). Then we define one derived predicate for each regular subexpression whose last operator is the $+$ operator. Then we will maintain all these derived relations as auxiliary relations. The way to maintain them is very much like the maintenance of transitive closure of directed graphs after edge insertions, except that more insertion rules are needed because the underlying regular expressions are more involved. The details can be found in [12].

In [10] an FOIES for the same generation view over acyclic graphs is given.

5 Set-based updates

In [12] extension to the FOIES for the TC of directed graphs is given to deal with the insertion of a special type, called *Cartesian closed*, of sets of edges. Essentially the extended FOIES can maintain the TC after the insertion of the union of a bounded number of Cartesian products of node sets. This is especially useful for the situation when the TC is over a view defined by union of conjunctive queries; a decision procedure is also given for deciding if that view is Cartesian closed for insertions of tuples to the base relations. Interestingly, this extension can also be used to build an FOIES for the maintenance of views defined by arbitrary regular chain Datalog programs (Section 4).

In [8] the FOIES for TC of acyclic graphs is extended to deal with the deletion of "acyclic" edges, the deletion of "anti-chain" sets of edges, and the deletion of "anti-chain" sets of nodes, all from arbitrary directed graphs.

6 Space complexity

The ability to maintain a view defined by a more powerful language using an algorithm written in a less powerful language may incur some extra cost in space for data storage. This cost can be measured in terms of the maximal arity of the auxiliary relations. Questions of interest include the following:

- What is the arity of the most space efficient FOIES for a view?
- Construct an FOIES, using auxiliary relations with no more than of a given arity, for a view.
- Is the arity hierarchy strict?

These questions have been considered in [10, 15] and some answers are known.

The most space-efficient FOIES for some views have been given. For example, for transitive closure of undirected graphs, the minimum arity of its FOIES is exactly two. (Observe that the FOIES given above for this view uses ternary auxiliary relations.)

The technique used to show that a certain arity is the optimal is either a modified Ehrenfeucht-Fraïssé game technique or an information theoretic technique [10, 15]. In general, it has been shown that the arity-based hierarchy is strict for all arities. However, the strictness proof uses queries with input relations having arities much larger than the auxiliary relations. It is still open whether the hierarchy remains strict for arities two or larger when the input relations of queries have arities bounded by a fixed number such as two, or by the arities of the auxiliary relations.

One other class of views with known optimal FOIES arity bounds are concerned about counting. Examples include the following: The parity query (whether a set has an even number of elements) can be maintained without auxiliary relations [10]. The MOD3 query (whether the number of elements in a set is a multiple of 3) can be maintained using just unary auxiliary relations [10]. Somewhat surprisingly, the EQCk query which tests whether two k-ary relations have equal cardinalities, is known to have an FOIES using only binary auxiliary relations [14]. Most other views with known optimal FOIES arity bounds are concerned about relatives of counting and transitive closure.

An FOIES can be either deterministic or nondeterministic, depending on whether its (stored) auxiliary relations are deterministic or nondeterministic, i.e. whether they depend on the order of the updates to reach a database state. The following results are reported in [11]. The nondeterministic FOIES are more powerful than the deterministic ones: deterministic FOIES using auxiliary relations with arity $\leq k$ are shown to be strictly weaker than their nondeterministic counterparts for each $k \geq 1$. Furthermore, there is a simple view which has a nondeterministic FOIES with binary auxiliary relations but does not have any deterministic FOIES with auxiliary relations of any arity.

7 SQL as the maintenance language

While an FOIES uses relational calculus as its maintenance language, practical relational systems support more powerful query languages such as SQL. Thus it is of interest to study incremental evaluation systems with SQL as the maintenance language.

There are two variants of such incremental evaluation systems. The notation SQLIES_{nest} is used to denote incremental evaluation systems where both the input database and the answer are flat relations, but the auxiliary database may involve nested relations. The notation SQLIES is used to denote systems whose auxiliary database is also restricted to flat relations. (SQLIES can also create a very large number of new symbols, unlike FOIES which does not create any.) Thus SQLIES approximates more closely what could be done in a relational database, which can store only flat relations.

Many questions about the power of SQLIES have been answered recently. [5] showed that SQLIES using no auxiliary relations is unable to maintain transitive closure of arbitrary graphs. In [7], it was proved that transitive closure of arbitrary graphs remains unmaintainable in SQLIES even in the presence of auxiliary data whose degrees are bounded by a constant, or are extremely small compared to the size of the input database.

On the positive side, [26] recently showed that if the bounded degree constraint on auxiliary data is removed, transitive closure of arbitrary graphs becomes maintainable in SQLIES. This is also true for the alternating path query, which is complete for polynomial-time. In [26] it was also shown that SQLIES_{nest} and SQLIES are equivalent. That means the restriction to flat relations does not incur a loss in power. Since many problems have a clearer and simpler implementation in SQLIES_{nest}, this equivalence gives us a way to "port" such theoretical implementations to the more realistic platform of commercial SQL database systems.

One can also ask what exactly is the limit of the power of SQLIES. Results aimed at answering this question have recently become available [28]. On the positive side, all relational queries expressible in second-order logic, and hence having the polynomial-hierarchy data complexity [23], are maintainable in SQLIES in a uniform manner. On the negative side, this is very close to the upper bound on the power of SQLIES. Furthermore, one must store a great deal of auxiliary information, either as nested relations or by creating new constants. Nonetheless, such maintenance systems can give much more power to practical systems.

8 Other related work

In this section we sketch some other related work. Such work can be divided into work related to FOIES, and work using a host programming language as the maintenance language, either from a database perspective or from a more general graph algorithmic perspective. Pointers to more general directions are also given.

Other work on FOIES: [14] discussed some relationships between FOIES and Σ_1^1 arity hierarchies. [16] shows how to maintain tree isomorphism (which cannot be defined even in relational calculus extended with the transitive closure operator). [4] considered the maintenance of constrained transitive closure of graphs with weighted edges (or nodes) by conjunctive queries. [29] investigated how to maintain the all-pairs shortest paths view and other related views for undirected graphs after insertions and deletions, using relational calculus, $+, <$.

[6] discusses how to maintain the transitive closure using SQL, by translating results using the relational calculus. [31] reports an implementation of incremental maintenance of some recursive views.

Work using host language as maintenance language but from a database perspective: FOIES is related to efficient maintenance of (stratified) databases [3, 24], where the goal is to efficiently compute the standard model of a stratified database after a database update. The latter is similar to FOIES in using the previous standard model (analogous to the auxiliary relations) to simplify the task of computing the standard model (query answer) after the update. Rather than storing intermediate relations, these approaches store reasons (or "supports") for including computed facts [3], or use meta-programs [24] to compute the difference between successive models. In [32], incremental evaluation of Datalog$^\neg$ is studied for its application to parallelism. Their approach associates with each derived fact a collection of records of counters, one for each iteration in bottom-up evaluation. The counters remember the number of times the fact is derived, and the number of times the fact is deleted. The algorithms can handle general Datalog$^\neg$ programs by using these counters from the appropriate iterations, but at the price of using recursive algorithms.

Graph algorithms: Also related to FOIES are online graph algorithms [21, 19, 25, 22]. Graph algorithms for online evaluation of transitive closure of graphs are given in [21, 19, 18], and a method to optimize transitive queries by using subtrees in graphs constructed in previous evaluations is presented in [22]. The main difference is that they use more elaborate data structures and recursive algorithms, whereas an FOIES only uses relations and nonrecursive queries.

Further pointers: The framework of FOIES is also closely related to a branch of computational complexity and of finite model theory, called descriptive complexity [20]. Moreover, [17] contains a collection of papers on many different issues regarding materialized views.

9 Concluding remarks

In summary, we can see that incremental evaluation systems are a convenient way to add expressive power to existing database systems, and they are also a way to speed up the process of computing the view contents. Some of these systems can be readily implemented in real applications, although some other practical issues such as indexing may need to be considered.

One advantage of an FOIES over IES using other maintenance languages is that it has great potential for parallelization, because it uses the relational calculus, which has a constant-time parallel complexity AC_0 [1], as its maintenance language.

There are still many research problems to be considered. For example, views with known FOIES are still a minority among possible views, and it is of interest to decide whether FOIES exist for the majority of views. An interesting problem is to find more space-efficient fragment of all SQLIES. An interesting theoretical problem is whether the arity hierarchy is strict for a limited input arity.

References

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

[2] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Proc Symposium on Principles of Programming Languages, Texas*, pages 110–120, Jan, 1979.

[3] K. R. Apt and J.-M. Pugin. Maintenance of stratified databases viewed as a belief revision system. In *Proc. ACM Symp. on Principles of Database Systems*, pages 136–145, 1987.

[4] G. Dong and R. Kotagiri. Maintaining constrained transitive closure by conjunctive queries. *Proc. Int. Conf on Deductive and Object-Oriented Databases*, Switzerland, Springer-Verlag, 1997.

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

[6] G Dong, L Libkin, J Su and L Wong. Maintaining transitive closure of graphs in SQL. *Int'l J. of Information Technology*, 5(1):46–78, 1999.

[7] 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, Jan 1997.

[8] G. Dong and C. Pang. Maintaining transitive closure in first-order after node-set and edge-set deletions. *Information Processing Letters*, 62(3):193–199, 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. Arity bounds in first-order incremental evaluation and definition of polynomial time database queries. *Journal of Computer and System Sciences*, 57(3):289–308, December 1998. Preliminary version in *Proc of ACM Symp. on Principles of Database Systems*, 1995.

[11] G. Dong and J. Su. Deterministic FOIES are strictly weaker. *Annals of Mathematics and Artificial Intelligence*, 19(1-2):127–146, 1997.

[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 R. Topor. Incremental evaluation of datalog queries. In *LNCS 646: Proceedings of 4th International Conference on Database Theory, Berlin, Germany*, pages 282–296. Springer-Verlag, October 1992.

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

[15] G. Dong and L. Zhang. Separating Auxiliary Arity Hierarchy of First-Order Incremental Evaluation Using (3+1)-ary Input Relations. TR 97/13, CS Dept, Univ of Melbourne.

[16] K. Etessami. Dynamic Tree Isomorphism via First-Order Updates. PODS 1998: 235-243.

[17] A. Gupta and I. S. Mumick (eds). Materialized Views: Techniques, Implementations, and Applications. MIT Press, 1999.

[18] M. R. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In FOCS 1995: 664-672.

[19] T. Ibaraki and N. Katoh. On-line computation of transitive closure of graphs. *Information Processing Letters*, 16:95-97, 1983.

[20] N. Immerman. Descriptive Complexity. Springer, New York. December, 1998.

[21] G.F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48:273-281, 1986.

[22] H. Jakobsson. On materializing views and online queries. In *Proc. Int. Conf on Database Theory*, LNCS 646, Springer-Verlag, 1992.

[23] D. Johnson. *A Catalog of Complexity Classes*, volume A of *Handbook of Theoretical Computer Science*, pages 67–161. North Holland, 1990.

[24] V. Küchenhoff. On the efficient computation of the difference between consecutive database states. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proc. Second Int. Conf. on Deductive Object-Oriented Databases*, LNCS 566, pages 478–502. Springer-Verlag, 1991.

[25] J. La Poutre and J. van Leeuwen. Maintenance of transitive closures and transitive reductions of graphs. Tech Report RUU-CS-87-25, Dept of CS, University of Utrecht, The Netherlands, 1987. Extended abstract in LNCS 314, pp. 106-120.

[26] L. Libkin and L. Wong. Incremental recomputation of recursive queries with nested sets and aggregate functions. In *LNCS 1369: Proc of 6th Int'l Workshop on Database Programming Languages, Estes Park, Colorado, August 1997*, pages 222–238. Springer-Verlag.

[27] L. Libkin and L. Wong. Query languages for bags and aggregate functions. *J. of Computer and System Sciences*, 55(2):241–272, October 1997.

[28] L. Libkin and L. Wong. On the power of incremental evaluation in SQL-like languages. In *Proc of 7th Int'l Workshop on Database Programming Languages*. See also: SQL can maintain polynomial-hierarchy queries. Technical report, Institute of Systems Science, Singapore, 1997.

[29] C. Pang, R. Kotagiri and G. Dong. Incremental FO(+, <) maintenance of all-pairs shortest paths for undirected graphs after insertions and deletions. In *Proc of Int'l Conf on Database Theory*, Jerusalem, Jan 1999. More details in C Pang's PhD thesis: Maintenance of Reachability in Graphs Using First-Order Queries with Addition and Less-than. Univ of Melbourne, 1999.

[30] S. Patnaik and N. Immerman. Dyn-FO: A parallel dynamic complexity class. *J. of Computer and System Sciences*, 55(2):199–209, Oct 1997. Preliminary version in *Proc. of ACM Symp. on Principles of Database Systems,* 1994.

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

[32] O. Wolfson, H. M. Dewan, S. J. Stolfo, and Y. Yemini. Incremental Evaluation of Rules and Its Relationship to Parallelism. In *Proc. of ACM SIGMOD Conference*, pages 78–87, 1991.