Maintaining Transitive Closure of Graphs in SQL

Guozhu Dong
Dept. of CSE
Wright State University
Dayton, Ohio 45435
gdong@cs.wright.edu

Leonid Libkin
Bell Laboratories
600 Mountain Ave
Murray Hill, NJ 07974
libkin@bell-labs.com

Jianwen Su
Dept. of CS
University of California
Santa Barbara, CA 93106
su@cs.ucsb.edu

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

Abstract

It is common knowledge that relational calculus and even SQL are not expressive enough to express recursive queries such as the transitive closure. In a real database system, one can overcome this problem by storing a graph together with its transitive closure and maintaining the latter whenever updates to the former occur. This leads to the concept of an *incremental evaluation system*, or IES.

Much is already known about the theory of IES but very little has been translated into practice. The purpose of this paper is to fill in this gap by providing a gentle introduction to and an overview of some recent theoretical results on IES.

The introduction is through the translation into SQL of three interesting positive maintenance results that have practical importance – the maintenance of the transitive closure of acyclic graphs, of undirected graphs, and of arbitrary directed graphs. Interestingly, these examples also allow us to show the relationship between power and cost in the incremental maintenance of database queries.

1 Introduction

It is common knowledge that the expressiveness of relational calculus and even SQL is limited. For example, recursive queries such as the transitive closure cannot be defined [2, 23] in these languages. However, in a real database system, one can try to overcome this problem by storing a graph together with its transitive closure and maintaining the latter whenever updates (i.e. the insertion or deletion of edges) to the former occur. In other words, the recursive queries are evaluated and maintained incrementally. The result of such a recursive query can be thought of as a view of the database and the incremental evaluation of the query as view maintenance. The above leads to the concept of an *incremental evaluation system*, or IES.

Incremental evaluation is often seen as merely a means to avoid expensive re-computation. However, from what we have said above of the transitive closure query, 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 have been done otherwise. Coming back to 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 recomputation is not even the issue here, for the query is not even do-able in SQL without incremental evaluation in the first place!

A salient ingredient in an IES is its ambient language. All maintenance in an IES must be expressible in the ambient language of the IES. For example, if the IES is a commercial relational database system, then its ambient language is SQL: it is allowed to use SQL but not C nor other languages to express the maintenance. This restriction on the ambient language arises from the modeling of the practical constraints imposed by real systems, mostly for reasons of efficiency, optimizability, security, and control. At the same time, the use of restricted ambient language also gives rise to an opportunity to investigate its theoretical limits.

Much is already known about the theory of IES [9, 19, 11, 12, 17, 14, 13, 16, 8, 22, 27, 26, etc.]. However, very little has been translated into practice, perhaps for the reason that work on the theory of IES is cast in an abstract mathematical form and the translation to SQL database systems is not always obvious.[1] The objectives of this paper are two folds: Firstly, we select three interesting positive results that have practical importance and show how to realize them in commercial SQL database systems. Secondly, we aim to provide a gentle

[1] We are only aware of one system, called ADEPT [29], that implements incremental maintenance of transitive closure of some classes of graphs.

overview of some recent theoretical results on IES.

The three results that we translate into SQL concern the maintenance of the transitive closure of various kinds of graphs. The computation of transitive closure is the determination of the existence of a path between nodes in a graph. Our choice is motivated from two perspectives. Firstly, transitive closure is *the* canonical representative of recursive queries. It is known to be inexpressible in relational calculus and SQL [2, 23]. Secondly, transitive closure has so much practical importance to the extent that several non-standard versions of SQL included special operators for implementing it.[2] Thus a technique for maintaining transitive closure in standard commercial SQL database systems would be very interesting. We should also emphasize that the approach illustrated in (Section 4 of) this paper can be generalized; in fact, it leads to a uniform way to implement all queries in the polynomial hierarchy [21] using standard commercial SQL database systems.

Organization

We consider maintaining the transitive closure of three kinds of graphs in commercial SQL database systems: acyclic directed graphs, undirected graphs, and arbitrary directed graphs.

Acyclic graphs are the focus of Section 2. The transitive closure of these graphs are very easy to maintain in commercial SQL database systems, requiring no more than the equivalent of pure relational calculus [13]. This example is also a good illustration of the power of the IES model, because it is well known [2, 23] that pure relational calculus and even SQL cannot compute *from scratch* the transitive closure of such graphs.

Undirected graphs are the focus of Section 3. The first known technique for maintaining the transitive closure of undirected graphs using relational calculus (or equivalently, first-order logic) as the ambient language was that of [27]. A more space-efficient technique using relational calculus was reported later in [14, 15]. The maintenance of the transitive closure of undirected graphs using SQL is more involved and more expensive than acyclic graphs. In particular, the maintenance of the transitive closure of acyclic graphs is very economical on space because all we need to store are the transitive closure and the graph itself, whereas some additional binary relations must be maintained for the maintenance of the transitive closure of undirected graphs. Such additional relations are called "auxiliary" relations. These auxiliary relations should not be confused with temporary intermediate relations: The former are used to store

[2]We also note that the SQL3 proposal does include a recursive construct that enables it to express the transitive closure. However, majority of systems are still only SQL92 compliant.

information between updates whereas the latter are only used to simplify the expression of the SQL queries.

Arbitrary directed graphs are the focus of Section 4. In contrast to the other two classes of graphs, the maintenance of the transitive closure of arbitrary directed graphs is much more complicated and costly. In fact, at the time of writing, it is still open whether the transitive closure of such graphs can be maintained using pure relational calculus [9] after edge deletions. However, a technique for maintaining the transitive closure of arbitrary directed graphs using SQL was recently discovered [22]. The technique is quite expensive in terms of space: We need to maintain an auxiliary relation which makes use of up to an exponential number of integers not appearing in the input graph. In contrast, the maintenance for undirected and acyclic graphs does not make use of constants not in the input graph.

In Section 5, we discuss the complexity of our three example implementations. We demonstrate that for acyclic and undirected graphs only a small constant number of joins are required to carry out the maintenance, giving them considerable performance advantage over typical transitive closure implementations based on iterated joins.

Finally, Section 6 concludes the paper with an account of the more theoretical aspects of IES. In particular, we contrast results on IES that use pure relational calculus as their ambient language to those that use SQL as their ambient language.

2 Transitive Closure of Acyclic Graphs

It is appropriate to introduce the practical aspect of IES using a simple interesting example. So we show how to maintain the transitive closure of acyclic graphs in SQL database systems here. It is a good illustration of the power of the IES model, because it is well known [2, 23] that pure relational calculus and even SQL cannot compute from scratch the transitive closure of such graphs.

We base this section on the theoretical work reported in [13, 10]. In particular, the SQL queries given below are derived from [13, 10]. We assume the following schemas: $G(Start, End)$ for the input graph and $TC(Start, End)$ for the transitive closure. The interpretation of these two tables is as follow. A tuple (x, y) is in the table G if and only if there is a directed edge from the node x to the node y in the input graph. A tuple (x, y) is in the table TC if and only if there is a directed path from the node x to the node y in the input graph. Our problem is to use SQL queries to maintain the relationship between G and TC described above when an edge is added to or deleted from the table G.

Maintenance Under Insertions

Suppose an edge (a, b) is inserted. We maintain TC as follows. First all new tuples and possibly some old ones are constructed and stored in a temporary relation, *TC-NEW*. Then the truly new tuples in *TC-NEW* are merged into *TC*.

```
    SELECT *
(1) FROM (SELECT Start = TC.Start, End = b
          FROM TC
          WHERE TC.End = a
         UNION
(2)       SELECT Start = a, End =  TC.End
          FROM TC
          WHERE b = TC.Start
         UNION
(3)       SELECT Start = TC1.Start, End = TC2.End
          FROM TC AS TC1, TC AS TC2
          WHERE TC1.End = a AND TC2.Start = b
         ) AS T
    INTO TEMP TC-NEW

    INSERT INTO TC-NEW (Start, End)
(4)   VALUES (a, b)

    SELECT *
    FROM TC-NEW AS T
    WHERE NOT EXISTS (SELECT *
                      FROM  TC
                      WHERE TC.Start=T.Start AND TC.End=T.End)
    INTO TEMP DELTA

    INSERT INTO TC
      SELECT *
      FROM DELTA
```

Essentially, the new transitive closure is obtained by adding to the old transitive closure the following (Figure 1): (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.

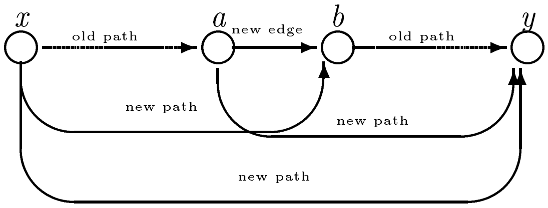

Figure 1: Transitive closure of graph when new edge is inserted

Maintenance Under Deletions

Suppose an existing edge (a, b) is deleted. The maintenance of TC is a slightly more complicated problem and some theoretical insight is required in order to see why it can be done using nothing more than SQL or even pure relational calculus [13].

Step 1: Finding the suspects

We derive and store in a temporary table $SUSPECT(Start, End)$ all those pairs (x, y) such that there is a path in the old graph (the one before the deletion) from x to y that goes through the edge (a, b).

```
SELECT *
FROM (SELECT Start = X.Start, End = Y.End
      FROM TC AS X, TC AS Y
      WHERE X.End = a AND Y.Start = b
    UNION
      SELECT Start = X.Start, End = b
      From TC AS X
      WHERE X.End = a
    UNION
      SELECT Start = a, End = X.End
      FROM TC AS X
      WHERE X.Start = b
    UNION
      SELECT Start = a, End = b
      FROM TC AS X
      WHERE X.Start = a AND X.End = b)
INTO TEMP SUSPECT
```

Step 2: Finding the trusty guys

We derive and store in a temporary table *TRUSTY*(*Start*, *End*) those paths in *TC* that are clearly unaffected by the deletion of the edge (a, b). Obviously, these can be obtained by (1) deleting *SUSPECT* from *TC* and (2) including the other edges of G.

```
     SELECT *
(1)  FROM (SELECT *
            FROM TC
            WHERE NOT EXISTS (SELECT *
                              FROM SUSPECT
                              WHERE SUSPECT.Start = TC.Start AND
                                    SUSPECT.End = TC.End)
           UNION
(2)         SELECT *
            FROM G
            WHERE G.Start <> a AND G.End <> b)
     INTO TEMP TRUSTY
```

Step 3: Deleting the bad guys

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. From the result of [13, 10], these constitute all paths that should be in the new transitive closure. To see this, consider a path in the desired transitive closure going through exactly the nodes $x_1, x_2, ..., x_k$ in the given order. Suppose there is $i < j$ such that (x_i, x_j) is in *SUSPECT* but not in $TRUSTY$. Then there is a tightest pair of u and v such that $i \leq u < v \leq j$, and (x_u, x_v) is in $SUSPECT$. If $v = u + 1$, (x_u, x_v) is an edge and thus is in $TRUSTY$ and both (x_1, x_u) (provided $u \neq 1$) and (x_v, x_k) (provided $v \neq k$) are in $TRUSTY$. Thus this path can be obtained by concatenating up to three $TRUSTY$ paths. If $v > u + 1$, then both (x_1, x_{u+1}) and (x_{u+1}, x_k) are in $TRUSTY$. Thus this path can be obtained by concatenating two $TRUSTY$ paths. All other paths are already captured by $TRUSTY$. So we can simply delete from TC all other paths.

```
     SELECT *
(1)  FROM (SELECT *
            FROM TRUSTY
          UNION
(2)         SELECT T1.Start, T2.End
```

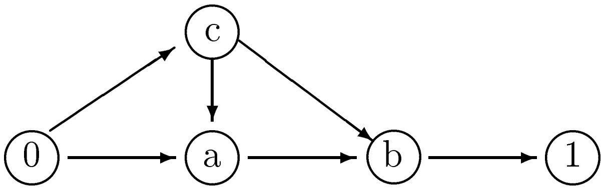

Figure 2: An acyclic graph

```
          FROM TRUSTY T1, TRUSTY T2
          WHERE T1.End = T2.Start
        UNION
(3)       SELECT T1.Start, T3.End
          FROM TRUSTY T1, TRUSTY T2, TRUSTY T3
          WHERE T1.End = T2.Start AND T2.End = T3.Start)
    INTO TEMP TC-NEW

   DELETE FROM TC
   WHERE NOT EXISTS (SELECT *
                     FROM TC-NEW T
                     WHERE T.Start = TC.Start AND T.End = TC.End)
```

An Example for the Deletion Case

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

Step 1 The contents of *SUSPECT* are as follows:

SUSPECT

Start	*End*
0	b
0	1
a	b
a	1
c	b
c	1

TRUSTY

Start	*End*
0	a
0	c
c	a
c	b
b	1

Step 2 The contents of *TRUSTY* are given above.

Step 3 All the good paths are now derived from $TRUSTY$ through zero, e.g. for the case of $(0, c)$, one, e.g. $(0, b)$, or two joins, e.g. $(0, 1)$, followed by projections. TC is updated by deleting those paths that are not good. It is more instructive to visualize all the edges other than (a, b) as a sequence of edges.

3 Transitive Closure of Undirected Graphs

In this section we show how to maintain the transitive closure of undirected graphs in SQL. An undirected graph contains the edge from a node y to a node x whenever it contains the edge from x to y. In contrast to the acyclic graphs case of the previous section, the transitive closure of these graphs can only be maintained if we store additional relations which are called *auxiliary* relations. The auxiliary relations are used to maintain information between updates to the graph.

The first IES that maintains the transitive closure of undirected graph using nothing more than pure relational calculus was given in [27] and an improved (space-wise optimal) IES was subsequently developed in [14]. The SQL queries sketched below are mainly derived from the former, except for the explicit maintenance and use of the total order.

We again assume the following schemas: $G(\mathit{Start}, \mathit{End})$ for the input undirected graph and $TC(\mathit{Start}, \mathit{End})$ for the transitive closure. Since values in a tuple have explicit positions, we assume that G and TC stored are symmetric. Although each update on G specifies an (ordered) pair (a, b), the actual changes to G are always made in terms of both (a, b) and (b, a). The SQL queries given in this section assume that the updates are given in ordered pairs. Without this assumption the maintenance is much more involved; see [14] for the solution without this assumption.

To maintain TC, we need to use some auxiliary relations that we must also maintain using SQL: $LESSTHAN(\mathit{Small}, \mathit{Large})$ for a total order on all nodes in the graph, $FOREST(A, B)$ for a spanning forest of the graph ($FOREST$ is also symmetric), $THROUGH(A, V, B)$ 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.

The order relation $LESSTHAN$ is used for choosing an edge from a set of edges satisfying the same condition. This ordering is needed because a graph can have several distinct spanning forests and our incremental evaluation system has to select only one of them. It does not matter which one we select, but we do have to select one. If an ordering on the nodes are available from the underlying relational database system, then we do not need *LESSTHAN*. For example, if the nodes are strings then we can use the string comparison operator of SQL instead of *LESSTHAN*.

Deriving Transitive Closure From *THROUGH*

Recall that the table *THROUGH* is set up to contain a tuple (a, m, b) if and only if a and b are connected and m is on the unique path from a to b in the spanning forest in *FOREST*. We can derive the transitive closure of our undirected graph as a view of *THROUGH* straightforwardly.

```
CREATE VIEW TC (Start, End) AS
  SELECT DISTINCT Start=A, End=B
  FROM THROUGH
```

We need to demonstrate how to maintain the auxiliary tables *LESSTHAN*, *FOREST*, and *THROUGH* in SQL. First, we define the following view, *GNODES*, to hold all nodes in G.

```
CREATE VIEW GNODES (Node) AS
  SELECT Node=Start
  FROM G
 UNION
  SELECT Node=End
  FROM G
```

Note that UNION in SQL removes all duplicates by default.

Maintenance of the Total Order *LESSTHAN*

The first auxiliary relation we maintain after an update to G is *LESSTHAN*. Recall that *LESSTHAN* is a total ordering on the nodes to help us later in choosing an edge from a set of edges satisfying the same condition. Note that if an ordering on the nodes are available from the underlying relational database system, then we do not need *LESSTHAN*. For example, if the nodes are strings then we can use the string comparison operator of SQL instead of *LESSTHAN*.

Suppose an edge (a, b) is inserted. We update *LESSTHAN* by executing *Expand*(a), *Expand*(b), *Initial*(a, b), where *Expand*(x) is the following update that makes x larger than all other nodes in the total order, provided that x is new.

```
INSERT INTO LESSTHAN (Small, Large)
  SELECT Small=Node, Large=x
  FROM GNODES
  WHERE x NOT IN (SELECT * FROM GNODES)
```

and *Initial*(x, y) is the following update which simply inserts (a, b) to *LESSTHAN* when *LESSTHAN* is empty.

```
INSERT INTO LESSTHAN (Small, Large)
  SELECT DISTINCT Small=x, Large=y
  FROM G
  WHERE NOT EXISTS (SELECT * FROM LESSTHAN)
```

Suppose an edge (a, b) is deleted from G. We update *LESSTHAN* by executing *Shrink*(a), *Shrink*(b), where *Shrink*(x) is the following update that removes x from the total order, provided the node x no longer exists.

```
DELETE FROM LESSTHAN
  WHERE x NOT IN (SELECT * FROM GNODES)
        AND (Small=x OR Large=x)
```

Maintenance of *FOREST* AND *THROUGH* under Insertions

Suppose a new edge (a, b) is inserted. We first update *LESSTHAN* as described above. There is a 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 maintain *FOREST* as follows:

```
INSERT INTO FOREST
  SELECT A=Start, B=End
  FROM  G
  WHERE NOT EXISTS (SELECT *
                    FROM  THROUGH
                    WHERE A=a AND B=b)
        AND (Start=a AND End=b OR Start=b AND End=a)
```

The queries for adjusting *THROUGH* resembles in a way the maintenance of *TC* of directed graphs after the insertion of edges. However, *THROUGH* is symmetric, i.e. if (x, v, y) is in *THROUGH* then so is (y, v, x); this complicates the expression. To make it simple, we first create the following temporary relation.

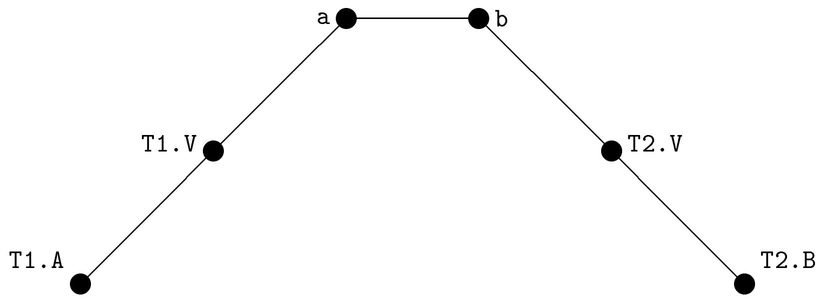

Figure 3: “Connecting” two paths using the new edge

```
SELECT *
FROM (SELECT A=N.Node, V=N.Node, B=N.Node
      FROM GNodes N
     UNION
      SELECT *
      FROM THROUGH)
INTO TEMP T-STAR
```

Using *T-STAR* we apply the following updates.

```
    INSERT INTO THROUGH
(1)    SELECT A=T1.A, V=N.Node, B=T2.B
       FROM  GNODES AS N, T-STAR AS T1, T-STAR AS T2
       WHERE T1.B=a AND T2.A=b AND N.Node=T1.V
      UNION
(2)    SELECT A=T1.A, V=N.Node, B=T2.B
       FROM  GNODES AS N, T-STAR AS T1, T-STAR AS T2
       WHERE T1.B=a AND T2.A=b AND N.Node=T1.V AND N.Node=T2.V
      UNION
(3)    SELECT A=T1.B, V=N.Node, B=T2.A
       FROM  GNODES AS N, T-STAR AS T1, T-STAR AS T2
       WHERE T1.B=a AND T2.A=b AND N.Node=T1.V
      UNION
(4)    SELECT A=T1.B, V=N.Node, B=T2.A
       FROM  GNODES AS N, T-STAR AS T1, T-STAR AS T2
       WHERE T1.B=a AND T2.A=b AND N.Node=T1.V AND N.Node=T2.V
```

Parts (1) and (2) above “connect” paths using the new edge (a, b) (see Figure 3), while Parts (3) and (4) make *THROUGH* symmetric.

Maintenance of *FOREST* AND *THROUGH* under Deletions

Suppose an existing edge (a, b) is deleted. We proceed by first updating *LESSTHAN*. If (a, b) is in *FOREST*, we remove it. Deleting (a, b) from *FOREST* may cause one tree to split into two. When this happens, there can be none *or* several edges in G which connect these two trees. For the former, we only need to eliminate relevant tuples in *THROUGH* to complete the maintenance. For the latter, we first delete relevant tuples (to (a, b)) from *THROUGH*; then we pick a replacement edge and insert it into *FOREST*; finally we insert tuples that are relevant to the replacement edge. The procedure of inserting the replacement edge is identical to the maintenance of *FOREST* and *THROUGH* upon an insertion and thus the details are omitted. We describe the deletion and replacement edge selection steps in the following.

Step 1: Identify a replacement edge

In the case when the deleted edge (a, b) is in *FOREST*, we select a replacement edge and put it into a temporary relation *REP*. This is done in two steps. First we find all possible replacement edges.

```
SELECT *
FROM G
WHERE EXISTS (SELECT DUMMY = 1
              FROM THROUGH AS T1, THROUGH AS T2
              WHERE T1.A=Start AND T1.V=a AND T1.B=b AND
                    T2.A=a AND T2.V=b AND T2.B=End)
  AND EXISTS (SELECT *
              FROM FOREST
              WHERE A=a AND B=b)
INTO TEMP REP-ALL
```

We then pick the smallest edge in *REP-ALL* according to the total order *LESSTHAN* and store it in *REP*. Note that *REP* has exactly one edge. Note that if the underlying relational database provides an ordering on the nodes, then this piece of SQL codes can be replaced by using the appropriate comparison operation in SQL.

```
SELECT DISTINCT Start, End
FROM REP-ALL AS R, LESSTHAN
WHERE NOT EXISTS (SELECT *
                  FROM  REP-ALL AS R1, LESSTHAN AS LT
                  WHERE R1.Start=LT.Small AND R.Start=LT.Large
```

```
                    OR (R1.Start=R.Start AND
                        R1.End=LT.Small AND R.End=LT.Large)
  AND Small=Start AND Large=End
INTO TEMP REP
```

Step 2: Delete relevant edges in *THROUGH*

Intuitively, if a tuple (x, v, y) in *THROUGH* such that the edge (a, b) is on the unique path between x, y, the tuple must be deleted. This happens exactly when *THROUGH* contains both (x, a, b) and (a, b, y).

```
    SELECT *
(1) FROM (SELECT T.*
          FROM THROUGH AS T, THROUGH AS T1, THROUGH AS T2
          WHERE T1.V=a AND T1.B=b AND T2.A=a AND T2.V=b
            AND T1.A=T.A AND T2.B=T.B
            AND EXISTS (SELECT * FROM  FOREST WHERE A=a AND B=b)
        UNION
(2)       SELECT T.*
          FROM THROUGH AS T, THROUGH AS T1, THROUGH AS T2
          WHERE T1.V=a AND T1.B=b AND T2.A=a AND T2.V=b
            AND T1.A=T.B AND T2.B=T.A
            AND EXISTS (SELECT * FROM  FOREST WHERE A=a AND B=b))
    INTO TEMP DELTA

    DELETE FROM THROUGH
      SELECT *
      FROM DELTA
```

Again, Part (2) above is to delete tuples that are symmetric to those deleted in Part (1).

Step 3: Insert the replacement edge

The step to insert the edge in *REP* into *FOREST* and maintain *THROUGH* is almost identical to the insertion case.

Step 4: Delete (a, b) **from** *FOREST*

The final step is to delete (a, b) from *FOREST*.

```
DELETE FROM FOREST
  WHERE A=a AND B=b OR A=b AND B=a
```

An Example

It should be pointed out that G, *FOREST*, and the first and last columns of *THROUGH* are symmetric and we only show half of the edges for clarity; furthermore, *LESSTHAN* is not symmetric and only the chain part is shown.

Suppose our graph G is as follows.

G

Start	*End*
a	b
c	d
c	e
d	e

LESSTHAN

Small	*Large*
a	b
b	c
c	d
d	e
⋮	

FOREST

A	*B*
a	b
c	e
d	e

THROUGH

A	*V*	*B*
a	a/b	b
c	c/e	e
d	d/e	e
c	c/d/e	d

Then the corresponding *LESSTHAN*, *FOREST*, and *THROUGH* relations can be as above. We use the notation $(a, a/b, b)$ as a shorthand for the two tuples of (a, a, b) and (a, b, b).

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

+FOREST

A	*B*
b	c

+THROUGH

A	*V*	*B*
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 the *FOREST* relation because it connects two previously disconnected trees.

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

G

Start	*End*
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 *LESSTHAN*. For *FOREST* and *THROUGH*, the algorithm will perform the following steps.

Step 1: *REP-ALL* will contain (a, c) and (a, e), and *REP* has the smaller edge of the two, (a, c).

Step 2: The following tuples and their symmetries are deleted from *THROUGH*:

$-$*THROUGH*

A	*V*	*B*
a	a/b	b
a	a/b/c	c
a	a/b/c/e	e
a	a/b/c/e/d	d

Steps 3 and 4: The replacement edge is inserted into *FOREST* and *THROUGH* is updated accordingly using the insertion algorithm. The deletion edge (a, b) is deleted from *FOREST*. The resulting relations are as follows.

G

Start	*End*
a	c
a	e
b	c
c	d
c	e
d	e

LESSTHAN

Small	*Large*
a	b
b	c
c	d
d	e
⋮	

FOREST

Start	*End*
a	c
b	c
c	e
d	e

THROUGH

A	V	B
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 Transitive Closure of Arbitrary Directed Graphs

Arbitrary directed graphs are the most complicated graphs considered in this paper. In this section we present SQL queries for maintaining the transitive closure of these graphs. In contrast to acyclic graphs, it is not possible to use SQL queries to maintain the transitive closure of arbitrary directed graphs without using auxiliary relations [8]. Also, in contrast to undirected graphs, it is not known whether it is possible to maintain the transitive closure of arbitrary directed graphs using pure relational calculus. So we have to consider using more powerful features of SQL that we have managed to avoid in the two previous sections—aggregate functions and `GROUPBY` operator. These are features that makes SQL strictly more powerful than pure relational calculus [23, 25].

We use the technique from [22] to maintain the transitive closure of arbitrary directed graphs by SQL queries. The idea is to maintain complete information of all possible paths between the nodes of the graph. A path can basically be represented by its start node, its end node, and the set of all its intermediate nodes. To keep all the paths, it may appear that we need to store a nested relation, which of course cannot be done directly in a first-normal form relational database.

The trick around this little problem is to generate some unique numbers to identify each path. Then a path that is identified by the number i and starts at x, ends at y, and going through intermediate nodes n_1, ..., n_k can be represented by the set of tuples (i, x, y, n_1), ..., (i, x, y, n_k). Since the numbers identifying different paths are different, all these sets of tuples can be stored in the same table. As to how to generate these unique numbers, we just use the standard arithmetics and aggregate functions available in a typical commercial

SQL database system!

In the rest of this section, we present the SQL queries for realizing the above solution to the problem of maintaining the transitive closure of arbitrary graphs. These queries are translated from theoretical results in [22].

We again use the schemas $G(Start, End)$ and $TC(Start, End)$ to represent the input graph and its transitive closure. In addition, we use two auxiliary relations $R(Pnum, St, End, IntS, IntD)$ and $DUMMY(Pnum, St, End, IntS, IntD)$. In both R and $DUMMY$, the attributes are interpreted as follow: $Pnum$ is the number assigned to uniquely identify a path; St is the first node of the path; End is the last node of the path; $IntS$ and $IntD$ are two consecutive nodes on the path ($IntS$ is source, $IntD$ is destination.)

The auxiliary relation R is maintained in such a way that it contains a tuple (k, x, y, a, b) if and only if the path whose number is k starts from x, ends in y, and goes through the edge from a to b. The relation $DUMMY$ contains exactly one tuple $(0, NULL, NULL, NULL, NULL)$ that is needed to handle computation of `MAX` aggregate function when R is empty.

In what follows, $pairing(x, y)$ is the pairing function on natural numbers:

$$pairing(x, y) = \frac{(x+y)(x+y+1)}{2} + y$$

Deriving Transitive Closure From R

Given the property of R, the transitive closure of G can be easily derived from R.

```
CREATE VIEW TC AS
  SELECT DISTINCT Start= St, End = End
  FROM R
```

It remains to explain how we maintain the auxiliary table R in SQL.

Maintenance Under Deletions

Suppose an existing edge (a, b) is deleted. Then R is reconstructed trivially by deleting every path that goes through (a, b).

```
SELECT Pnum
FROM R
```

```
WHERE IntS = a AND IntD = b
INTO TEMP DEADPATH

DELETE FROM R
WHERE Pnum IN DEADPATH
```

Maintenance Under Insertions

Insertion is more complicated, mostly because SQL is not well designed [5, 4]. Suppose a new edge (a, b) is inserted into G. The reconstruction of R requires the following 8 steps.

Step 1

Create a temporary table *V1*$(Pair, Comp1, Comp2, St, End)$. The attributes of this view are interpreted as follow: $Pair$ is a number identifying a new path that is being created by concatenating paths whose numbers in R are *Comp1* and *Comp2*, via the new edge (a, b); *St* is the first node of the path; and *End* is the last node of the path. The purpose of this view is to generate new numbers to identify new paths formed by the linking of two old paths by the new edge (a, b).

```
SELECT DISTINCT
  Pair =  pairing (R1.Pnum, R2.Pnum),
  Comp1 = R1.Pnum,
  Comp2 = R2.Pnum,
  St    = R1.St,
  End   = R2.End
FROM R R1, R R2
WHERE R1.End = a  AND R2.St = b
INTO V1
```

Step 2

Create the first set to be inserted into R, given by paths formed by connecting two existing paths using the edge (a, b). The numbers that uniquely identify these new paths have already been created in *V1* of the previous step. All we need to do now is for each such new path, (1) add all intermediate edges of the its two component paths and (2) add (a, b) as an intermediate edge linking the two components. This step is accomplished by the SQL query below.

```
   SELECT *
   FROM (
(1)    SELECT DISTINCT
         Pnum = V1.Pair,
         St   = V1.St,
         End  = V1.End,
         IntS = R.IntS,
         IntD = R.IntD
       FROM V1, R
       WHERE (R.Pnum = V1.Comp1 AND R.St = V1.St AND R.End = a)
          OR (R.Pnum = V1.Comp2 AND R.St = b AND R.End  = V1.End)
    UNION
(2)   SELECT DISTINCT
        Pnum = V1.Pair,
        St   = V1.St,
        End  = V1.End,
        IntS = a,
        IntD = b
      FROM V1)
   INTO TEMP VINS1
```

Step 3

Create a temporary table *TEMP1* to give us all the paths we have found so far. This is a preparatory step for generating new numbers to identify the remaining new paths. The use of the $DUMMY$ table is necessary to ensure that this view is nonempty.

```
   SELECT *
   FROM (
      SELECT * FROM VINS1
    UNION
      SELECT * FROM R
    UNION
      SELECT * FROM DUMMY)
   INTO TEMP TEMP1
```

Step 4

Find the maximum path number in *TEMP1* and calculate a safe lower bound $NewId$ for new path numbers. This lower bound is needed before we can generate new numbers to identify the remaining new paths.

```
SELECT DISTINCT Id = pairing (1+Pnum, Pnum)
FROM TEMP1
WHERE Pnum = (SELECT MAX(Pnum) FROM TEMP1)
INTO TEMP NewId
```

Step 5

Create the temporary table *VINS2* to account for the edge (a, b) considered by itself as a path.

```
SELECT Pnum = NewId.Id,
       St   = a,
       End  = b,
       IntS = a,
       IntD = b,
FROM NewId
INTO TEMP VINS2
```

Step 6

Create new paths that are constructed by adding the new edge (a, b) in front of an existing path. Every such new path contains exactly (1) all nodes from the existing path and (2) the new edge (a, b).

```
   SELECT *
   FROM (
(1)   SELECT DISTINCT
        Pnum = pairing(R.Pnum, NewId.Id),
        St   = a,
        End  = R.End,
        IntS = R.IntS,
        IntD = R.IntD
      FROM R, NewId
      WHERE R.St = b
    UNION
(2)   SELECT DISTINCT
        Pnum = pairing(R.Pnum, NewId.Id),
        St   = a,
        End  = R.End,
        IntS = a,
        IntD = b
```

```
      FROM R, NewId
      WHERE R.St = b)
  INTO TEMP VINS3
```

Step 7

Create a new view *VINS4* to account for path which are formed by appending the new edge (a, b) to the tail of an existing path. Every such new path contains exactly (1) the nodes from the exiting path and (2) the new edge (a, b).

```
   SELECT *
   FROM (
(1)   SELECT DISTINCT
        Pnum = pairing(R.Pnum, NewId.Id),
        St   = R.St,
        End  = b,
        IntS = R.IntS,
        IntD = R.IntD
      FROM R, NewId
      WHERE R.End = a
    UNION
(2)   SELECT DISTINCT
        Pnum = pairing(R.Pnum, NewId.Id),
        St   = R.St,
        End  = b,
        IntS = a,
        IntD = b
      FROM R, NewId
      WHERE R.End = a)
   INTO TEMP VINS4
```

Step 8

All new paths are now accounted for. We simply insert all of them into R to finish off the update.

```
    INSERT INTO R
      SELECT * FROM VINS1
    UNION
      SELECT * FROM VINS2
    UNION
```

```
  SELECT * FROM VINS3
UNION
  SELECT * FROM VINS4
```

An Example for the Insert Case

Assume that the graph in G contains two edges (x, a) and (b, y). Then the table R is:

Pnum	St	End	IntS	IntD
1	x	a	x	a
2	b	y	b	y

Suppose we want to insert the new edge (a, b). Here is the step-by-step account of the process.

Step 1 Create *V1* which contains one tuple $(6, 1, 2, x, y)$, here $6 = \textit{pairing}(1, 2)$.

Step 2 Create *VINS1*. The first `select` statement produces $(6, x, y, x, a)$ and $(6, x, y, b, y)$. The second `select` produces $(6, x, y, a, b)$. So we get

Pnum	St	End	IntS	IntD
6	x	y	x	a
6	x	y	b	y
6	x	y	a	b

Step 3 Create *TEMP1* as the union of R, *VINS1*, and $DUMMY$.

Step 4 $NewId$ contains $\textit{pairing}(7, 6) = 97$.

Step 5 Create *VINS2*, which contains one tuple $(97, a, b, a, b)$.

Step 6 Create *VINS3* corresponding to paths that start with (a, b). The identifier of the only such path is $\textit{pairing}(2, 97) = 5047$. So we get *VINS3* as

Pnum	St	End	IntS	IntD
5047	a	y	b	y
5047	a	y	a	b

Step 7 Create *VINS4* corresponding to paths that end with (a, b). The identifier of the only such path is $\textit{pairing}(1, 97) = 4948$. So we get $VINS4$ as

Pnum	St	End	IntS	IntD
4948	x	b	x	a
4948	x	b	a	b

Step 8 Insert all of them into R. The updated value of R is

Pnum	St	End	IntS	IntD
1	x	a	x	a
2	b	y	b	y
6	x	y	x	a
6	x	y	b	y
6	x	y	a	b
97	a	b	a	b
5047	a	y	b	y
5047	a	y	a	b
4948	x	b	x	a
4948	x	b	a	b

After that, the transitive closure TC can be extracted as the projection of the new value of R onto the *St* and *End* attributes:

Start	End
x	a
b	y
x	y
a	b
a	y
x	b

Remark

One may notice that path identifiers grow very fast. One can introduce some extra steps to renumber the paths. This can be done as follows.

First we create a mapping *NM3* from the old *Pnum*'s to new consecutive numerical identifiers. This can be accomplished in three extra steps using `GROUPBY` and the standard aggregate function `COUNT` of SQL.

```
CREATE VIEW NM1(Pnum) AS
  SELECT DISTINCT Pnum
  FROM R
```

```
CREATE VIEW NM2(Pnum) AS
  SELECT First  = s.Pnum, Second = r.Pnum
  FROM  NM1 s, NM1 r
  WHERE s.Pnum <= r.Pnum

SELECT First  = NM2.First, Second = COUNT(NM2.Second)
FROM  NM2
GROUPBY First
INTO TEMP NM3
```

NM1 is a copy of all the *Pnum*'s currently in used. *NM2* pairs each *Pnum* to those *Pnum*'s that are less than it. Then we can simply do a `GROUPBY` and `COUNT` on *NM2* to generate the table *NM3* which maps each old *Pnum* to its "rank" (ie. the number of existing *Pnum*'s less than that old *Pnum*.) These ranks can then be used as our new consecutive numerical identifiers. Continuing with our example, this yields:

First	Second
1	1
2	2
6	3
97	4
4948	5
5047	6

Finally, we use a join to apply this mapping to renumber the *Pnum*'s.

```
UPDATE R
  SET Pnum = NM3.Second
  FROM R, NM3
  WHERE R.Pnum = NM3.First
```

This update results in the following value of R, where *Pnum* now takes on small consecutive values.

Pnum	St	End	IntS	IntD
1	x	a	x	a
2	b	y	b	y
3	x	y	x	a
3	x	y	b	y
3	x	y	a	b
4	a	b	a	b
6	a	y	b	y
6	a	y	a	b
5	x	b	x	a
5	x	b	a	b

5 Complexity

We re-iterate the main point of this paper: There is much more to the idea of incremental evaluation than the view that it is merely a means to avoid expensive re-computation. For example, transitive closure 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 recomputation is not even the issue here, for the query is not even do-able in SQL without incremental evaluation in the first place!

Nevertheless, it is still useful to consider the complexity of our incremental evaluation systems. For this purpose it is useful to make comparisons with some typical algorithms that compute transitive closure of graphs *from scratch.* A fairly standard database-style algorithm is based on iterated joins, which essentially repeats the query `SELECT Start=G.Start, End=TC.End FROM G, TC WHERE G.End = TC.Start INTO TC` as many times as there are edges in G. Note that while this query is in SQL, the process that iterates it must be programmed in an external language such as C. The time complexity of this algorithm is $O(n^3)$ for a dense graph having n edges. In the presence of appropriate indices, the complexity can be reduced to $O(n^2 \log n)$. A standard nondatabase-style algorithm is the Warshall algorithm [28]. Note that this is a main-memory algorithm and thus extra work is required if the graph resides in a database. The cost of this algorithm is $O(v^3)$ for a graph having v vertices, which is approximately $n^{3/2}$ for a dense graph having n edges.

Let us begin with acyclic graphs. The cost of inserting a new edge can be estimated in terms of the number of joins as follows (we can ignore the cost of other operations, as the cost of joins dominates the overall cost.) 1 join is required to calculate *TC-NEW*, 1 join is required to calculate *DELTA* (since the existence test has the cost of a join), giving us two joins in total. The cost of deleting an existing edge can also be estimated in terms of the number of joins. 1 join is required to calculate *SUSPECT*, 1 join is required to calculate *TRUSTY*, 3 joins are required to calculate *TC-NEW* (actually, this one can be easily optimized to 2 joins), 1 join is required to carry out the final deletions, giving us 6 joins in total. Assuming that the graph is dense so that the number of edges in G and TC are both approximately n, then the cost of maintaining the transitive closure of acyclic graphs under our setting is at worst approximately $6n^2$, as all relations involved in our 6 joins would also be approximately n edges in size. This is considerably better than the cost of the standard database-style transitive closure algorithm. However, it is slightly poorer than the Warshall algorithm. In the presence of suitable indices (which we can easily create), the cost of a join over a pair of relations of size n is $O(n \log n)$. Then the cost of our incremental evaluation system is reduced to approximately $6n \log n$, which is better than the standard database algorithm with indices as well as the Warshall algorithm.

Let us now consider the undirected graphs. We ignore the cost of maintaining *LESSTHAN*, since in real life, nodes in the graphs are atomic objects like strings and numbers, for which an order can be obtained from the underlying relational database system. The cost of inserting a new edge is estimated as follows. No join is required to calculate *FOREST*, no join is required to calculate *T-STAR*, 8 joins are required to update *THROUGH*, giving us 8 joins in total. The cost of deleting an existing edge is estimated as follows. 1 join is required to calculate *REP-ALL*, 1 join is required to calculate *REP* (assuming the use of *LESSTHAN* is replaced by the appropriate comparison operation of the underlying database system), 4 joins are used to compute *DELTA*, no join is needed to update *FOREST*, 8 joins to update *THROUGH* (this part was not shown), giving a total of 14 joins. Assuming that the graph is dense so that the number of edges in G and TC are both approximately n. Then the number of edges in *THROUGH* is approximately $n^{3/2}$. Even though the joins involve *THROUGH*, a careful inspection shows that only about n of these $n^{3/2}$ in *THROUGH* are involved in these joins, since these joins are always preceded by selections. Thus the cost of maintaining undirected transitive closure under our incremental setting is about $14n^2$, as all the relations involved in our 14 joins would also be about n edges in size. Even for moderate n, this is still better than the cost of the standard database-style transitive closure algorithm. However, it is not as good as the Warshall algorithm. In the presence of suitable indices, the cost of our incremental evaluation can be reduced to approximately $14n \log n$, which is better than the standard database-style transitive closure algorithm as well as the Warshall algorithm.

So we see that our incremental evaluation systems for acyclic and undirected transitive closure have two advantages: They are expressible using nothing more than SQL and they are also relatively more efficient than typical transitive closure algorithms based on re-computation. For the case of arbitrary directed graphs, the complexity of our incremental evaluation system is not so good. In a dense graph, the number of paths is exponential with respect to the number of edges in the graph. So for dense graphs, the size of R, which stores all possible paths, would also be exponential. As a consequence, the worst-case cost in terms of time is also exponential. The point of our incremental evaluation system for the transitive closure of arbitrary graphs is thus a theoretical one: It is possible to compute the transitive closure of arbitrary graphs in SQL in an incremental setting.

6 Theory of Incremental Evaluation Systems

Having discussed the SQL queries of IES for transitive closures, let us give a brief overview of the theory of IES. We first recall the concept of IES. Suppose we have a query Q. An IES($\mathcal{L}$) for maintaining the query Q is a system consisting of an input database I, an answer database A, an optional auxiliary database, and a finite set of maintenance functions that correspond to the different kinds of permissible updates to the input database. These maintenance functions take as input, the corresponding update, the input database, the answer database, and the auxiliary database; and they collectively produce as output the updated answer database and the updated auxiliary database. There are only two requirements: the condition $A = Q(I)$ must be maintained, and the maintenance functions must be expressible in the language $\mathcal{L}$. ($\mathcal{L}$ is called the ambient language of the IES.) We only consider queries from flat relations to flat relations; and in this paper permissible updates are restricted to the insertion and deletion of a single tuple. A further restriction is also imposed so that the constants that appear in the auxiliary database must also appear in the database or in the answer or in some fixed set.

The earliest formulation of IES is [18]; successive refinements were given in [11, 17, 15]. These papers considered the first-order incremental evaluation system, IES($\mathcal{FO}$), which uses first-order logic to express maintenance functions. It is thus equivalent to IES where pure relational calculus is used as the ambient language. A closely related formalism is dynamic first-order, DynFO, of [27]. While DynFO is similar to IES($\mathcal{FO}$) in many aspects, there are some important differences between the two, see [27, 15] for comparison. Here, we will use IES($\mathcal{FO}$) for illustration.

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 relational calculus [1], it can be expressed in $\mathsf{IES}(\mathcal{FO})$. The update function for *even* when a tuple t is deleted from R is given by

$$even^n(1) \quad \text{iff} \quad (R^o(t) \land \neg even^o(1)) \lor (\neg R^o(t) \land even^o(1)).$$

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

$$even^n(1) \quad \text{iff} \quad (R^o(t) \land even^o(1)) \lor (\neg R^o(t) \land \neg even^o(1)).$$

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. 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. That is, each auxiliary relation has at most k attributes. In general, we write $\mathsf{IES}(\mathcal{L})_k$ to mean the subclass of $\mathsf{IES}(\mathcal{L})$ where flat auxiliary relations of arities up to k can be used.

Much is already known about $\mathsf{IES}(\mathcal{FO})$. The transitive closure of acyclic graphs can be maintained in space-free $\mathsf{IES}(\mathcal{FO})$ [13]. The transitive closure of undirected graphs can be maintained in $\mathsf{IES}(\mathcal{FO})_3$ [27] and even in $\mathsf{IES}(\mathcal{FO})_2$ [14]. In a failed attempt to prove the strictness of the $\mathsf{IES}(\mathcal{FO})_k$ hierarchy, Dong and Wong proved that equi-cardinality of relations of arbitrary arities can be maintained in $\mathsf{IES}(\mathcal{FO})_2$ [19]. Dong and Su [14] showed that the $\mathsf{IES}(\mathcal{FO})_k$ hierarchy is strict for $k \leq 2$.

More recently, using a result of Cai [3], Dong and Su showed in [15] that the $\mathsf{IES}(\mathcal{FO})_k$ hierarchy is strict for every k. That is, for every $k > 0$ there is a query Q that can be maintained with the help of auxiliary relations of arity up to k, but cannot be maintained with the help of auxiliary relations of arity up to $k-1$, when the ambient language is the relational calculus. 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 $6k$. Even more recently, Dong and Zhang [20] separated $\mathsf{IES}(\mathcal{FO})_k$ from $\mathsf{IES}(\mathcal{FO})_{k+1}$ using an example query of arity $3k+1$. However, it is open if there is an $\mathsf{IES}(\mathcal{FO})$ for transitive closure of arbitrary directed graphs. It is also open whether the $\mathsf{IES}(\mathcal{FO})_k$ hierarchy remains strict if we restrict to queries having fixed input arity. For example, it is not known if the $\mathsf{IES}(\mathcal{FO})_k$ hierarchy is strict when restricted to graph queries.

Besides these unresolved problems, $\mathsf{IES}(\mathcal{FO})$ has the further problem of not properly reflecting the power of practical relational systems. This is because $\mathsf{IES}(\mathcal{FO})$ uses relational calculus as its ambient language, while practical relational systems use SQL, which is more powerful than relational calculus. This motivated Libkin and Wong to study incremental evaluation systems where the ambient language is $\mathcal{NRC}^{\text{aggr}}$, a theoretical reconstruction of SQL based on a nested relational calculus [23]. We use the notation $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ to denote the incremental evaluation system where both the input database and the answer are flat relations, but the auxiliary database may involve nested relations. We use the notation $\mathsf{IES}(\mathcal{SQL})$ when the auxiliary database is also restricted to flat relations. The rationale for the $\mathsf{IES}(\mathcal{SQL})$ is that it approximates more closely 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.

Many questions about the power of $\mathsf{IES}(\mathcal{SQL})$ have been answered recently. Dong, Libkin, and Wong showed that space-free $\mathsf{IES}(\mathcal{SQL})$ is unable to maintain transitive closure of arbitrary graphs [8]. In a later paper, they also proved that transitive closure of arbitrary graphs remains unmaintainable in $\mathsf{IES}(\mathcal{SQL})$ 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 [9]. On the positive side, Libkin and Wong recently showed that if the bounded degree constraint on auxiliary data is removed, transitive closure of arbitrary graphs becomes maintainable in $\mathsf{IES}(\mathcal{SQL})$ [22]. In fact, this query (and even the alternating path query which is complete for polynomial-time) can be maintained in $\mathsf{IES}(\mathcal{SQL})_2$, because the $\mathsf{IES}(\mathcal{SQL})_k$ hierarchy collapses to $\mathsf{IES}(\mathcal{SQL})_2$, that is, $\mathsf{IES}(\mathcal{SQL})_k = \mathsf{IES}(\mathcal{SQL})_2$ for $k \geq 2$ [22].

Another result of Libkin and Wong [22] states that $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$ and $\mathsf{IES}(\mathcal{SQL})$ are equivalent. That means the restriction to flat tables does not incur a loss in power. Since many problems have a clearer and simpler implementation in $\mathsf{IES}(\mathcal{NRC}^{\text{aggr}})$, 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 $\mathsf{IES}(\mathcal{SQL})$? Results aimed at answering this question have recently become available [24]. On the positive side, all relational queries expressible in second-order logic, and hence having the polynomial-hierarchy data complexity [21], are maintainable in $\mathsf{IES}(\mathcal{SQL})$ in a uniform manner. On the negative side, this is very close to the upper bound on the power of $\mathsf{IES}(\mathcal{SQL})$. From these results and the practical examples from earlier sections, we conclude that practical relational databases, as well as more advanced systems like *Kleisli* [6], possess remarkable power (through maintenance) in a way that was little suspected before.

Acknowledgement

Guozhu Dong acknowledges support by research grants from the Australian Research Council. Jianwen Su was supported in part by NSF grants IRI-9411330 and IRI-9700370. Part of this work was done while Guozhu Dong was at the University of Melbourne and while Limsoon Wong was visiting Bell Laboratories.

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 *Proceedings 6th Symposium on Principles of Programming Languages, Texas*, pages 110–120, January, 1979.

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

[4] C. J. Date. A critique of the SQL database language. *SIGMOD Record*, 14(3):8–52, November 1984.

[5] C. J. Date. Some principles of good language design. *SIGMOD Record*, 14(3):1–7, November 1984.

[6] S. Davidson, C. Overton, V. Tannen, and L. Wong. BioKleisli: A digital library for biomedical researchers. *International Journal of Digital Libraries*, 1(1):36–53, April 1997.

[7] G. Dong and R. Kotagiri. Maintaining constrained transitive closure by conjunctive queries. Proceedings of International Conference on Deductive and Object-Oriented Databases, Switzerland, 1997. Springer-Verlag.

[8] 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, page 8, 1996. Available at http: //www.springer.co.uk /eWiC /Workshops /DBPL5.html.

[9] 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.

[10] 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.

[11] G. Dong and J. Su. First-order incremental evaluation of datalog queries. In Catriel Beeri, Atsushi Ohori, and Dennis Shasha, editors, *Proceedings of 4th International Workshop on Database Programming Languages, New York, August 1993*, pages 295–308. Springer-Verlag.

[12] G. Dong and J. Su. Increment boundedness and nonrecursive incremental evaluation of datalog queries. In *LNCS 893: Proceedings of 5th International Conference on Database Theory, Prague, Czech Republic, January 1995*, pages 397–410. Springer-Verlag.

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

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

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

[21] D. Johnson. *A Catalog of Complexity Classes*, volume A of *Handbook of Theoretical Computer Science*, chapter 2, pages 67–161. North Holland, 1990.

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

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

[24] L. Libkin and L. Wong. SQL can maintain polynomial-hierarchy queries. Technical report, Institute of Systems Science, Heng Mui Keng Terrace, Singapore 119597, 1997.

[25] G. Ozsoyoglu, Z. M. Ozsoyoglu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Transactions on Database Systems*, 12(4):566–592, December 1987.

[26] C. Pang, R. Kotagiri and G. Dong. Incremental $FO(+,<)$ maintenance of all-pairs shortest paths for undirected graphs after insertions and deletions. In *Proceedings of International Conference on Database Theory (ICDT)*, Edited by Buneman and Beeri. Jerusalem, January, 1999.

[27] S. Patnaik and N. Immerman. Dyn-FO: A parallel dynamic complexity class. *Journal of Computer and System Sciences*, 55(2):199–209, October 1997.

[28] R. Sedgewick. *Algorithms in C.* Addison-Wesley, 1990.

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