General Transitive Closures and Aggregate Functions *

Johann Eder
Universität Wien
Institut für Statistik und Informatik
Liebiggasse 4/3-4
A-1010 Wien / AUSTRIA

February 12, 1993

Abstract

General transitive closures are a convenient operation for processing recursive structures with relational languages, because they are easy to understand, efficiently to implement and expressive enough to support a broad range of practical applications. To further extend the expressiveness of general transitive closures, we study the use of aggregate functions together with general transitive closures. While general transitive closures are restricted to express linear recursion, general transitive closures with aggregate functions can be used to express some nonlinear recursions too. We will give some conditions for general transitive closures with aggregate functions to be well formed and bottom up evalueable. We show how these constructs can be integrated in an extended SQL.

Keywords: deductive databases, recursive query processing, SQL, general transitive closure

1 Introduction

There is a common understanding in the research community that relational database systems have reached their limits in supporting the demands for

*This work was partly supported by the Austrian *Fonds zur Förderung der wissenschaftlichen Forschung* under contract P6772P.

new sophisticated information systems. In particular the fact that relational query languages are not computationally complete and the representation of complex objects is cumbersome has triggered a lot of research in deductive databases and object oriented databases.

For introducing deduction we present here a rather pragmatic approach. We believe that many yet unsupported applications demand comparable modest deductive capabilities. Therefore, we will present an extension of SQL for processing of a class of recursive queries rather than introducing general recursive capabilities. Since SQL is the standard query language for relational databases these extensions should be easy to use and to integrate with existing databases. A major advantage of such an approach is that with the increased capabilities of SQL investments in systems and already collected data as well as in training of programmers and end-users can be used and nevertheless new kinds of applications can be built.

Recursion is integrated into the view definition of SQL in form of generalized transitive closures ([7]). A unique aspect of our approach is that it also covers relations containing duplicates (multiset relations) to be compatible with SQL ([8]). As a positive side effect, expressiveness is increased when general transitive closures are extended to multiset relations. The semantics of this new construct is defined in a formal way to open possibilities for automatic query optimization and it is defined procedurally to be better understood by traditional programmers. Since recursively defined views can be used in queries like all other views, end users can take advantage of the increased functionality without knowing anything about recursive queries or deduction. Furthermore, tools built upon SQL like report writers or fourth generation languages can be used unchanged on top of recursively defined views.

In this paper we extend our approach by introducing aggregate functions in the definition of general transitive closures. This extension allows some non linear recursive queries to be expressed while general transitive closures are restricted to linear recursion.

This development differs from the mainstream research in deductive databases where deductive query languages are developed within the paradigm of logic programming (see [4] for an overview) as it follows a pragmatic approach introducing only small extensions and maintaining full upward compatibility with SQL.

In [12] a different approach for introducing general transitive closures as means for recursive query processing is reported. Our approach however is fully integrated into relational languages and does not require the graph metaphore for expressing recursive queries.

Previous proposals for extending SQL with a capability for processing recursive structures like [5] or the tree traversal construct of [11] differ from this approach as they work on limited sets of graphs (or relations) and their semantics is only defined procedurally.

In [1] relational algebra is extended with the operator alpha to express a certain class of recursive queries. For non-linear recursions a special relation-valued attribute delta containing the history of the traversal is introduced transforming intermediate relations to non first normal form.

The approach in [10] also extends SQL and deals with duplicates - like our approach. However, it is based on calculus in a more top down approach while we build upon an algebraic fixpoint operator with a bottom up approach and provide an easier procedural interpretation of general transitive closures.

The remainder of this paper is organized as follows: In section 2 we review previous work on general transitive closures for recursive views in SQL. In section 3 we discuss aggregate functions and define an aggregation operator. In section 4 we define aggregate closures and discuss their properties. In section 5 we extend the view definition of SQL to formulate aggregate closures, and in section 6 we draw some conclusions.

2 General Transitive Closures

Recursion is often introduced into relational query languages by means of a fixpoint operator [2]. This means, that a query can be formulated as lfp (R = f(R)), where R is a relation, f is a relational expression, and lfp is the least fixpoint operator. The semantics of this construct is to evaluate R to the least set fulfilling the equation. This general recursive construct has the disadvantage, that a fixpoint does not always exist, and general recursion can be very inefficient to evaluate. Therefore, it seems more promising, to restrict the expression f so that a fixpoint always exists, and efficient evaluation algorithms can be developed.

For this purpose we start from the well known transitive closures and generalize transitive closures as long as the recursive expressions remain well defined and bottom up evaluable.

The transitive closure of a binary relation R is defined as least fixpoint of

$$V = R \cup comp(R, V)$$

Thereby, comp stands for a composition, which is defined as equi-join with the join attributes projected out.

To overcome some shortcomings in the expressiveness of the transitive closure, the definition can be extended in the following ways:

- The equi-join can be replaced by a theta-join or by a selection on the cartesian product.
- The projection may be extended to a projection-expression where the values of some attributes can be defined by evaluable functions like arithmetic expressions or string expressions.

The notion of a general transitive closure was introduced in [6] and [12]. In [7] it was applied to relational languages.

Definition 2.1 (General transitive closure) *We define the general transitive closure of a relation R by a composition expression compex by the least fixpoint of the following equation:*

$$V = R \cup compex(R, V)$$

Compex stands for a composition expression, which is a selection on the cartesian product of R and V together with a projection which may include functions like arithmetic and string expressions. So compex can be described as projex(select P R x GR)). In other terms, the compex expression can be described as a usual SELECT statement of SQL.

Since general transitive closure is defined by union, R and GR have to have the same schema. So the composition expression has to project the cartesian product of R and GR on this schema. The expressions in this projection allow the computation of attribute values of a tuple in the projected relation in terms of attribute values of it's corresponding tuple in the cartesian product R x GR.

The selection on the cartesian product of R and GR can be regarded as transitivity condition, as it determines, if two tuples can be connected. In the general transitive closure the transitivity condition is more general than just identity of attribute values.

Example : The relation *direct* containing inforamation about direct flights consists of the following attributes: $from - city$ and $to - city$ for the connected cities, *departure* for the time the plane departs, *arrival* for the time it arrives, and *distance* for the distance of the flight. The query, we want to formulate, shall produce a table containing all flight connections between cities together with the total distance of each flight. We define, that there is a connection between two cities, if there is a direct flight between these two cities, or there is a connection from the first city to an intermediate city and a direct flight from the intermediate city to the second city, which departs after the arrival of the connection from the first city.

The relation *connection* (short 'C') is defined as general transitive closure of the relation *direct* (short 'D'), i.e as least fixpoint of the following equation:

$$C = D \cup \pi_{C.from,D.to,C.dist+D.dist} \sigma_{D.from=C.to,D.dep<C.arr} D \times C$$

There are two reasons for extending the concept of general transitive closures to relations containing duplicates:

- Increasing expressiveness. In particular, general transitive closures of multiset-relations allow to process reconvergent structures.
- Making general transitive closure suitable for relational languages which allow duplicates in tables.

The operations of relational algebra extend very natural to multiset-relations. In [8] we formally defined these operations in accordance with the respective definitions in SQL. We will not distinguish between operations on set-relations and multiset-relations, where it is not necessary. For the multiset-union we will use the symbol $\sqcup$, e.g. $[a, a, b, b, c] \sqcup [a, a, b, d] = [a.a, a, a, b, b, b, c, d]$.

Definition 2.2 (General transitive closure of multisets) *The general transitive closure of a multiset-relation R by a (multiset-) composition expression compex is defined by the least fixpoint of the following equation:*

$$V = R \sqcup compex(R, V)$$

The expression compex consists of a multiset-selection on the multiset-cartesian product of R and T together with a multiset-projection which may contain functions like arithmetic or string expressions in terms of attributes of the cartesian product. So compex can be regarded as SELECT statement of SQL without duplicate elimination.

Example : This example deals with the well known parts hierarchy problem or bill of materials problem. A relation *comp* is given with the attributes part, subpart and quantity. A tuple $< a, b, c >$ of this relation means that part a contains c pieces of part b. In this relation a certain part can be subpart of different aggregates. (For example the same type of screw may be used in different machines or even in different subparts of a single machine). We wish to formulate the query which parts in what quantity a given aggregate consists of.

We define the relation *parts* as general transitive closure of multisets of the relation *comp*. So the relation *parts* is defined as least fixpoint of the following equation:

$$parts = comp \sqcup \pi_{comp.part,parts.subpart,comp.quantity*parts.quantity}$$

$$\sigma_{comp.subpart=parts.part} comp \times parts$$

Theorem 2.1 *A least fixpoint for the general transitive closure of multisets always exists.*

We define $\mathcal{P}^m(D)$ as the set of all multisets over the domain D. We define the partial order $\leq$ as follows: $M_1 \leq M_2$, iff all elements of M_1 appear in M_2 with at least the same cardinality. It is easy to see, that $(\mathcal{P}^m(D), \leq)$ is a complete lattice, and that $M_1 \sqcup compex(M_1, M_2)$ is continous and monotone with respect to $\leq$. Therefore, Tarski's fixpoint theorem [13] can be applied and the existence of a least fixpoint is assured.

3 Aggregate Functions

3.1 Definition of Aggregate Functions

In SQL aggregate functions are powerful constructs for formulating queries. In the syntax of SQL aggregate functions are interwoven with projection and arithmetic expressions. To be better able to reason about aggregate functions we introduce a special aggregation operator. This aggregation operator has the property that the result has the same scheme as the 'input'-relation - a property we need for recursive views.

In literature aggregate functions for relational query languages have been introduced in various ways. As we aim at extending SQL we will formalize aggregate functions as they are defined in SQL.

Definition 3.1 (aggregation) *The* syntax *of the aggregation operator agg applied to a relation R with schema S is defined as follows:* $agg_L R : \forall a \in S : a \in L \vee \Theta(a) \in L, \Theta \in \{min, max, sum\}$.

We call L the aggregation list, $G = \{a \in S \mid a \in L\}$ *the grouping attribute(s) and* $A = S - G$ *the aggregated attributes.*

The semantics *of the aggregation operator is defined as follows: Let* $P = agg_L R,$. $\forall t \in agg_L R :$

1. $\exists t' \in R : t[G] = t'[G]$
2. $\forall t' \in agg_L R : t \neq t' \rightarrow t[G] \neq t'[G]$
3. $\forall a \in S, min(a) \in L : t[a] = min(\pi_a \sigma_{G=t[G]} R)$
4. $\forall b \in S, max(b) \in L : t[b] = max(\pi_b \sigma_{G=t[G]} R)$
5. $\forall c \in S, sum(c) \in L : t[c] = \sum \pi_c \sigma_{G=t[G]} R$

Example: Let R be a relation with the attributes a, b, c, and d. $agg_{a,min(b),max(c),sum(d)} R$ would read in SQL as follows:

```
Select a, min(b), max(c), sum(d)
From   R
Group by a
```

3.2 Properties of Aggregate Functions

In this section we will discuss some properties of aggregate functions. For the following propositions let D be domain, and R a set-relation and M, M', N multiset-relations over D. Further let agg_L be an aggregate function for relations over D.

Proposition 3.1 (set result) *$agg_L M$ is a set-relation, irrespective whether R is a set- or a multiset-relation, and the grouping attributes are a key (superkey) of the result relation.*

Proposition 3.2 $\forall t \in D : agg_L\{t\} = \{t\}$

It is easy to see that the application of an aggregate operation on a singleton relation results in this very relation.

Proposition 3.3 (idempotence) $agg_L(agg_L R) = agg_L R$

Each tuple of $agg_L R$ represents a different partition of R. A subsequent application of agg_L keeps the partitioning and Prop. 3.3 follows from Prop.3.2. Hence agg_L is idempotent.

Proposition 3.4 $agg_L(M \sqcup M') = agg_L(M \sqcup agg_L M')$*, and if* $agg_L M = agg_L M'$*, then* $agg_L M \sqcup N = agg_L M' \sqcup N$

It is easy to see, that in the case of multiset-relations and multiset-union the aggregation operation can be applied to a part of a relation first, without changing the result.

Note however, that this proposition does not hold for set-relations and set-union in general, because of the 'sum'-aggregation.

Proposition 3.5 *If L does not contain sum, the following proposition holds: Let $\{\ M\ \}$ be the set of all tuples contained in the multiset-relation M:* $agg_L M = agg_L\{M\}$

This means, that if L does not contain sum, then the aggregate of a multiset-relation is the same, irrespective whether duplicates have been eliminated or not.

Proposition 3.6 $\nexists a \in S, sum(a) \in L \rightarrow agg_L R \cup R' = agg_L R \cup agg_L R'$

If the aggregation contains only min und max, then the aggregation can be applied to a part of a relation first.

Proposition 3.7 *If K is a key of the relation R and* $\forall a \in K : a \in L$*, then* $agg_L(R) = R$*.*

Since K is subset of the grouping attributes, each partition imposed by those attributes consists of exactly one tuple, as the values of the key-attributes are unique.

Proposition 3.8 *The set of grouping attributes A is a superkey in* $agg_L\ R$*.*

All aggregated attributes are functional dependent from the grouping attribute, according to the definition of aggregation.

The propositions above will be needed in the sequel to discuss whether aggregate closures are well formed and bottom up evaluable.

4 Aggregate Closure

4.1 Extending general transitive closure with aggregate functions

The extension of the general transitive closure concept with aggregate operations increases expressiveness, since general transitive closures are restricted to linear recursions, while the introduction of aggregate functions. will allow to express several non-linear recursions.

For aggregate closures the fixpoint with respect to the subset relation cannot be used (since $agg_L R \not\subseteq R, and R \not\subseteq agg_L R$). Therefore, we have to define an different partial order which includes the subset order.

Definition 4.1 ($\preceq_L$) *Let* $r, t \in D$*.* $r \preceq_L t$ *with respect to the aggregator* agg_L *with grouping attribute* $A :\Leftrightarrow$

$$\begin{array}{lllll} \forall a \in S: & a \in L & \rightarrow & r[a] = t[a], \\ & min(a) \in L & \rightarrow & r[a] \leq t[a], \\ & max(a) \in L & \rightarrow & r[a] \geq t[a], \\ & sum(a) \in L & \rightarrow & r[a] \leq t[a]. \end{array}$$

Let $R, T \in \mathcal{P}(D)$*.* $R \preceq_L T$ *with respect to the aggregator* $agg_L :\Leftrightarrow \forall r \in R \exists t \in T : r \preceq_L t$*.*

Note: Since the result of an aggregate operation is a set, we do not have to extend the lattice to multisets.

For the following let G be the grouping attribute of the aggregation L, and $\mathcal{P}(D)^G \subseteq \mathcal{P}(D)$ be the set of all relations over D which have G as superkey.

Proposition 4.1 *$\preceq_L$ is a partial order on $\mathcal{P}(D)^G$.*

It is easy to see that $\preceq_L$ is reflexive, anti-symmetric and transitive.

Note: There is another partial order with reverse inequality in the case of sum. However, for the rest of this consideration we will stay with the above definition for sake of simplicity. The results can be easily transferred to the other partial order.

Theorem 4.1 *$(\mathcal{P}(D)^G, \preceq_L)$ is a complete lattice.*

Proof: With prop. 4.1 we know, that $\preceq_L$ is a partial order on $\mathcal{P}(D)^G$. We now have to show, that for any subset of $\mathcal{P}(D)^G$ inf and sup exist. We define Lmin as an aggregation list, where sum is replaced in L by min, and Lmax as an aggregation list where we replace in L min by max, max by min and sum by max. Let R $\subseteq$ $\mathcal{P}(D)^G.inf(R) := agg_{Lmin}((\bigcup R) \bowtie (\bigcap_{X \in R} \pi_G X), sup(R) := agg_{Lmax} \bigcup R$. It is easy to verify, that inf(R) is a greatest lower bound and sup(R) is a least upper bound.

Definition 4.2 (Aggregate closure) *The aggregate closure ac of a relation R by a composition expression compex, and an aggregate operation agg_L is defined by the least fixpoint of the following equation:*

$$V = agg_L(R \cup compex(R, V))$$

The aggregate closure ac_m of a multiset-relation M by a composition-expression compex and an aggregate operation agg_L is defined by the least fixpoint of the following equation:

$$V_m = agg_L(M \sqcup compex(M, V_m))$$

Example: This example is taken from the anti-trust control problem. A relation *owns* is given with the attributes owner, company and share. A tuple $< a, b, c >$of this relation says that an owner a has a share of c percent of company b. Companies can themselves be owner of other companies. We want to formulate a query to determine which companies are controlled by a given owner. A company is controlled by an owner, if this owner, together with the companies he controls, holds more than 50% of this company. We want to specify the query to derive a relation *controls*, with the attributes owner, company, and share expressing all control - relationships determined by the *owns* relation.

We define a relation controls (short C) as aggregate closure of the relation has-share (short H) by

$$agg_{owner,company,sum(share)} H \sqcup \pi_{C.owner,H.company,H.share}$$

$$\sigma_{C.company=H.owner,C.share>50} H \times C$$

Like for general transitive closures, the procedural definition of the semantics of aggregate closures is given through (naive) fixpoint evaluation.

```
Vold := ∅;
Vnew := R;
while Vnew ≠ Vold do
    Vold := Vnew;
    Vnew := agg_L (R ⊔ compex(R, Vold))
endwhile;
```

This algorithm serves only for a procedural definition of the semantics, a query against an aggregate closure is evaluated with more efficient algorithms like differential fixpoint evaluation [9].

4.2 Monotonicity of Aggregate Closures

General transitive closures have the nice property that the fixpoint always exists, and that this fixpoint is bottom up evaluable. However this does not hold for aggregate closures where the fixpoint iteration may not terminate for three reasons:

1. There is an infinite number of tuples in the result. This may only happen, if attributes in the grouping list are computed by arithmetic expressions in the compex expression. This problem also occurs for general transitive closures.

2. Values of some tuples are infinite. This problem may also occur for general transitive closures, when attributes are computed and no bound is specified for the growth of values for these attributes. (For a discussion thereof see [7])

3. There does not exist a fixpoint. This problem cannot appear for general transitive closure. We will demonstrate this problem through the following example:

   Let R be a relation with the schema {a, b, c}. Let the aggregate closure V of R be defined as $V = agg_{a,b,sum(c)} R \sqcup \pi_{V.a,R.b,R.c} \sigma_{V.b=R.a,V.c>30,V.c<50} R \times V$. Let the relation R consist of the following tuples:

a	b	c
x	y	10
x	z	40
z	y	20
y	z	20

   It is easy to see, that a fixpoint for this equation does not exist. Therefore, we have to derive some conditions which are sufficient that the fixpoint of an aggregate closure exists, and that it is bottom up evaluable.

Since the first two problems were already analyzed for general transitive closures, we will concentrate on the third problem. Unfortunately, it is not possible to define a partial order with respect to which the aggregate closure transformation is monotone. For checking the existence of a fixpoint we, therefore, have to check, whether the transformation is increasing. And we will give sufficient conditions for that. For the following let T denote the aggregate closure transformation, i.e. $T(V) = agg_L(R \cup compex(R, V))$, and V^i is the value of Vold in the fixpoint iteration algorithm after the i^{th} iteration.

First we have to define a new partial order, based on the observation, that T(V) may contain more tuples than V, but 'better ones'. Therefore, the inequalities for min and max in $\preceq_L$ have to be reversed. Note, that $\preceq_L$ is the order for chosing among several fixpoints, while $\preceq_L^+$ is the order for which the sequence $(T^i(\emptyset))$ shall be monotone.

Definition 4.3 ($\preceq_L^+$) *Let $r, t \in D$. $r \preceq_L^+ t$ with respect to the aggregator agg_L with grouping attribute $G :\Leftrightarrow$*

$$\begin{array}{llll} \forall a \in S: & a \in L & \rightarrow & r[a] = t[a], \\ & min(a) \in L & \rightarrow & r[a] \geq t[a], \\ & max(a) \in L & \rightarrow & r[a] \leq t[a], \\ & sum(a) \in L & \rightarrow & r[a] \leq t[a]. \end{array}$$

Let $R, Q \in \mathcal{P}(D)$. $R \preceq_L^+ Q$ with respect to the aggregator $agg_L :\Leftrightarrow \forall r \in R \exists t \in Q : r \preceq_L^+ t$.

Definition 4.4 (increasing) *The transformation T is increasing, iff for all $i \geq 0$: $V^i \preceq_L^+ V^{i+1} = T(V^i)$.*

If the transformation T is increasing, then the sequence $(T^i(\emptyset))$ is monotone, and the problem described above cannot appear, i.e. the fixpoint iteration cannot dangle between two different relations.

To check whether the transformation is increasing one has to analyze, which aggregate functions are used, whether there is one or more aggregated attributes, whether it is a set or multiset aggregate closure, whether aggregated attributes appear in the selection condition, and whether attributes depend on aggregated attributes.

Here we will give a sufficient condition for T being increasing which is easier to check.

Definition 4.5 (tuplewise monotone) *A composition expression compex is tuplewise monotone with respect to $\preceq_L^+$, iff $\forall t, t' \in \mathcal{D} : \{t\} \preceq_L^+ \{t'\}, \forall r \in R : compex(\{r\}, \{t\}) \preceq_L^+ compex(\{r\}, \{t'\})$.*

For the definition of tuplewise increasing we take the order $\preceq_L$ (without +!), so that for example a tuple in T(t) has a higher value in a min attribute than t.

Definition 4.6 (tuplewise increasing) *A composition expression compex is tuplewise increasing with respect to $\preceq_L$,*
iff $\forall t \in \mathcal{D} : \forall r \in R : \pi_{S-G}\{t\} \preceq_{L-G} \pi_{S-G} compex(\{r\}, \{t\})$ or $compex(\{r\}, \{t\}) = \emptyset$.

Theorem 4.2 *If T is a set transformation without the sum - aggregate function, and compex (of T) is tuplewise monotone and tuplewise increasing, then T is increasing.*

Proof by induction. Obviously, $\emptyset \preceq_L^+ R$. We assume that $V^{i-1} \preceq_L^+ V^i$. We have to show that $V^i \preceq_L^+ V^{i+1}$, i.e. $\forall t \in V^i, \exists t' \in agg_L(R \cup compex(R, V^i))$ with $t \preceq_L^+ t'$, which follows from the induction hypothesis, compex being monotone and increasing and the properties of aggregation.

Theorem 4.3 *If T is a multiset transformation and compex (of T) is tuplewise monotone and tuplewise increasing, and all values of attributes aggregated by sum are positive, then T is increasing.*

Proof by induction in analogy to the proof of Theorem 4.2. For the sum aggregated attributes we use the fact that since compex is a tuplewise monotone multiset operation for all sets Q, Q': Q $\preceq_L^+$ Q' implies that $|$ Q $| \leq |$ Q' $|$ and $|$ compex(R, Q) $| \leq |$ compex(R, Q') $|$.

To develop an efficient algorithm for checking whether a transformation is increasing which also covers more cases than those of the previous theorems is subject of current research.

5 Extending the view definition in SQL

The formulation of recursively defined tables by means of general transitive closure or aggregate closure of set- or multiset-relations is proposed to be embedded in the view definition of SQL. The following syntax of the view definition statement is an extension of that in standard SQL.

```
CREATE VIEW <view-name> (<attributed-column-list>)
AS [DISTINCT | ALL] FIXPOINT of [<type>]
              <table-name> [(<column-list>)]
[AGGREGATE  <aggregate-list> ]
BY SELECT <list>
   FROM  <table name>, <viewname>
   WHERE <constraint-list>
```

Example: As an example of this view-definition we formulate the view needed to solve the anti-trust problem of example 4. The base table is the table owns with the attributes owner, company, and share.

```
CREATE VIEW controls (owner, company, share)
AS FIXPOINT OF owns
BY
AGGREGATE  owner, company, sum(share)
SELECT     c.owner, o.company, o.share
FROM       controls c, owns o
WHERE      c.company = o.owner
  AND      c.share > 50
```

Traversal recursions as reported in [12] are special cases of this construct. We will give an example for the formulation of graph related recursions. The relation G with the attributes a, b, w represents a graph such that a tuple $< a_1, b_1, w_1 >$ stands for a directed arc from node a_1 to node b_1 with weight w_1.

Shortest path:

```
CREATE VIEW short (a, b, w)
AS FIXPOINT of G
AGGREGATE a, b, min(w)
BY SELECT G.a, short.b, G.w + short.w
FROM      G, short
WHERE     G.b = short.a
```

In general, that an aggregate closure represents a graph traversal can be determined through an analysis of the appearance of the attributes in the view definition. Let R be the base relation and V the defined view. If the attributes of V (resp. R) can be partitioned into 3 sets A, B, W, such that A $\cup$ B is the grouping attribute of the aggregation, the project-expression list of compex is V.A, R.B, f(V.W, R.W) and the where-condition of compex is V.B = R.A, then the aggregate closure represents a graph traversal problem. Such views can be analyzed using the path-algebra described in [3]. However, this consideration demonstrates that aggregate closures are more expressive than graph traversals.

6 Conclusion

Aggregate closures - an extension of general transitive closures - have been introduced to meet demands for increased functionality of query languages for relational databases. The approach is rather pragmatic as it employs comparable modest extensions to SQL with the aim to support a range of practical applications while maintaining full compatibility with SQL and being secure and easy to understand. It has been shown, that the introduction of aggregate functions in the definition of recursive views increases the expressiveness of general transitive closures. However, this extension has the drawback that the existence of fixpoints is no longer guaranteed, but we gave a sufficient condition for aggregate closures to be well formed and bottom up evaluable.

Further research includes the development of an efficient algorithm for analyzing the existence of a fixpoint of an aggregate closure and for guaranteeing termination of the fixpoint iteration as well as the adoption of efficient fixpoint algorithms for aggregate closures.

References

[1] R. Agrawal. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE Trans. on Software Engineering*, 15(3):335–347, 1988.

[2] Alfred Aho and Jeffrey Ullmann. Universality of data retrieval languages. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 110–120, 1979.

[3] B. Carre. *Graphs and Networks.* Claredon Press, Oxford, 1979.

[4] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases.* Springer Verlag, 1990.

[5] E.K. Clemons. Design of an external schema facility to define and process recursive structures. *ACM Trans. on Database Systems*, 6(2):295–311, 1981.

[6] U. Dayal and J. M. Smith. Probe: A knowledge-oriented database management system. In M. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems.* Springer-Verlag, 1986.

[7] J. Eder. Extending SQL with general transitive closure and extreme value selections. *IEEE Transactions on Knowledge and Data Engineering*, 2(4):381–390, 1990.

[8] J. Eder. General transitive closure of relations containing duplicates. *Information Systems*, 15(3):335–347, 1990.

[9] U. Güntzer, W. Kiessling, and R. Bayer. On the evaluation of recursion in (deductive) database systems by efficient differential fixpoint iteration. In *Proc. 3rd Intern. Conf. on Data Engineering*, pages 120–129, 1987.

[10] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *Proc. of the 16th International Conference on Very Large Databases*, pages 264–277, 1990.

[11] Oracle Cooperation. *SQL*Plus Users's Guide*, 1987.

[12] A. Rosenthal, S. Heiler, U. Dayal, and F. Manola. Traversal recursion: A practical approach to supporting recursive applications. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 166–176, 1986.

[13] A. Tarski. A lattice theoretical fixpoint theorem and it's applications. *Pacific Journal of Mathematics*, n.5, 1955.