Implementing Generalized Transitive Closure in the Paradise Geographical Information System

Biswadeep Nag

Technical Report #1272

June 1995

Implementing Generalized Transitive Closure in the Paradise Geographical Information System

Biswadeep Nag
Computer Sciences Department
University of Wisconsin-Madison

June 14, 1995

Abstract

The generalized transitive closure operator can be used to ask and answer a number of sophisticated queries on a database which is viewed as a directed graph. These queries cannot be expressed in SQL or relational algebra and neither can they be answered by commercially available relational or object-oriented database systems. In this report we describe the design and implementation of the generalized transitive closure operator in Paradise, an experimental geographical information system, a domain in which this operator is especially useful. We show how semi-naive evaluation, a basic technique for evaluating recursive queries, can be extended to compute aggregates and we also give a formal proof of the correctness of our method. We also discuss how selections can be used to optimize the evaluation of the query. The report concludes with a few examples of how these queries can now be expressed in the extended SQL like query language of Paradise and what the results look like.

1 Introduction

1.1 Generalized Transitive Closure

Transitive Closure is a graph operator which computes the set of all reachable paths in a graph G. When this set of paths is computed subject to various selection and aggregation conditions, the computation is referred to as *Generalized Transitive Closure.*

For example, consider a graph representing the information about the flights in an airline database. The vertices represent airports while the edges represent flight connections. Simple transitive closure would give us all the cities (or airports) reachable from an airport through one or more flights. Assuming that an edge also contains a number of relevant attributes like distance, flight cost, carrier name, flight departure and arrival times and so on, generalized transitive closure would allow us to ask more sophisticated questions. For example, we could find the set of all possible flight paths between two airports, the cheapest of these, the shortest of these with regard to time, the shortest of these with regard to distance, only the flight paths with less than 3 flight connections, only the flights where the last connection is through NorthWest, the flights where any stopover delay does not exceed 2 hours, only the flights which do not go through Bangkok and so on and so forth. Using this one operator, we can ask essentially all the reachability queries we would need to ask in practice.

1.2 Applications of Transitive Closure

There are other domains to which generalized transitive closure can be applied. One is the classic bill of materials or inventory of parts problem. Besides these there are of course some toy problems regarding finding all the ancestors of a person, but these usually require only simple transitive closure. What is of most interest to us in this report, however, is the application of generalized transitive closure in the domain of Geographical Information Systems (GIS). GIS's are used by earth, atmospheric and space scientists to store and analyze satellite remote sensed data. Traditionally GIS's have used files as the medium of storage, but recently DeWitt and others have proposed the use of specialized DBMS's as GIS's [3].

There are a number of ways in which generalized transitive closure may be used in a GIS. One is the modelling of irrigation channels as directed graphs to find out the reachability and effectiveness of the irrigation network. Another is to use a GIS as a sophisticated route finder. We can map the highway network of a region on to a graph, and then find the shortest paths between cities and so on. Basically any kind of graph traversal and reachability problem can be reduced to a generalized transitive closure problem.

Currently, the effectiveness of a DBMS in handling GIS queries is measured in terms of how well it performs on the Sequoia 2000 storage benchmark introduced by Stonebraker et al[7] . Query

11 of the benchmark applies transitive closure in a catastrophic situation where there has been a chemical spill close to an irrigation channel, and the query tries to find out which irrigation streams the chemical might flow into.

1.3 The State of the Art

None of the commercially available DBMS's support transitive closure. Perhaps one of the reasons for this is the lack of standard query languages which can express transitive closure. It is only recently that languages like SQL3 [4] and POSTQUEL [5] have been proposed to handle these queries. Among DBMS's, POSTGRES and Illustra claim to be close to achieving the goal of implementing transitive closure, but there have not been any published results. There are a number of experimental systems though, which support general recursive queries. One example is CORAL[6] and there are others like Starburst and LDL. In this respect however, we share the view expressed by Dar [1] and others that all the heavy machinery required to support general recursion is not really necessary for this particular class of recursive queries, and yet, most recursive queries which occur in practice are in fact expressible as generalized transitive closure.

1.4 Overview of the Report

In this report we discuss the major issues which arose during the implementation of generalized transitive closure in the experimental DBMS/GIS Paradise. Our initial goal was to augment Paradise so that it could run Sequoia benchmark query # 11, but finally we were able to go quite some distance beyond that. Paradise can now perform transitive closures together with general concatenation (CON) and aggregation (AGG) functions. The SQL like query language used in Paradise had to be suitably extended for this purpose. Paradise can now answer almost all varieties of general transitive closure queries, including the ones presented in the introductory section of this report.

In section 2 of the report we formally define the problem and in section 3 we describe the intended solution and also the reasons behind why we chose our method of implementation. In section 4 we discuss optimization issues regarding transitive closure queries, and in particular about selection and aggregation. In section 5 we describe the actual implementation in Paradise and finally we conclude in section 6.

2 Problem Definition

2.1 Graph Transitive Closure

A relation R with two (possibly composite) attributes S and T, defined over the same domain, can be represented as a directed graph G in which every tuple of R with values s and t for S and T respectively is represented by an arc (s,t). We denote the other attributes of R (if any) collectively as L.

The *transitive closure* of a graph G is another graph $TC(G)$ such that two vertices s and t in $TC(G)$ are connected by an edge (s,t) if and only if t is reachable from s in the graph G. In that case, we also say that t is in the transitive closure of s. Note that this is equivalent to calculating the transitive closure of the relation R with respect to the equi-join operator on attributes S and T.

S and T are called the *closure attributes* of R, and L the *label attribute* of R as $l \in L$ can be used to label the edge (s,t) if the tuple $(s,t,l) \in R$.

2.2 Generalized Transitive Closure

A generalized transitive closure query Q has the form :

$$Q \equiv \pi\ \delta\ AGG\ \sigma\ CON_{\xi}\ PATHS_{\lambda}\ (R[S,T,L])$$

The following is a description of each of the operators in the above query :

PATHS This is the path enumeration operator.

- λ The path closure predicate. It contains the equality condition on the closure attributes (primary closure condition) and any other selection on the arcs of R participating in the closure.

CON The label concatenation operator. This is a function which takes a set of arc labels $\{L_1, L_2 \ldots L_k\}$ and produces a path label P_L.

- ξ The arc selection predicate which determines to which arcs CON should be applied.

σ This selects a subset of the final set of paths.

AGG The label aggregation operator. Computes an aggregate function on the path label. Can also do a *group by*.

δ The path-set selection operator which selects a path-set based on the value of the AGG operator.

π The projection operator.

Types of transitive closure:

- Complete transitive closure (CTC). Reachability from all nodes of the graph.
- Partial transitive closure (PTC). Reachability from a specified subset S of source nodes. This can in fact be expressed as part of the selection operator σ.

2.3 The Subset which has been Implemented

As part of the implementation we decided to concentrate on a workable subset of the generalized transitive closure operators. This means that all the above operators were not implemented in their most general form. But the implementation endeavours to capture the essential functionality of a general transitive closure query.

In particular, the path closure predicate λ can only be an equality condition on two attributes. CON must be a function which has already been implemented in Paradise. It is not difficult to implement these functions, but they cannot be specified directly by the user. Most, but not all instances of σ are supported. AGG does not do a group by. The δ predicate is not supported. π is not currently supported, but should be easy to implement.

3 The Basic Evaluation Algorithm

3.1 Expressing Transitive Closure in Datalog

Let E(X, Y, C) represent the set of edges in a graph, together with an attribute label C and let T(X, Y, C) be the transitive closure relation.

```
T(X, Y, C) :- E(X, Y, C)
T(X, Y, C) :- T(X, Z, C1), E(Z, Y, C2), C = CON(C1, C2)
```

If we bring in the AGG operator, things become more complicated, and conceptually we have to think of storing the path between the vertices X and Y as P in the relation path(X, Y, P, C).

```
agg_path(X, Y, P, C) :- agg_label(X, Y, C), path(X, Y, P, C)
agg_label(X, Y, AGG(<C>)) :- path(X, Y, P, C)
path(X, Y, P1, C1) :- path(X, Z, P, C), edge(Z, Y, EC),
                              append([edge(Z, Y)], P, P1), C1 = CON(C, EC)
path(X, Y, [edge(X, Y)], C) :- edge(X, Y, C)
```

In the above datalog query, only one path is finally chosen, depending on the value of the aggregate on the set of paths, represented as $AGG(< C >)$. Of course in the case of aggregate functions like SUM, the choice of the path is not important.

3.2 The Choice of the Algorithm

A very large number of transitive closure algorithms have been proposed in the literature. Some of these algorithms have been carried over from graph theory and some have been explicitly designed with databases in mind. The algorithms in the latter class make efforts to reduce the evaluation cost in terms of page I/O. The cost of evaluation varies quite a bit with the size of the set of source nodes, and so it matters whether we are doing complete or partial transitive closure. The algorithms which do well on CTC have a preprocessing step usually involving a topological sort of the graph [2]. This orders the nodes in a way such that consecutively accessed nodes are placed close together on disk.

In our implementation, we assumed PTC. The number of source nodes may be more than one, but in general the selectivity of this initial selection of source nodes will be high. In that case, most of the algorithms do not differ much in performance. The four factors which were kept in mind while choosing the evaluation method were efficiency, generality, ease of implementation and applicability to the database context. Based on these ideas, we decided to implement generalized transitive closure using *semi-naive evaluation.*

Semi-naive evaluation is a general method for evaluating recursive queries. It is an iterative algorithm which makes sure that a fact is not derived more than once using the same derivation. Its performance is also stable over wide ranges of source set selectivities. It is a very simple algorithm which makes good use of existing primitives like joins and selections and makes no assumptions about when and how tuples are brought into memory. This is especially important in the case of a practical DBMS like Paradise where algorithms at this level don't have control over the buffer manager. In terms of our criteria for the choice of the evaluation algorithm, semi-naive fits the bill almost perfectly.

3.3 Semi-Naive Rewriting and Evaluation

The original datalog program for expressing transitive closure is given below. It calculates transitive closure as the set of all vertices in the graph which are reachable from a given vertex.

$$T(X,Y) :- T(X,Z),\ e(Z,Y)$$

Semi-naive rewriting of the above:

$$\delta T^{new}(X,Y) :- \delta T^{old}(X,Z),\ e(Z,Y)$$

The general algorithm for semi-naive evaluation :

$\delta T^{old} = \{$ Initial Set $\}$
while $(\delta T^{old} \neq \emptyset)$
{
  $\delta T^{new} = \delta T^{old} \bowtie G$
  $T^{old} = T^{old} \cup \delta T^{old}$
  $\delta T^{old} = \delta T^{new} - T^{old}$
  $T = T \cup \delta T^{old}$
  $\delta T^{new} = \emptyset$
}

Notice that in the semi-naive rewritten rule, we did not need to use T^{old} anywhere. Therefore we can eliminate it from the algorithm and just use T instead. Simplifying the above algorithm then gives us :

δT^{old} = { Initial Set }

while $(\delta T^{old} \neq \emptyset)$

{

$\delta T^{new} = \delta T^{old} \bowtie G$ ... (1)

$\delta T^{old} = \delta T^{new} - T$ ... (2)

$T = T \cup \delta T^{old}$ ... (3)

}

which is precisely what we are using in this implementation.

There are three major steps in this algorithm. Step (1) is a join. Step (2) is duplicate elimination and step (3) is insertion into a relation. There are also a couple of steps which are not shown explicitly here. One is the intermediate selection which is done between steps (1) and (2) and there is also a final selection which is done after coming out of the while loop.

4 Optimizing Selections and Aggregations

4.1 The Importance of Being Selective

The algorithm which we have used (and which we will describe later in greater detail) has a complexity $O(E * V)$ where the graph has V vertices and E edges. This is of course the worst case complexity when we have to look at every vertex and every edge to compute the result. Obviously, this is rather expensive, specially for large graphs. In practice, however, it might often be possible to restrict our search to a small part of the graph, and this would bring down the evaluation cost dramatically. This is the reason why selections are important, and this has been reiterated by Stonebraker in his analysis of Sequoia benchmark query # 11 [7].

Traditionally, selections in queries are expressed by means of the `where` clause in SQL. We will call these *explicit* selections. In addition to these, transitive closure queries can have implicit selections, where either the graph or the source set or both, have been processed before the query is issued. In that case when, for example, the source relation has been materialized by a previous query, we will say that a selection is *implicit*. In [1] a very elegant formal structure to classify all the various kinds of selections has been presented, and this we shall discuss in the following subsection.

4.2 Classifying and Utilizing Selections

Classification of constraints (C) which can restrict the result of a transitive closure query:

Monotonically Negative If a path P satisfies C, then every subpath of P satisfies C. For example a length bound.

Decomposable P satisfies C if and only if every subpath of P satisfies C.

Extension Monotonic P satisfies C if and only if every extension of P satisfies C. For example PTC.

Non Monotonic If C is none of the above.

The places in the semi-naive evaluation algorithm where a selection can be applied are :

$C_{preprocess}$ As a preprocessing step on the graph before the algorithm starts.

$C_{initial}$ As the first step of the algorithm to restrict the set of source nodes.

$C_{intermediate}$ During recursive enumeration of paths, restrict the set of arcs which are to be included in the transitive closure.

C_{final} After all paths have been enumerated, apply the selection.

It has been shown in [1] that for certain types of constraints, certain types of selections are necessary and sufficient and these conclusions are summarised below. It is to be noted that in the case of a conjunction of constraints, there will be a corresponding superposition of selections.

Constraint	$C_{preprocess}$	$C_{initial}$	$C_{intermediate}$	C_{final}
Monotonically Negative	C	$\emptyset$	C	$\emptyset$
Extension Monotonic	$\emptyset$	C	$\emptyset$	$\emptyset$
Decomposable	C	$\emptyset$	$\emptyset$	$\emptyset$
Non Monotonic	$\emptyset$	$\emptyset$	$\emptyset$	C

In our implementation, we assume that $C_{preprocess}$ and $C_{initial}$ are handled by separate queries executed prior to the transitive closure query. The transitive closure query itself can handle

$C_{intermediate}$ and C_{final}. There is one limitation to this approach. The user has to distinguish what kind of selection a constraint results in, and has to provide this information to the DBMS through the query language. This will be described in detail in the section on implementation experiences.

4.3 Aggregate Retaining Evaluation

4.3.1 Path Condensations

Let $P(s,t) = \{E_k\}, k = 1, \ldots n$ from s to t. Let L_i be the label of E_i. Then $P(s,t)$ has a path label $L(s,t)$ where

$$L(s,t) = CON(L_1, L_2, \ldots, L_n)$$

Storing all the paths generated by the transitive closure algorithm is too expensive. Therefore, we need to have a concise representation of a path which retains only the information necessary to extend the path using transitive closure and this is done using *path condensations.* The condensation of the path $P(s,t)$ is the triple $< s, t, L(s,t) >$. There are restrictions about when the condensation of a path is sufficient and these are detailed in [1].

When we are dealing with aggregates, we have to consider a *set* of paths, and once again, this can be concisely represented by a *path set condensation.* Let $\Psi(s,t) = \{P_k(s,t)\}, k = 1, \ldots, m$ be a set of paths from s to t. Let L_i be the path label of P_i. The path set label of $\Psi(s,t)$ is defined by

$$L_\Psi(s,t) = AGG(L_1, L_2, \ldots, L_m)$$

Even though conceptually we could think that all the paths between two points are first enumerated and then the aggregate is applied, in practice the aggregation can be done incrementally every time we add a path to the set or add an edge to extend the path. This method depends on the following theorem.

Theorem 1 *If CON and AGG form a path algebra, then AGG can be* distributed *over CON.*

The basic semi-naive evaluation algorithm presented in section 3 has to be modified a bit to calculate aggregates correctly. This modified algorithm is called *Aggregate Retaining Evaluation* (ARE) and is also described in [8]. ARE works as follows :

```
DeltaSet = StartSet
ResultSet = Empty          // Contains (vertex, aggregate value) pairs
while (DeltaSet is not empty)
{
      NewDeltaSet = DeltaSet Join BaseSet
                                        // Make NewDeltaSet duplicate free
      If a vertex in NewDeltaSet is already in ResultSet,
            Form a new tuple by aggregating the old and new aggregate values
      If NewDeltaSet has new tuples,
                                        // Either new vertices or new aggregates
            DeltaSet = NewDeltaSet
      Add the tuples in DeltaSet to ResultSet
                                        // Eliminate old tuples if duplicates
}
```

Theorem 2 *ARE is correct, complete and irredundant.*

Proof :

Define *path(X, Y, C)* in ARE and *path(X, Y)* in semi-naive evaluation (SNE) as *corresponding facts*, C being the aggregate value for the path. Obviously, this is not a one-one relation. We represent a derivation as a list of rules denoted by [D].

ARE is correct :

If a fact is deduced by ARE using a certain derivation [D], the corresponding fact is also deduced by SNE using the same derivation [D].

Proof : By induction on the number of steps in the derivation.

Basis : In step 1, ARE finds the nodes which are adjacent to the start node(s). So if it deduces a fact *path(X, Y, C)*, then (X, Y) is an edge, (where C is the weight of the edge) and hence SNE must deduce the fact *path(X, Y)*.

Hypothesis : If a fact is deduced by ARE in step k using a certain derivation [D], the corresponding fact must be deduced by SNE in step k using the same derivation [D].

Step : Let a fact *path(X, Y, C)*, be deduced by ARE in step $k+1$ using derivation [D]. Then a fact *path(X, Z, C1)* should have been deduced by ARE in step k, (using derivation $[D']$) such that *e(Z, Y)* is an edge. By the hypothesis, *path(X, Z)* must have been deduced by SNE in step k using derivation $[D']$. Which means that *path(X, Y)* is deduced by SNE in step $k+1$ using derivation [D] as the edge *e(Z, Y)* is the same for both ARE and SNE.

Therefore by induction, ARE is correct.

ARE is complete :

If a fact is deduced by SNE, a corresponding fact is also deduced by ARE.

Proof : By induction on the number of steps in the derivation of a fact.

Basis : In step 1, SNE finds the edges adjacent to the source nodes, and these are also found by ARE.

Hypothesis : If a fact is deduced by SNE in step k, a corresponding fact is also deduced by ARE in step k.

Step : Consider a fact $path(X, Z)$ deduced by SNE in step $k+1$. This means that a fact $path(X, Y)$ must have been derived by SNE in step k such that $e(Y, Z)$ is an edge. By the hypothesis, some fact $path(X, Y, C1)$ must have been deduced by ARE in step k, and in step $k+1$, ARE will deduce a fact $path(X, Z, C2)$ using $CON(path(X, Y, C1), AGG[path(Y, Z, C2)])$.

Therefore by induction, ARE is complete.

ARE is irredundant :

A fact is never reinferred by ARE using the same derivation.

Proof : By induction on the number of steps taken to derive a fact.

Basis : This is trivially true as the first facts generated could not have been inferred before.

Hypothesis : A fact deduced by ARE in step k cannot have been inferred in step $< k$ using the same derivation.

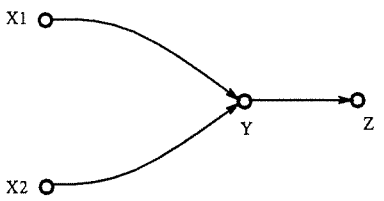

Figure 1:

Step : Suppose a fact $path(X, Z, C1)$ is inferred in step $k+1$. Then there must have been a fact $path(X, Y, C)$ which had been inferred in step k. Now according to the hypothesis, either this fact is completely new, or has a new derivation. In either case, the fact $path(X, Z, C1)$ has a new derivation in step $k+1$.

So by induction, ARE is irredundant.

Aggregation produced by ARE is correct :

Preliminary :

We will first prove that the number of times AGG is applied to a path fact in ARE is equal to the number of derivations of the fact in SNE. This is important for aggregates like *count* and *sum*.

Basis : For the first derivation of this path in SNE, it must also be the first derivation of the path in ARE, otherwise we contradict what we have shown before while proving that ARE is correct. So we could trivially say that the number of times AGG is applied equals the number of derivations of the fact upto which now equals 1.

Hypothesis : If k be the number of applications of AGG to a path fact in ARE, then there are k derivations of the fact in SNE.

Step : If this is the $k+1$th derivation, then by the ARE algorithm, we apply AGG again, and by the induction hypothesis, it has been applied k times before. So we have applied AGG $k+1$ times for the $k+1$ derivation.

Final Proof :

Assume that AGG and CON form a path algebra, i.e., CON distributes over AGG. This means that (Fig. 1) :

$$AGG[CON(path(X1, Y, C1), path(Y, Z, C)), CON(path(X2, Y, C2), path(Y, Z, C))]$$
$$= CON(AGG[path(X1, Y, C1), path(X2, Y, C2)], path(Y, Z, C))$$

Using this, it is possible to show by induction, that ARE does calculate the path with the correct aggregate value. In what follows, *path length* refers to the number of edges and is distinct from the *path weight* which is the CON of the weight of the edges.

Basis : When path length = 1 and there are no parallel edges, AGG is just applied once, and so the path weight is just given by CON which must be correct. If there are parallel edges, we apply AGG two at a time on the parallel edges, and so the result must be correct since AGG is associative.

Hypothesis : At step k, ARE computes the correct AGG value (as known for path lengths $\leq k$) for all vertices reached so far.

Step : Let ARE compute a fact $path(X, Z, C)$ at step $k+1$. We shall show that C is the correct value when AGG is applied over all paths between X and Z of length $\leq k+1$.

Consider all paths between X and Z (of length $\leq k+1$) as shown in Fig. 2. Let the vertices adjacent to Z be called $V_1, V_2, \ldots V_n$. Now consider all paths from X to Z which go through a certain vertex V_i, $1 \leq i \leq n$. By the distributive property described above, the AGG over all these paths is given by

$$AGG[CON(path(X, V_i, C1), edge(V_i, Z, C2))]$$
$$= CON(AGG[path(X, V_i, C1)], edge(V_i, Z, C2))$$

Now that path from X to V_i is of length $\leq k$ and by the induction hypothesis, the correct value of the aggregate for that path, i.e. $path(X, V_i, C1)$, has already been calculated in the previous iteration. Therefore the distributive property ensures that ARE calculates the correct aggregate value.

Now if we consider the set of *all* paths from X to Z, this is equivalent to grouping the paths by the V_i that each of them passes through, first applying the aggregate over each group and then on all the groups together. We have just proved above that the aggregate value for one group is correct. Then by this procedure we get the correct overall aggregate value as the AGG function is associative.

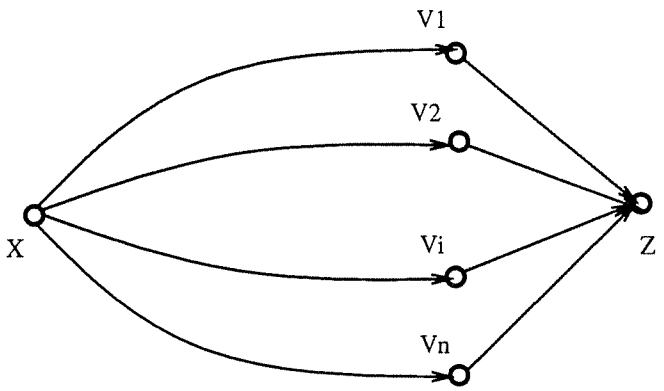

Figure 2:

Finally, the result of all these aggregations will depend upon how many times *AGG* is applied. We have proved before that the number of applications of *AGG* is correct (equal to the number of derivation of a fact). This means that the aggregate value produced by ARE is correct.

5 Implementation Experiences

5.1 If there is Paradise on Earth, ...

Paradise (*Para*llel *D*ata *I*nformation *S*ystem) is an experimental system designed to store and manage large volumes of geographical or satellite remote sensed data [3]. Paradise is object-relational in design with support for user defined ADTs. Among the different datatypes supported by Paradise are points, polylines, arrays and raster images in addition to the usual integer, real and string data types. Each of these types also have methods defined on them to query and manipulate their data. This allows Paradise to implement operations like *overlap* and spatial joins.

As discussed before, transitive closure has a special application in the GIS domain. The base graph for the query is constructed out of a set of edges (stored as polylines) and vertices (stored as points). In general, these ADT's can be n-dimensional, but in this implementation we have concentrated on the 2-dimensional case. Note that this also allows for the existence of self-loops and parallel edges.

Each vertex therefore, is represented as an ordered pair (x, y) while an edge is represented as a set of points $\{(x_i, y_i)\}$. We treat as distinguished the *begin* and *end* points of an edge, denoted as

b and e respectively. The schema for the relevant part of the database looks like this :

```
river   ( b Point, e Point, edge PolyLine, ... )      // Base Graph
startpt ( p Point, length Real )                      // Start Set
```

Notice that the base relation `river` may contain any number of other attributes in addition to the ones involved in closure. `startpt` denotes the start set. In addition to the starting point, it also contains the initial value of the aggregate.

5.2 The Query Language

Paradise supports an extended subset of SQL as the query language. The main extensions to SQL are in the support of new key words like `overlap` denoting spatial joins and also the support for calling methods on an object belonging to an ADT. This latter feature will be extensively used in formulating transitive closure queries.

Standard SQL cannot express recursive queries like transitive closure, though emerging standards for SQL3 attempt to rectify this. Therefore for Paradise we had to design a few extensions to the query language to provide all the information needed to compute the transitive closure of a relation. The grammar for this new part of the language is as follows :

```
<transClosureQuery> ::= SELECT* [<aggClause>]
                        <fromClause>
                        <startClause>
                        <closureClause>
                        <composeClause>
                        [<whereClause>]
<aggClause>      ::= <aggFunc> <attribute>
<fromClause>     ::= FROM <relName>
<startClause>    ::= BEGIN FROM <relName>
<closureClause>  ::= CLOSE <predicate>
<composeClause>  ::= COMPOSE <composeExpr>
<composeExpr>    ::= <relName> '(' <exprList> ')'
```

```
<whereClause>    ::= WHERE <finalPred>  AND <intermediatePred>
                   | WHERE <intermediatePred>
                   | WHERE <finalPred>
<finalPred>      ::= '[' <predicate> ']'
<intermediatePred> ::= <predicate>
```

A few words on the new clauses that have been added to the query language. The *startClause* specifies the relation which contains the start point(s). The *closureClause* specifies the transitive closure predicate which is used for the join in Step 1 of the semi-naive algorithm. The *composeClause* specifies how the new tuples for the *NewDeltaSet* are calculated, and finally the *whereClause* is used to specify the $C_{intermediate}$ and C_{final} selections. Note that the user has to distinguish these two kinds of selections using brackets.

For example, the Sequoia benchmark Query # 11 can be expressed as follows :

```
select*
from river R
start startpt S
closure S.p = R.b
compose S(R.e, R.length.plus(S.edge.length()))
where S.length < 20
```

If we compare it to the POSTQUEL version, shown in [7], the queries look remarkably similar :

```
append* to
temp(GRAPH.identifier, GRAPH.segment, length=temp.length+length(GRAPH.segment))
where end(temp.segment) = begin(GRAPH.segment) and
      GRAPH.identifier notin {temp.identifier} and
      temp.length < 20
```

The change in the query language turned out to be a pretty big task. Changes had to be made to the parser, type-checker and the predicate tree generator. The initial version of the implementation had all the predicate trees hand-coded, but in the final version, all the predicate trees for joins, selections and so on were directly generated from the parser. This will allow the system to handle base relations with arbitrary schemas.

In addition to an SQL command interface, Paradise also has a graphical front-end browser built using the Tcl/Tk toolkit. Using this front end, the user can explicitly issue SQL queries. In addition, the user can also query the database graphically by sketching an area on the screen and zooming in. This zoom in is accomplished by the front end by appending additional overlap constraints to the query. Because of the way we designed the extensions to the query language, these additional constraints appear as intermediate selection predicates in the *whereClause* and can be handled transparently by the transitive closure operator.

5.3 Implementing Transitive Closure

After deciding that semi-naive evaluation (actually ARE, but they are rather similar) would be used, the next question was the choice of the join method to be used in Step 1 of the algorithm. Currently nested loops join and Grace hash join are the only two join methods available in Paradise. Since the join was on a spatial attribute (Point) and presently hash joins are not allowed on spatial attributes, nested loops join turned out to be the automatic choice. There of course exists a spatial join operator in Paradise but the code is rather complex and is not easily adaptable to modification. For transitive closure, we could not use the whole of the join operator, but had to pull out relevant parts of the code and put into a new *transitive closure* operator. The main reason for that is that the nested loops join operator assumes that the result relation will be materialized but this is not the case in transitive closure. In fact the selections and aggregations are done without materializing the result. As regards the overhead of nested loops join, this is considerably reduced by creating a clustered index on the inner relation which in this case is a base graph. The index is of course bulk loaded.

From the very beginning we decided that it was not possible to store all the paths generated by the algorithm as the size of this set grows exponentially with the size of the graph. Therefore we decided only to store path (or path-set) condensations. Also, for the time being we would output only one path per vertex in the result. Or in other words, the result graph generated would be just a tree. As we will show later, this restriction can eventually be removed with a little additional work. So for each vertex V, we store its aggregate value L_v and also the last edge in the path from a source node to V. This must be one of the edges incident on V. All the intermediate relations, δT^{old}, δT^{new}, etc store these 3 pieces of information. The edge is actually stored as a pointer (rid) to the actual tuple. So in general we will deal with 3 schemas, the one for the base relation, the

one for the start relation and the one for the result relation which looks like this (notice the extra aggregate value at the end):

```
result ( b Point, e Point, edge PolyLine, ..., length Real )
```

The next step was deciding how each of the intermediate sets (relations) would be stored in the database. To answer this, we looked at how each set would be accessed. The outer relation (the start set) would be accessed sequentially in the join and so is stored as an ordinary relation. The inner relation, the base graph, is stored as an extent (another name for a Paradise relation) with a clustered index on the join (closure) attribute. For the set T (the result set) we needed an efficient way of checking duplicates. Therefore T was stored as a B^+-tree with the point as the key and the aggregate value and the edge rid making up the element. This enables us to search for a duplicate vertex in $O(log\ n)$ time.

While implementing the transitive closure algorithm, we performed one other optimization. An extent in Paradise is a collection of catalog information plus the file which contains the actual data. The creation and destruction of extents incurs some overhead because of the contention on the catalog which must be locked with an exclusive lock. Therefore we made sure that all the temporary relations shared the same catalog information with the extent for the *start* relation. So creation or destruction of the temporary relations just involved creation or destruction of the data files, without any modifiction of catalog information.

5.4 Implementing Final Selections

Implementing intermediate selections is straight-forward as the selection predicate can be immediately applied to a tuple produced as a result of the join. For final selections we do not know whether a path satisfies the selection predicate until the whole path has been enumerated. For example if we want the shortest path from the source to a certain destination city, we know that a path leads to the destination only after we have reached it. This implies that we must perform the final selection on the final set generated by transitive closure.

However, it is not simple to retrieve the path from the source to the destination from this final set. This is because the final set just contains a collection of edges, and the final selection just gives us the last edge in the path. Therefore we effectively have to do another closure starting from

this last edge, but going backwards this time. This of course is slightly simpler than the forward transitive closure because we do not have to calculate aggregates, but it should be noted that the presence of non-monotonic constraints such as destination selection can considerably increase the execution time of the algorithm.

5.5 Performance Expectations

As we have mentioned before, the most important performance determinant in transitive closure is the effectiveness with which the system handles selections. One of the most widely accepted techniques for propagating selections in recursive queries like this is the use of Magic Sets. In addition, we can make use of projections using *factoring*. In our implementation, we have directly propagated these selections inside the loop which does the semi-naive evaluation so that the selections are automatically done while evaluating the recursive query. Therefore we expect that we will get a performance similar to what we would have got if we had used Magic Sets to propagate selections, used factoring and then used semi-naive rewriting to evaluate the query. This is just about the best that can be done with current technology.

6 Results, Conclusions and Future Work

This work aims to demonstrate the feasibility of implementing generalized transitive closure algorithms in an object-relational database system like Paradise. Paradise is a general purpose system using SQL as the query language and without any prior support for recursive queries. We have extended the query language to express transitive closure queries and have modified the query evaluation engine to processs these queries efficiently using semi-naive evaluation. The algorithms make no assumptions about the form and placement of data and neither do they assume anything about how data is transferred to and from the buffer pool. Now Paradise can answer most transitive closure queries, especially those which occur in the GIS domain.

The primary benchmark for such queries is Query # 11 of the Sequoia 2000. We have been successful in running the query, but unfortunately, we do not know the starting point of the query as this is not mentioned in [7]. So the numbers we obtained have not standardized. Part of the future work involves obtaining the start point and benchmarking our system against this query.

Semi-naive evaluation (and its cousin ARE) are very general algorithms which are relatively easy to implement. In this case we are using them to compute transitive closure but it would be possible to use them to support general recursive queries if desired. We have first proved and then verified empirically that the path labels and aggregate values calculated by ARE are correct. Even though we do not know of any other system which handles *generalized* transitive closure with aggregates without general recursive query support, we have shown that this is is a perfectly viable proposition. Performance-wise we have seen that except for the case when we have final selections, the response time of the system is well within tolerable limits. It might be possible to possible to optimize final selections further.

Finally there is a broad area in which further work can still be done. There are two kinds of results which may be generated by a transitive closure query : one is a set of vertices and the other is a set of edges, the two sets being independent of each other. For example, consider the query which finds the set of paths between two nodes. Currently we are retrieving only one out of the (possibly) many paths. This is because right now, the set of edges has a one to one correspondence with the set of vertices produced by the transitive closure. This however is too restrictive. If we can produce the set of edges independently of the set of vertices, we would be able to overcome this restriction.

To summarize, we have succesfully implemented generalized transitive closure in Paradise. Both the query language and the server have been extended to calculated the closure sets of vertices, edges, path labels and aggregates. It is expected that this added functionality will significantly enhance the usefulness of Paradise to users of Geographical Information Systems.

Acknowledgments

This effort would not have been successful without receiving active support from the people who have made Paradise a reality. I am grateful to Jignesh Patel for suggesting the problem and to Navin Kabra for explaining the design of the parser to me. I am also grateful to Jiebing Yu for answering at least 10 of my questions about Paradise each day: this work owes a lot to his encouragement and advice. Finally I must thank Professor Raghu Ramakrishnan for patiently listening to all my ideas and helping me find my way during crucial stages of the design.

References

[1] Dar S., Augmenting Databases with Generalized Transitive Closure, University of Wisconsin-Madison, *Ph.D. Dissertation*, 1993.

[2] Dar S. and R. Ramakrishnan, A Performance Study of Transitive Closure Algorithms, *Proceedings of SIGMOD 94*, pp. 454 - 465.

[3] DeWitt D.J. et al, Client-Server Paradise, *Proceedings of VLDB 94*.

[4] Melton J. (ed.), (ISO-ANSI Working Draft) Database Language SQL (SQL3), ANSI X3.135-1992, Nov 1992.

[5] Mosher C., The POSTGRES Version 4.0 Reference Manual, Electronics Research Laboratory, University of California, Berkeley, July 1992.

[6] Ramakrishnan R. et al, Implementation of the CORAL Deductive Database System, *Proceedings of SIGMOD 93*, pp. 167 - 176.

[7] Stonebraker M. et al, The Sequoia 2000 Storage Benchmark, *Proceedings of SIGMOD 93*, pp. 2 - 11.

[8] Sudarshan S. and R. Ramakrishnan, Aggregation and Relevance in Deductive Databases, *Proceedings of VLDB 91*, pp. 501–511.

A Sample Output

We will now illustrate the results produced by running some transitive closure queries on an airline flights database. The base graph and the result graph (with thickened lines) are shown as displayed by the Paradise graphical front end. In these queries, the source node is the city "Cal" and all the flights move in a westerly direction.

The following are the relation schemas :

```
flights (edge PolyLine, b Point, e Point, src String, dst String, time Real)
F (p Point, dist Real)
```

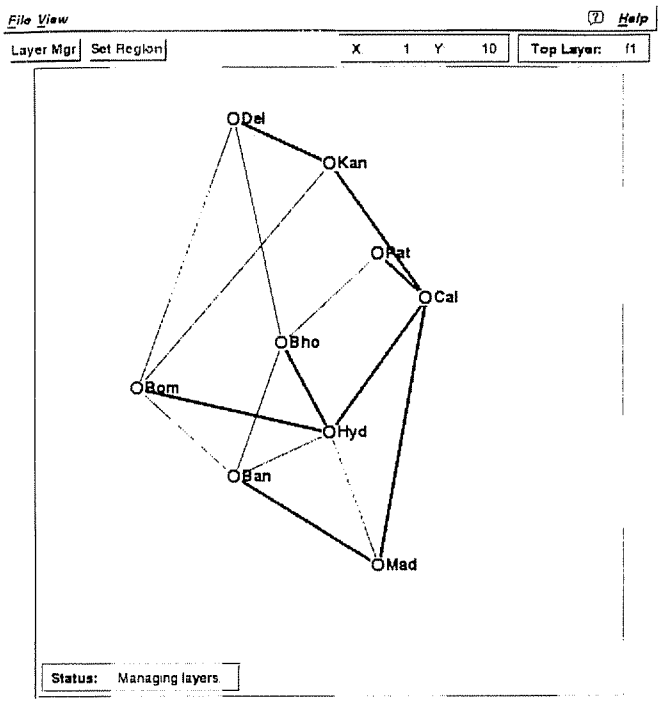

Figure 3: All cities reachable from "Cal"

Query for Figure 3 :

```
select*
from flights
start F
closure F.p = flights.b
compose F(flights.e, F.dist.plus(flights.time))
```

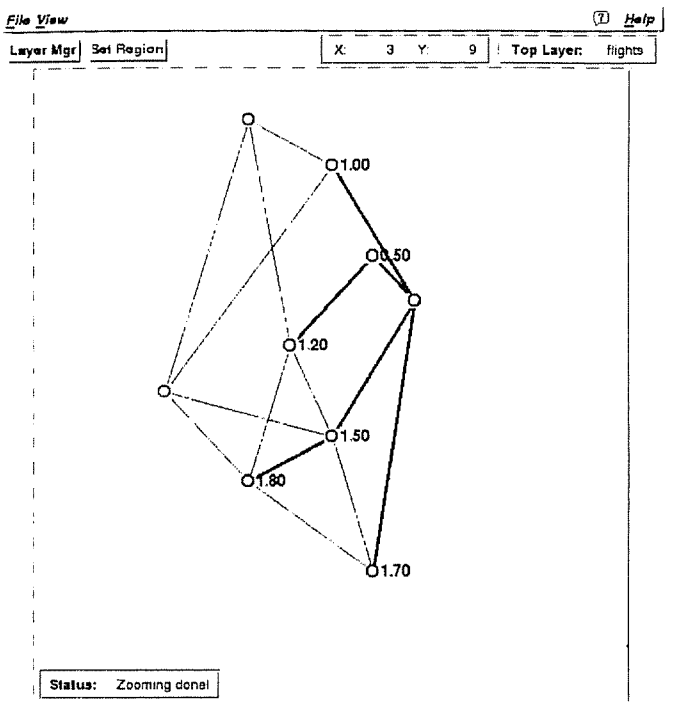

Figure 4: All cities within 2 hours flying time from "Cal"

Query for Figure 4 :

```
select*
from flights
start F
closure F.p = flights.b
compose F(flights.e, F.dist.plus(flights.time))
where F.dist < 2
```

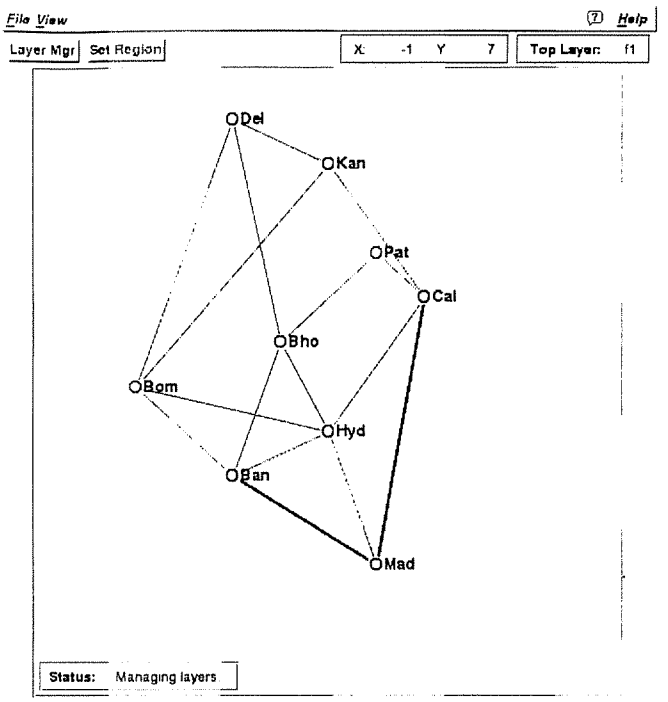

Figure 5: A flight plan from "Cal" to "Ban"

Query for Figure 5 :

```
select*
from flights
start F
closure F.p = flights.b
compose F(flights.e, F.dist.plus(flights.time))
where [F.dst = "Ban"]
```

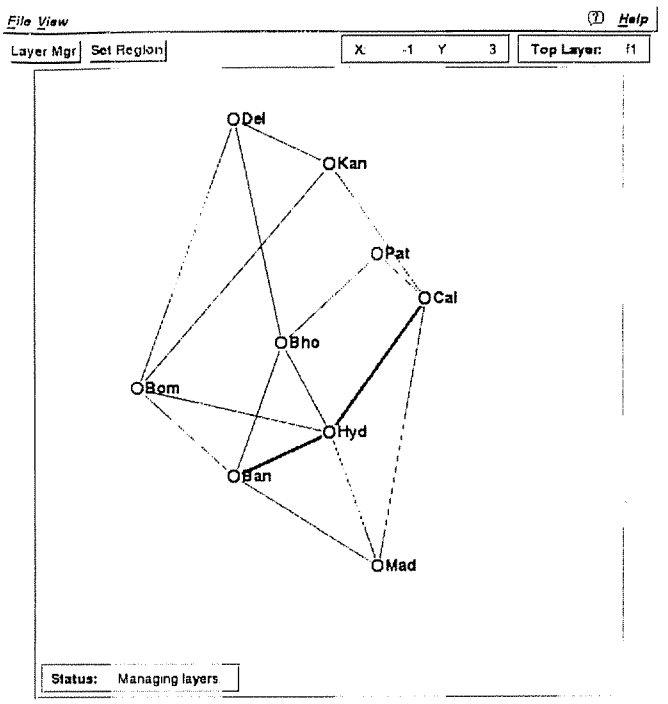

Figure 6: The shortest time flight plan from "Cal" to "Ban"

Query for Figure 6 :

```
select* min(F.dist)
from flights
start F
closure F.p = flights.b
compose F(flights.e, F.dist.plus(flights.time))
where [F.dst = "Ban"]
```

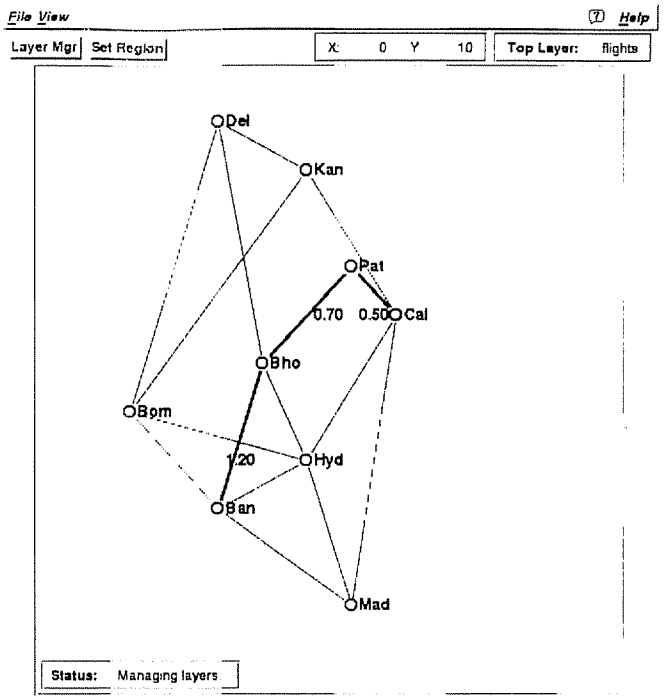

Figure 7: A flight plan from "Cal" to "Ban" such that no flight is more than 1.5 hours and it does not touch "Hyd"

Query for Figure 7 :

```
select* min(F.dist)
from flights
start F
closure F.p = flights.b
compose F(flights.e, F.dist.plus(flights.time))
where [F.dst = "Ban"] and flights.time <= 1.5 and flights.dst != "Hyd"
```

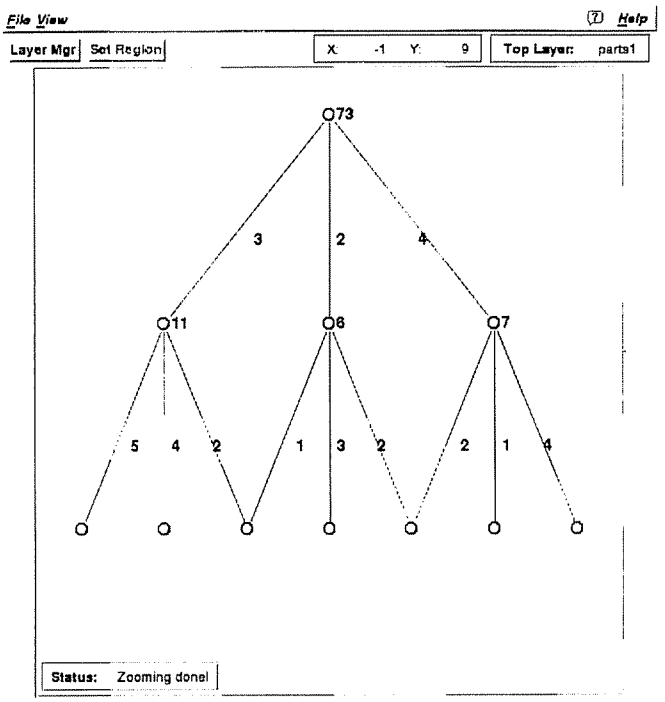

Figure 8: An inventory of parts listing the total number of subparts a part is made of

Query for Figure 8 :

```
select* sum(P.totalParts)
from parts1
start P
closure P.x = parts1.b
compose P(parts1.e, P.totalParts.mult(parts1.numParts))
```