Augmenting Databases with Generalized Transitive Closure

Shaul Dar

Technical Report #1206

January 1994

AUGMENTING DATABASES WITH GENERALIZED TRANSITIVE CLOSURE

by

Shaul Dar

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the
UNIVERSITY OF WISCONSIN — MADISON
1993

Abstract

The need to compute the *generalized transitive closure* (GTC) of a graph arises in diverse database applications such as finding the cheapest flight sequence between two cities, or computing the bill-of-materials of a complex part; however, current database systems do not support GTC queries.

We develop a framework for augmenting databases with generalized transitive closure functionality. This framework includes an algebraic model, language extensions, optimization techniques and efficient algorithms for the formulation and evaluation of GTC queries. The *algebraic query model* supports both path enumeration and path aggregation queries and allows selections on arcs, paths and sets of paths. The SQL/TC *language* extends SQL with the ability to express such queries in a declarative and concise fashion. The answer to a query may include the sequence of arcs in a path, or the aggregation of information for different paths between the same endpoints. We illustrate the expressive power of SQL/TC through many examples and contrast it with earlier proposals, including the proposed extension to ANSI/SQL. We investigate the *optimization* of GTC queries involving selections and label computations, and describe techniques to reduce both the number of paths generated and the space required to store these paths. We present a taxonomy of path problems based on algebraic properties of the label computation functions, and identify characteristics of transitive closure algorithms that make them suitable for solving path problems. We survey the field of transitive closure *algorithms* proposed in the database literature. We start with reachability algorithms and then explore how these algorithms may be extended to evaluate general and partial transitive closure queries. We perform a comprehensive *performance study* of reachability algorithms for the computation of the complete and partial transitive closures of large graphs. We identify the factors that influence the I/O behavior of the algorithms in this spectrum of queries. We also take a critical look at the evaluation methodology for transitive closure algorithms; we demonstrate that cost metrics based on tuple or successor list operations, used in many previous studies, cannot be reliably used to understand the performance of the algorithms when I/O cost is at a premium.

Acknowledgements

This thesis was hard work! I suspect (or would like to believe) that this is the case for other Ph.D. students as well. I consider myself lucky to have worked with people who not only taught me a great deal about computer science, but also kept encouraging and supporting me throughout the years that I spent working on my dissertation at the University of Wisconsin and at AT&T Bell Labs.

Rakesh Agrawal had been my ''de-facto'' advisor for most of my thesis work. Soon after I arrived at Bell Labs, Rakesh left AT&T and joined IBM Almaden in California. Yet, we continued to work together, and Rakesh kept pushing me to go on. He came to both my preliminary and final defense exams in Wisconsin and I am grateful to him. I look up to Rakesh for doing original, interesting, and useful research, and I hope I learned from him to look critically not only at other people's work, but more importantly, at my own.

After returning to Wisconsin, I had the good fortune of working with Raghu Ramakrishnan on the latter part of my thesis and having him as my thesis advisor. I would first like to say that, contrary to common belief, I think Raghu's good jokes outnumber the bad ones. Jokes aside, Raghu is a great advisor — he is very involved in his students' work, is always enthusiastic, and he finds time in his overloaded schedule to devote to his students. Working with him was truly enjoyable. Several other professors at the C.S. department contributed to my work. I initially started looking into the topic of my thesis as a database class project, suggested by Mike Carey. Mike followed my thesis as it evolved, shedding much needed red ink on the drafts of several chapters. Larry Travis was my advisor at the UW while I was at Bell Labs and was always supportive of my work and willing to help in any way possible. I also thank Yannis Ioannidis for his feedback, and Jeff Naughton for sitting in on my defense committee.

Between 1988 and 1992, I spent almost three and a half years at AT&T Bell Labs in Murray Hill, New Jersey. For the majority of this time, I was working on the Ode object-oriented database project with Narain Gehani. Narain has been a good friend, encouraging when I made progress and honestly critical when I was going the wrong way. I learned a lot from him about programming languages and systems research, and particularly about following a research program through — pushing an idea all the way from design to prototype to a working system. I also had the privilege of collaborating with H. V Jagadish on both transitive closure and OODB research. During most of my stay in New Jersey, when my advising committee was distributed between California and Wisconsin, forwarding my thesis was a frustrating experience. I am thankful to Jag for helping me keep moving. To the extent that I obtained some proficiency in ''hacking'' code quickly, this is largely due to Jacques Gava, my office mate at Bell Labs. I also enjoyed interacting with Alex Biliris, Inderpal Mumick, and many of the interesting people who work at Bell Labs. I look forward to rejoining the database research group at Murray Hill in a few weeks.

During my last year at the university of Wisconsin I shared an office with Divesh Srivastava, with whom I had many interesting and instructive conversations on logic programming, philosophy, and Indian cooking. I benefited from and enjoyed my interactions with the students in the database group at the UW, including Paul Bober, Kurt Brown, Mike Franklin, Joe Hellerstein, Mark McAuliffe, Manish Mehta, Praveen Seshadri, Odysseas Tsatalos, Seth White, and Janet Wiener, among others. Linda Winger helped me use her transitive closure algorithms implementation and promptly answered many questions while I was learning the code. I also appreciate Mike Litzkow's help in using Condor, and the assistance of the Computer Science Lab personnel, who were always friendly and quick to the rescue.

I defended my thesis early on a Friday morning after working on my talk all night (such last-moment panic is a time-honored Wisconsin tradition that I did not dare to break). I got home around noon and fell dead asleep. A few hours later, I was awakened by the ring of the telephone. My brother and sister called from Israel, where it was midnight, to find out how I did. I told them, and asked them to let my father know too. Moments like that make it all worth it. Maybe...

Contents

Chapter 1

Introduction

1.1 Motivation

Transitive closure is an operator on graphs that establishes connections (paths) between nodes in the graph. The need to compute the transitive closure of a graph arises in diverse database applications. For example, given a graph representing an airline database, with nodes representing airports, and arcs representing flight connections, we may wish to find a flight sequence between two airports. Given a graph representing an assembly database, with nodes representing parts, and arcs representing part inclusion, we may wish to find all subparts required to construct a complex part. *Generalized transitive closure* is an extension of the transitive closure operator with computation over labels (properties) associated with the arcs in the graph. For example, in the airline database we may want to find the *cheapest* flight sequence between two airports, while in the assembly database we may need to know the *quantity* of each subpart needed to construct the complex part. Many such problems of practical interest can be expressed as generalized transitive closure queries (aka. *path problems*).

Current commercial database management systems do not support generalized transitive closure queries — their query languages cannot express such queries, and their optimizer and evaluation engine are not equipped to process these queries. A user may formulate a generalized transitive closure query using an embedded query language in a relational system, or a database programming language in an object-oriented system. In either case, however, the formulation of the query would be procedural, and the database system would not be able to optimize the query and evaluate it efficiently.

The goal of this thesis is to provide a comprehensive framework for augmenting a database system with generalized transitive closure functionality. Our framework includes the following components.

- **Query language extensions**, allowing a user to formulate generalized transitive closure queries declaratively.
- **Optimization techniques**, assisting the query optimizer in transforming the user's query into a more efficient form.
- **Evaluation algorithms**, enabling the database engine to efficiently answer the query.

We present our framework as an extension to relational database systems, and in particular the SQL query language. This choice was made because most commercial database systems today are relational and use SQL as their data definition and manipulation language. However, most of the material presented in this thesis is equally applicable to other database models.

1.2 Outline

In Chapter 2 we introduce a model for generalized transitive closure queries. We define and illustrate the operators that are used to formulate a query, and in particular, to express *selections* and *label computations*. For example, in the airline database selections may be used to find flight sequences from Jerusalem to Santiago with either Delta or TWA airlines. Label computations may be used to calculate the total distance or average connection time of flight sequences satisfying these constraints. After presenting the query operators, we map the operands manipulated by the operators — labeled arcs, paths, and sets of paths in a graph — into tuples and relations in a database. We then develop a taxonomy of path problems based on algebraic properties of the label computation functions, and, in the process, provide many examples to illustrate the utility of the generalized transitive closure operator.

In Chapter 3 we present an extension to SQL that permits the user to pose generalized transitive closure queries. The answer to a query may include the start and end nodes of a path as well as the sequence of arcs in the path. It may also include the aggregation of information for different paths between the same endpoints. Our notation preserves the spirit of SQL, and allows a declarative and concise formulation of transitive closure queries. We describe the syntax and semantics of our extension, which we call SQL/TC, both formally and through many examples. We also provide a detailed comparison with other proposals for extending SQL with recursive capability, and in particular with a proposed extension to ANSI/SQL.

In Chapter 4 we investigate the optimization of generalized transitive closure queries with respect to the specified selections and label computations. We propose two complementary optimization techniques: i) applying selections on paths during the closure computation, so that paths that are not in the result, and that are not needed to compute the result, are pruned as early as possible, and ii) representing paths that are in the result, or that are needed to compute the result, in a condensed form. The condensed representation records the information that is necessary for the specified computations and selections to be performed. The combined impact of these two techniques is that the number of paths generated during the closure computation and the storage required for each path are reduced.

In Chapter 5 we examine many of the algorithms proposed in the literature to compute transitive closure in a database context. We use algorithms for computing the simple transitive closure of the whole graph, i.e., ''full reachability'', as a starting point. We then discuss how these algorithms may be extended in two directions: i) to compute generalized transitive closure queries, and ii) to compute partial transitive closure queries, i.e. queries that specify some selection on the source nodes. Given the plethora of transitive closure algorithms, we classify them into four families, based on their underlying data representation: iterative, matrix-based, graph-based and hybrid algorithms. We illustrate the evolution of algorithms in each family, as well as the relationship between algorithms in different families.

Chapter 6 provides a comprehensive performance study of a restricted class of algorithms, namely ''reachability'' algorithms. We compare the the page I/O cost of the algorithms for the computation of full transitive closure or partial transitive closure on a range of input graphs. Our investigation includes graph parameters such as the density and ''locality'' of arcs in the graph, query parameters such as the selectivity of the query, and system parameters such as the buffer size and the page replacement policy. We show that significant cost tradeoffs exist between the algorithms in this spectrum of queries, and identify the factors that influence the performance of the algorithms. We also take a critical look at the evaluation methodology and cost metrics used in previous performance studies of transitive closure. We demonstrate that tuple or successor list based cost metrics are unreliable indicators of the I/O cost of the algorithms, which is the dominant factor in the computation of the transitive closure of a large graph.

Finally, in Chapter 7 we present our conclusions and highlight the contributions of our work.

Chapter 2

Model of Computation

This chapter provides the theoretical basis for augmenting a database with a generalized transitive closure operator. After giving some background in Section 2.1, we present in Section 2.2 a model for generalized transitive closure queries. The model incorporates the generation of the transitive closure graph and the selections and label computations that may be performed during the computation of the closure. In Section 2.3 we tie the query model to the relational database model by showing how the elements of the query model — labeled arcs, paths, and sets of paths — may be represented as tuples and relations. In Section 2.4 we develop a taxonomy of path problems (generalized transitive closure queries) based on properties of the label computation functions. In the process, we give many examples to illustrate the utility of the generalized transitive closure operator. Finally, in Section 2.5 we discuss related work.

2.1 Background and Notation

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 from node s to node t. We denote collectively the other attributes of R (if any) by L. Values of the L attributes of R are represented by labels attached to the arcs of G. We use the notation $E(s,t)$ to denote an *arc* from node s to node t. A *path* from node s to node t, $P(s,t)$, is an ordered set of arcs $\{E_k(source, destination)\}$ for $k = 1, \ldots, n$, such that $s = E_1.source$, $E_1.destination = E_2.source$, $\cdots$, $E_n.destination = t$. A *path-set* from node s to node t, $\psi(s,t)$, is a set of paths from s to t, that is, $\psi(s,t) = \{P_k(s,t)\}$ for $k = 1, \ldots, m$. The transitive closure of R with respect to S and T, TC, corresponds to the transitive closure graph of G, $TC(G)$, which is a graph with the same nodes as G, in which there exists an arc from a node s to a node t iff there exists a path in G from s to t.

We call S and T the *closure attributes* of R, L the *label attributes* of R, and G the *underlying graph* of R. The value of a label of an arc in $TC(G)$ is derived from the values of labels of arcs in the corresponding path-set in G, as will be explained shortly.

If arc (s, t) is in G, we say that t is an immediate successor (or child) of s and that s is an immediate predecessor (or parent) of t. If arc (s, t) is in $TC(G)$, we say that t is a successor (or descendant) of s and that t is a predecessor (or ancestor) of s.

2.2 Query Model

A *generalized transitive closure* query Q has the form

$$Q \equiv \pi_A \; \delta \underset{Y,Z}{\mathrm{AGG}} \; \sigma \underset{X,\xi}{\mathrm{CON}} \; \underset{\lambda}{\mathrm{PATHS}} \; (R\,[S,T,L])$$

R is the input relation to the query. **PATHS** is a *path enumeration* operator. **CON** is a *label concatenation* operator. **AGG** is a *label aggregation* operator. π is a *projection* operator. The Greek letters λ, ξ , σ and δ represent selections applied to arcs, paths, or path-sets. Each selection has an associated predicate, which is an expression that yields a boolean value. We do not introduce a separate symbol to denote the expression used as the selection predicate. The upper case letters S, T, L, A, X, Y, Z represent attributes. **PATHS** and R must be present, while the other operators are optional.

As an example, consider relation Flights (Src, Dest, D_Time, A_Time, Airline, Price). We are interested in flights from New York to Tasmania. Since New York has three airports, we wish to separately find the cheapest flight sequence from each airport to Tasmania, but only if the total cost is less than 2500 dollars. In computing the cost, we can consider KLM connections to be free, since we have many frequent flier miles with this company. This query would be expressed algebraically in our model as follows:

$$\pi_{Src,MCT}\ \delta_{MCT<2500}\ \underset{CT,\,Src}{MCT:=MIN}\ \sigma_{\substack{Dest='Tasmania'\ \&\\ Src\in\{'JFK','LGA','EWR'\}}}\ \underset{\substack{Price,\\ Airline <> 'KLM'}}{CT:=SUM}\ \text{PATHS}_{\substack{Dest\,=\,Src\ \&\\ A\_Time\,<\,D\_Time}}\text{(Flights)}$$

The **PATHS** operator, with the specified λ closure condition $Dest=Src\ \&\ A\_Time<D\_Time$, forms flight sequences such that the destination of each flight is the source of the next flight, and each flight arrives before the following flight departs. The **CON** operator SUM finds the total cost of each flight sequence, using a ξ arc selection predicate $Airline<>'KLM'$ to exclude KLM flights from the total. We denote (using :=) the total cost of a flight sequence by CT. The σ path selection predicate $Dest='Tasmania\&Src\in\{'JFK','LGA','EWR'\}$ selects those flight sequences that lead from one of the New York airports (Kennedy, La Guardia, and Newark) to Tasmania. The **AGG** operator MIN partitions the flight sequences according to their source and destination, and computes the cost of the cheapest flight sequence separately for each partition. We denote that minimum cost total (using :=) by MCT. The δ path-set selection predicate $MCT<2500$ discards those (cheapest) flight sequences whose cost exceeds 2500 dollars. Finally, the projection operator π projects the path sets on the source and total cost attributes.

We now describe formally the operators that constitute a generalized transitive closure query and their semantics.

Path Enumeration Operator PATHS

The **PATHS** operator generates the set ψ of all paths in G using a *closure predicate* λ. Given the Flights relation above, we may compute the closure of this relation using Src and Dest as the closure attributes. Each path in the generalized transitive closure of relation Flights is a flight sequence between two airports.

Closure Predicate (λ)

The λ predicate selects arcs from R and determines how these arcs are to be composed into paths. Two types of conditions can appear in the closure predicate λ. The first type of condition specifies constraints on consecutive arcs in a path. We require at least one of these conditions be an equality condition between the values of consecutive arcs. The conjunction of such equality conditions is called the *primary closure condition.* The primary closure condition specifies the closure attributes of R. A primary closure condition with more than one conjunct specifies closure over composite attributes S and T. For example, the primary closure condition for the Flights relation might be that the destination of each flight be equal to the source of the next flight. An additional constraint on consecutive arcs might be that the arrival time of each flight be at least an hour before the departure time of the next flight.

The second type of condition restricts the arcs of R that should participate in the closure. For the Flights relation we might require that all flights be with United Airlines, or that no flight go through Chicago.

Label Concatenation Operator CON

The **CON** operator is used to obtain the label values of paths in ψ. It specifies label concatenation functions $CON_1, \cdots, CON_k$. Each CON_i is a function from a list of values to a single value. CON_i has associated with it an attribute $X_i \in L$, and generates a new (path) label attribute PL_i. The value of the PL_i attribute for each path P_j, that is, the *path label* $L_i(P_j)$, is computed by applying CON_i to the values of the X_i labels of the arcs in P_j. As an example, for the Flights relation we may compute the total cost of each sequence of flights. To do so, we apply the SUM function to the set of Price labels of the arcs in each path.

Arc Selection Predicate (ξ)

An *arc selection* predicate ξ_i may be associated with a label concatenation function CON_i. If ξ_i is specified, CON_i is applied only to those arcs that satisfy the selection criterion ξ_i. For example, in the Flights relation we might compute the average cost of flights with United Airlines. In this case, CON is the AVG function, and it is applied to the set of Price labels of the arcs whose Airline label is ''United''.

Path Selection Predicate (σ)

The σ predicate selects a subset ψ' of the paths in ψ. The σ predicate may refer to path labels of paths in ψ. If σ is omitted, then $\psi' = \psi$. For the Flights relation we might be interested in all flights from London to Auckland whose total cost is between 2000 and 3000 dollars. In this query, σ is a conjunction of three conditions. The first condition restricts the source of the flight sequence to London. The second condition restricts the destination of the flight sequence to Auckland. The third condition restricts the value of the path label representing the total cost of the flight sequence.

Label Aggregation Operator AGG

The **AGG** operator is used to obtain the label values of path-sets in ψ'. It specifies label aggregation functions $AGG_1, \cdots, AGG_l$. Each AGG_i is a function from a bag of values to a single value. AGG_i has associated with it an attribute $Y_i \in \{S, T, PL\}$, and generates a new (path-set) label attribute PSL_i. The value of the PSL_i attribute of the path-set ψ', that is, the *path-set label* $L_i(\psi')$, is computed by applying AGG_i to the values of the Y_i labels of the paths in ψ'. As an example, for the Flights relation we may compute the minimum cost of flying from London to Auckland. The MIN function is applied to the set of path labels representing the total cost of each qualifying path.

One may additionally attach to **AGG** a partitioning expression Z, which specifies a non-empty set of attributes $Z_1, \cdots, Z_p \in \{S, T, PL\}$. The *group-by* attributes $Z_i, \cdots, Z_p$ must be distinct from the *aggregated attributes* $Y_1, \cdots, Y_l$. If Z is specified the path-set ψ is first partitioned into equivalence path-sets $\psi'_1, \psi'_2, \cdots, \psi'_l$ according to the values of the composite attribute Z on the paths in ψ'. Then, each path label

$L_i(\psi'_j)$, is computed by applying AGG_i to the values in the Y_i attribute of the paths in ψ'_j. For the Flights relation, instead of finding a single cheapest flight from London to Auckland, we may wish to find the cheapest flight to Auckland from each of the London airports. The airport name in this case is the partitioning expression Z. The flight sequences are first partitioned according to the London airports. The MIN function is then applied to the set of total cost path labels in each partition.

If **AGG** is omitted, each path $P_j \in \psi'$ becomes a trivial path-set ψ'_j, with $L_i(\psi'_j) = L_i(P_j)$ and $PSL_i = PL_i$ for $i = 1, \cdots, l$.

Path-Set Selection Predicate (δ)

When a partitioning expression Z is specified, δ may be used to select a subset ψ'' of the path-sets in ψ'. The δ path-set selection predicate may refer to path-set labels of path-sets in ψ'. If δ is omitted, then $\psi'' = \psi'$. For the Flights relation we might be interested in all airports in London from which the minimum cost flight to Auckland is priced below 2000 dollars. The flight sequences are first partitioned by their source, and a path-set label representing the minimum-cost flight in each partition is computed. Finally, only those partitions whose minimum-cost flight path-set label satisfies the condition are selected.

Projection Operator π

π_A reduces the representation of path-sets in ψ'' from the attributes in $\{S, T, PSL\}$ to the attributes specified in A. In the last example, we computed a set of paths from London airports to Auckland. We may project this path-set on the source airport and the cost of the cheapest flight from this airport to Auckland.

Let X be the multiset $X_1, \cdots, X_k$, and Y be the multiset $Y_1, \cdots, Y_l$. The relationship between the attribute sets specified in the model is summarized by:

$$X \subseteq L, \quad Y_i \in \{S, T, PL\}, \quad Z \subseteq \{S, T, PL\} - Y, \quad A \subseteq \{S, T, PSL\}$$

This model is a declarative specification of a generalized transitive closure query. It is the responsibility of the query optimizer to optimize the evaluation of a generalized transitive closure query using transformations such as applying selections as early as possible and attempting to compute the label values "on the fly". Chapter 4 contains our results in that direction.

2.3 The Transitive Closure Relation

The transitive closure computation can be thought of as resulting in a non-1NF relation, called the *closure relation*. The schema of the closure relation is defined as follows. Let Q be a generalized transitive closure query over relation R. Let S and T be the closure attributes of R, and $L_1, \ldots, L_n$ be the label attributes of R. Assume the **CON** operator specified in Q defines new label attributes $PL_1, \ldots, PL_k$. The transitive closure relation consists of the attributes S, T, $PL_1, \ldots, PL_k$ and the special attribute PATH. Each PATH value is an ordered relation that consists of the attributes S, T, $L_1, \ldots, L_n$.

A PATH relation represents a path in the graph. The order of the tuples in a PATH relation corresponds to the sequence of arcs in the corresponding path. When two tuples r' and r in the closure relation have identical values in their closure attributes, but different PATH values, we retain both tuples in the closure. These tuples represent alternative paths between the same two endpoints. However, to prevent infinite recursion when the underlying graph is cyclic, if r'.PATH $\supseteq$ r.PATH, then only r is retained in the closure, that is, we allow only simple paths [AHU75].

As an example, consider the relation and underlying graph of Figure 2.1. The corresponding closure relation, with a path label Tot_Dist representing the length of each path, is given in Figure 2.2. The Tot_Dist value is calculated by applying the SUM function to the Distance values of tuples in the corresponding PATH relation.

R		
Src	Dest	Distance
a	b	2
b	c	5
c	d	3
a	c	6

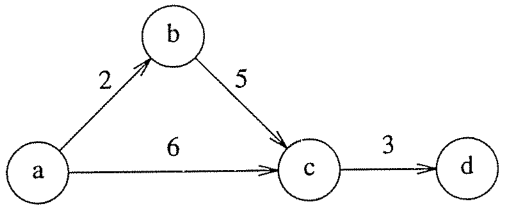


Figure 2.1. Relation R and its Underlying Graph

TC			
Src	Dest	Tot_Dist	PATH
a	b	2	{ <a, b, 2> }
b	c	5	{ <b, c, 5> }
c	d	3	{ <c, d, 3> }
a	c	6	{ <a, c, 6> }
a	c	7	{ <a, b, 2>, <b, c, 5> }
b	d	8	{ <b, c, 5>, <c, d, 3> }
a	d	9	{ <a, c, 6>, <c, d, 3> }
a	d	10	{ <a, b, 2>, <b, c, 5>, <c, d, 3> }

Figure 2.2. Closure Relation

In Section 3.5 we discuss normalized representations of the closure relation.

2.4 A Taxonomy of Path Problems

In this section we focus on the label computations specified in a generalized transitive closure query. Specifically, we study the case where the label concatentation and aggregation function, respectively *CON* and *AGG*, form an algebraic structure that allows the query to be evaluated efficiently, applying the computation of *CON* and *AGG* in interleaved fashion. We identify classes of such path problems and discuss the implications of our classification on the evaluation of these problems. We also clarify the relationship between two classes suggested previously in [CrN89] that was left open in that study.

2.4.1 Path Algebra

A **path algebra** [Car78] (also **closed semiring** [AHU75,CrN89,Meh84]) is a system A = (*D*, *BCON*, *BAGG*), where *D* is a set (domain), and *BCON* and *BAGG* are binary functions with the following algebraic properties:

1. BCON is closed and associative.
$\forall\ L_1, L_1 \in D\ BCON(L_1,L_1) \in D$
$\forall\ L_1,L_2,L_3 \in D\ BCON(L_1,BCON(L_2,L_3)) = BCON(BCON(L_1,L_2),L_3)$

2. BAGG is closed and associative.
$\forall\ L_1, L_2 \in D\ BAGG(L_1,L_2) \in D$
$\forall\ L_1,L_2,L_3 \in D\ BAGG(L_1,BAGG(L_2,L_3)) = BAGG(BAGG(L_1,L_2),L_3)$

3. BAGG is commutative.
$\forall\ L_1, L_2 \in D\ BAGG(L_1,L_2) = BAGG(L_2,L_1)$

4. BCON and BAGG have identity elements. The identity of *BCON* is denoted by ε and is also called the *unit* of the algebra. The identity of *BAGG* is denoted by θ and is also called the *zero* of the algebra. θ is also an annihilator wrt. *BCON*.
$\forall\ L \in D\ BCON(L,\varepsilon) = BCON(\varepsilon,L) = L$
$\forall\ L \in D\ BAGG(L,\theta) = L$
$\forall\ x \in D\ BCON(x,\theta) = BCON(\theta,x) = \theta$

Since *BCON* is closed and associative, it can be viewed as a function over a sequence (an ordered set) of values. Similarly, since *BAGG* is closed, associative and commutative, it can be viewed as a function over a set of values. Using the notation of Section 2.1, we denote the corresponding set functions by *CON* and *AGG*. We complete the definition of *CON* and *AGG* by
$CON(\overline{\varnothing}) = AGG(\varnothing) = \theta$
$\forall\ x \in D\ CON([x]) = AGG(\{x\}) = x$
where $[x]$ and $\{x\}$ are the list and set constructors, and $\overline{\varnothing}$ and $\varnothing$ are the empty list and empty set, respectively. From here on, unless the distinction is important, we use set notation for both *CON* and *AGG*.

If *P* is a path whose arcs have labels $L_1, \ldots, L_n$, we may write $CON(P)$ as shorthand for $CON(L_1, \ldots, L_n)$. Similarly, if ψ is a path-set whose paths have labels $L_1, \ldots, L_m$, we may write $AGG(\psi)$ as shorthand for $AGG(L_1, \ldots, L_m)$.

5. BCON distributes over BAGG.
$\forall\ L_1,L_2,L_3 \in D\ BAGG(BCON(L_1,L_2),BCON(L_1,L_3)) = BCON(L_1,BAGG(L_2,L_3))$

The motivation for Condition 5 is efficiency. Consider the situation in Figure 2.3.

We would like to compute a path-set label for this graph. According to our model, we need to first concatenate arcs *a* and *b* and *a* and *c*, and then aggregate the resulting paths *ab* and *ac*. It is better to first merge the paths *b* and *c* and

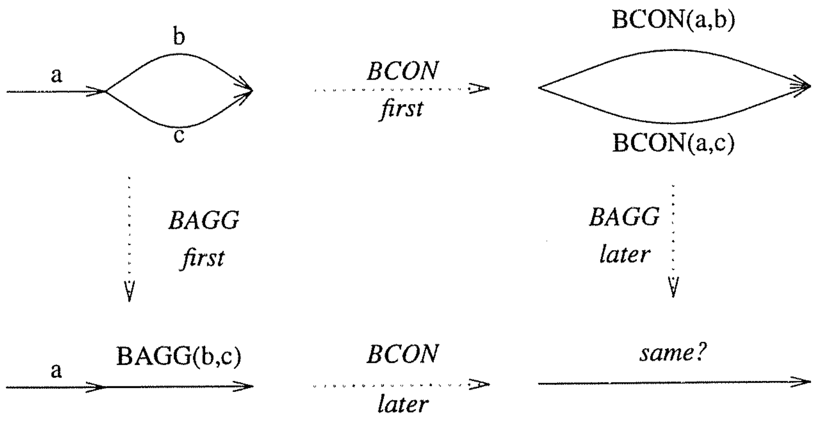


Figure 2.3. Distribution of *BCON* over *BAGG*

then concatenate a with the resulting path-set. In this example the difference is 2 concatenation and aggregation operations instead of 3, but in general the difference is between a polynomial and an exponential number of operations. The distribution of *BCON* over *BAGG* allows us to "move the aggregation inside" and still get the correct path-set label.

Example 1 — Path Algebra

Bill of materials: the domain D is positive integers, *BCON* is multiplication, *BAGG* is addition, ε is 1, and θ is 0. Consider an assembly database represented by an acyclic graph. Each node in the graph represents a part, and an arc from x to y with label q denotes that part x directly includes a quantity q of y parts. The bill of materials for some complex part is the list of all subparts and the quantity of each subpart needed to construct the complex part.

2.4.2 Ordered Path Algebra

We say that a path algebra A is *ordered* if it satisfies the following condition:

6. *BAGG* is idempotent:
$\forall\ L \in D\ BAGG(L,L) = L$

Let us denote by $\preceq$ the relation defined by
$L_2 \preceq L_1 \iff BAGG(L_1,L_2) = L_1$
From Condition 6, relation $\preceq$ is reflexive. It is easy to see that $\preceq$ is also anti-symmetric (since *BAGG* is a function), and transitive (since *BAGG* is associative). Hence, $\preceq$ is an *ordering* [Car78]. If $L_2 \preceq L_1$ we say that L_1 is *as good as* L_2.

These notions carry over to paths in a straightforward manner. Let P_1 and P_2 be paths, and let the path label of P_1 be L_1, and the path label of P_2 be L_2. If $L_2 \preceq L_1$ we say that P_1 is *as good as* P_2. Given a set of paths $P_1 \cdots P_k$, we say that P_i is an *optimal* (best) path in the set if it is as good as $P_1 \cdots P_{i-1}, P_{i+1} \cdots P_k$.

Example 2 — Ordered Path Algebra

Longest path: the domain D is real numbers, *BCON* is addition, *BAGG* is MAX, ε is 0, and θ is $-\infty$. The ordering $\preceq$ is $\geq$.

Example 3 — Ordered Path Algebra

Listing all paths ([Car78]): Let Σ be an alphabet, let Σ^* be the set of words over Σ, and let the domain D be $\rho(\Sigma^*)$ — the set of languages over Σ^* (that is, the power set of Σ^*). *BCON* is the (Cartesian) product
$BCON(X,Y) = \{x \circ y \mid x \in X, y \in Y\}$
where $\circ$ is concatenation of words. *BAGG* is set union. ε is the language $\Lambda = \{\lambda\}$, where λ is the empty word. θ is the empty language $\emptyset$. The ordering $\preceq$ is set inclusion, $\subseteq$.

Corollary 1

In an ordered path algebra with domain D, it follows from Condition 4 that θ is the least element of D with respect to $\preceq$:
$\forall\ L \in D\ \theta \preceq L$

Corollary 2

In an ordered path algebra, it follows from Condition 5 that *BCON* is *isotone* with respect to $\preceq$, that is, if $L_2 \preceq L_1$ then $BCON(L_2,L_3) \preceq BCON(L_1,L_3)$ and $BCON(L_3,L_2) \preceq BCON(L_3,L_1)$ [Car78].

The isotone property of *BCON* guarantees that if some part of a path is not optimal then the whole path cannot be optimal. Therefore we need not consider all paths: we may disregard sub-paths already known to be suboptimal when composing longer paths. This is the well known Bellman's principle of optimality [Bel58], and it underlies many specialized algorithms for computation of ordered path problems such as Dijkstra's shortest path algorithm [Dij59].

2.4.3 Absorptive Path Algebra

We say that a path algebra A is *absorptive* if it is ordered and satisfies the following condition:

7. $\preceq$ is absorptive wrt. *BCON*.
$\forall\ L_1,L_2 \in D\ BCON(L_1,L_2) \preceq L_1$ and $BCON(L_1,L_2) \preceq L_2$

In an absorptive path problem, extending a path cannot make it better. In particular, a cyclic path will never be better than the shorter acyclic path with all cycles removed. Therefore, absorptive path problems are well defined for cyclic graphs, in the sense that they are *cycle-indifferent* [ADJ90]: the computation of such a path problem over a cyclic graph will not take into account any cyclic paths. A non-absorptive path problem is in general ill-defined for a cyclic graph, since an infinite set of paths may be considered. To prevent that, we could redefine the problem to consider only simple paths. However, in order to determine whether a path contains a cycle, we must explicitly enumerate the nodes in the path, and such enumeration requires exponential effort. As an example, consider the shortest path problem with the domain $D = [+\infty, -\infty]$.

Even a non-absorptive ordered path problem is well-defined however, if the *graph is absorptive* — every (elementary) cycle in the graph has a sub-unitary label [Car78] :
$\forall\ P\ s.t.\ P\ is\ a\ cycle\ CON(P) \preceq \varepsilon$

Consider the shortest path problem again. If the graph has negative labels, but there are no negative cycles, the problem is well-defined.

Example 4 — Absorptive Path Algebra

The domain D is natural numbers, $BCON$ is multiplication, $BAGG$ is GCD (the greatest common divisor), ε is 1 and θ is ∞. The ordering $\preceq$ is "x *divides* y". Since $x*y$ divides y and $x*y$ divides x, this is an absorptive path algebra.

Cruz and Norvell [CrN89] define an *absorptive* path algebra differently, as a path algebra (Conditions 1 - 5) where the following condition (CN) holds:
CN: $\forall\ L \in D\ BAGG(L,\varepsilon) = \varepsilon$

We now show that this definition is equivalent to our definition of an absorptive path algebra.

Claim: Conditions 1 - 7 $\Longleftrightarrow$ Conditions 1 - 5 + Condition CN

Proof

$\Rightarrow$ For every absorptive path algebra Condition CN holds:

This follows directly from Condition 7.
$\forall\ L \in D\ L = BCON(L,\varepsilon) \preceq \varepsilon$

$\Leftarrow$ Every path algebra satisfying CN is absorptive.

We show first that every path algebra satisfying CN is ordered ($BAGG$ is idempotent), and then that it is absorptive.

$\forall\ L \in D\ BAGG(L,L) = BAGG(BCON(L,\varepsilon),BCON(L,\varepsilon)) = BCON(L,BAGG(\varepsilon,\varepsilon)) = BCON(L,\varepsilon) = L$

$\forall\ L_1,L_2 \in D\ BAGG(BCON(L_1,L_2),L_1) = BAGG(BCON(L_1,L_2),BCON(L_1,\varepsilon) =$
$BCON(L_1,BAGG(L_2,\varepsilon)) = BCON(L_1,\varepsilon) = L_1$.
Therefore $L_1 \preceq BCON(L_1,L_2)$ (showing $L_2 \preceq BCON(L_1,L_2)$ is the same). ■

We can therefore give the following equivalent definition of an absorptive path algebra (definition of absorptive semiring from [CrN89] rewritten in terms of the ordering $\preceq$):

7'. ε is the greatest element of D with respect to $\preceq$
$\forall\ L \in D\ L \preceq \varepsilon$

2.4.4 Maximizing Path Algebras

We say that a path algebra A is *maximizing* if it is absorptive and satisfies the following condition [Car78, CrN89]:

8. The ordering $\preceq$ is total
$\forall\ L_1,L_2 \in D\ L_2 \preceq L_1$ or $L_1 \preceq L_2$
In a maximizing path algebra, every two labels are comparable. Aggregation over a set of paths amounts to choosing the optimal path(s) in the set.

Example 5 — Maximizing Path Algebra

Shortest Path: the domain is positive reals, *BCON* is addition *BAGG* is MIN, ε is 0, and θ is ∞.

On the other hand, Example 4 above is an absorptive path algebra which is not maximizing — the ordering $\preceq$ is partial.

Figure 2.4 summarizes our taxonomy of path algebras. Table 2.1 shows examples of path algebras and their classification according to this taxonomy [1] (most examples are due to [Car78]).

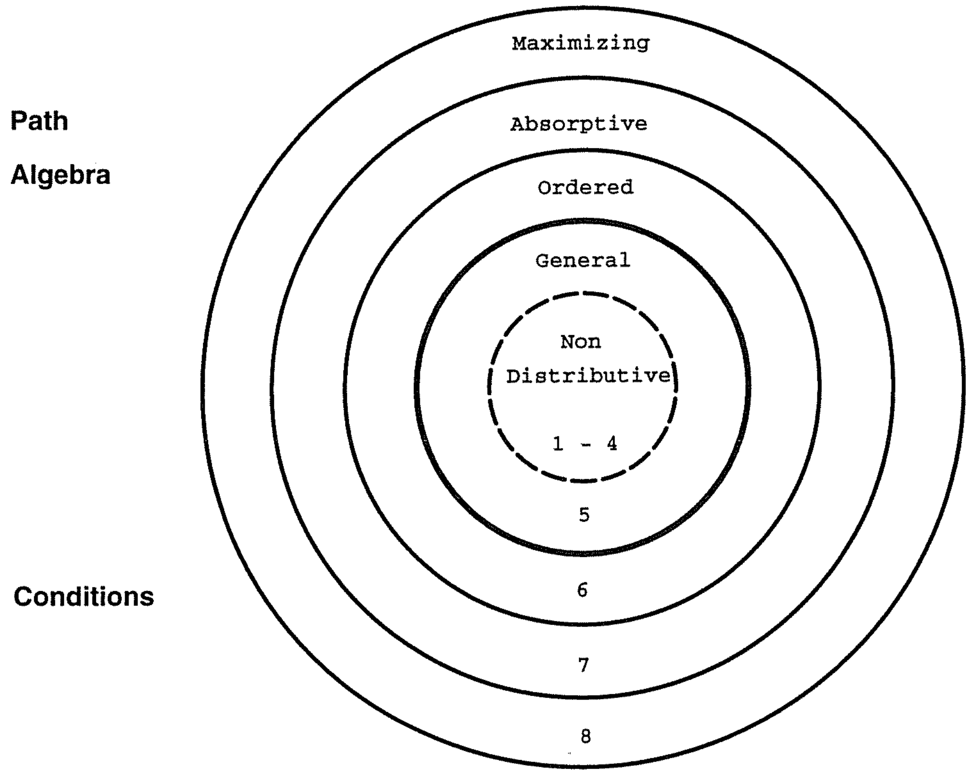


Figure 2.4. Hierarchy of Path Algebras

1. We use [0,1] to denote the set $\{ x \in R \mid 0 \leq x \leq 1 \}$. $x \times y$ denotes the Cartesian product, i.e $x \cdot y \equiv \{ \chi \circ \omega \mid \chi \in x \wedge \omega \in y \}$. $x \backslash y$ denotes that x divides y.

	Domain	*BCON*	*BAGG*	ε	θ	$\preceq$	Classification	Application
A1	$\{0,1\}$	$x \wedge y$	$x \vee y$	1	0	$x \leq y$	Maximizing	Reachability
A2	$R+ \cup \infty$	x+y	min(x,y)	0	∞	$x \geq y$	Maximizing	Shortest Path
A3	$R+ \cup \infty$	min(x,y)	max(x,y)	∞	0	$x \geq y$	Maximizing	Max. Capacity Path
A4	[0,1]	x*y	max(x,y)	1	0	$x \geq y$	Maximizing	Most Reliable Path
A5	$N \cup \infty$	x*y	GCD(x,y)	1	∞	$x \backslash y$	Absorptive	
A6	$R \cup \pm\infty$	x+y	max(x,y)	0	$-\infty$	$x \leq y$	Ordered	Longest Path
A7	$\rho(\Sigma^*)$	$x \times y$	$x \cup y$	Λ	$\varnothing$	$x \subseteq y$	Ordered	Listing all paths
A8	$I \cup \infty$	x*y	x+y	1	0		General	Bill of Materials
A9	$R+ \cup \infty$	x+y	x+y	0	0		N. Distributive	Total Effort
A10	[0,1]	x*y	1-(1-x)*(1-y)	1	0		N. Distributive	Total Reliability

TABLE 2.1. Examples of Path Algebras

2.4.5 Non-Distributive Path Algebras

So far we have considered increasingly more specialized classes of path algebras. It is interesting to explore less restrictive classes as well. In particular, we mentioned that the distributive property of a path algebra (Condition 5) allows efficient evaluation by polynomial algorithms. There are path problems of practical importance where Condition 5 is not met. We call such a problem a *non-distributive* path algebra. A9 and A10 in Table 2.1 are two examples of non-distributive path algebras. Example A9 is the paint problem [ADJ88b]. Assume that attached to each arc $E(s,t)$ is a label $L(s,t)$ representing the 'effort' required to traverse this arc. We wish to find the total effort required to traverse the graph through all possible paths. [2] The *BCON* and *BAGG* functions sum up the effort along each path and among multiple paths. Consider also Example A10 [Gua90]. Assume that attached to each arc in a graph is a label representing its reliability (the probability of its existence). We wish to compute the total reliability (probability) of the set of paths between two nodes. If we assume that the reliability of different paths is independent, than the *BAGG* function is given by *BAGG*(x,y) = 1-(1-x)*(1-y). Therefore, *BCON* does not distribute over *BAGG*.

Identifying more classes of path algebras and their properties is an interesting open research problem.

2.5 Related Work

Traversal recursion [DaS85, RHD86] was an early and influential proposal for extending databases with generalized transitive closure functionality. It supported both *path enumeration* and *path aggregation* queries. The former corresponds in our model to a query where a label aggregation operator **AGG** is not not specified, and individual paths are not merged, while the latter corresponds to a query where **AGG** is specified, and paths are merged into path-sets. Examples of path enumeration and aggregation queries are given in [RHD86], along with some suggestions on how such queries could be evaluated. However, the formulation of traversal recursion queries is quite ad hoc and complicated, making it difficult to integrate into a standard declarative relational query language,

2. As an example, suppose we wish to paint multiple trails in a zoo to assist visitors in getting to specific groups of animals: the birds trail may be painted yellow, the jungle trail green, the African trail brown etc. Roads that are common to multiple trails will be painted with multiple colors. We need to know the total amount of paint that we need to buy. Due to this example, this problem was called "the paint problem" in [ADJ88b].

such as SQL.

Agrawal [Agr87] extended relational algebra with a generalized transitive closure operator. The α operator, presented in [Agr87], provided the starting point for the algebraic query model of Section 2.2. $\alpha(R)$ computes the generalized transitive closure of R. The derivation information of each result tuple is captured in a Δ attribute. [Agr87] also introduced operators for label computation and for selection of paths.

We have extended the α operator with arc selections and path set selections. Our PATH relation is similar to Δ; however we treat PATH as an ordered relation. This allows the specification of a more general closure predicate, including constraints on consecutive arcs. These extensions considerably simplify query formulation. For instance, consider the example query in Section 2.2. In α, this query requires a join of Δ with itself, making it quite complex:

$$\alpha_{\rho:\ \mathrm{count}(\Delta \underset{Dest=Src}{\bowtie} \Delta) \,=\, \mathrm{count}(\sigma_{1.A\_Time \,<\, 2.D\_Time}(\Delta \underset{Dest=Src}{\bowtie} \Delta))}\ (Flights)$$

Kuck and Pax [KuP87] suggested the treatment of PATH as an ordered relation, and the use of *floating tuple variables* to access it. Floating tuple variables $V_1, V_2, \ldots, V_n$ declared over an ordered relation R are bound in order to n consecutive tuples from R. The timing constraint above could then be expressed as:

FORALL (PATH (V_1, V_2) V_1.A_Time < V_2.D_Time)

We also treat PATH as an ordered relation, but instead of expressing conditions on consecutive arcs using floating tuple variables, we express them through the closure predicate. We cannot use this mechanism for expressing constraints on arcs that are not consecutive, but rather k arcs away from each other, where $k > 1$. However, as we will point out in Chapter 3, constraints on consecutive arcs are incrementally computable — they can always be checked while paths are being concatenated — but constraints on nonconsecutive arcs may require access to the complete sequence of arcs in a path.

Apers [AHB86] introduced a μ calculus that allows the formulation of a transitive closure query with a predicate other than equality. A similar yet more restricted extension was proposed in [SiS88]. These proposals inspired us to allow additional constraints to be specified in the closure predicate. However, an arbitrary closure predicate, as proposed in [AHB86], will confuse the well understood semantics of transitive closure as a graph operation. Consider for example the closure condition that requires the departure time of a flight to be later than the arrival time of previous flight, without requiring the destination of the first flight to be the same as the source of the second flight:

$$\underset{A\_Time \,<\, D\_Time}{\mathrm{PATHS}}\ (Flights)$$

It is not clear what is being computed by such a query. Therefore, unlike [AHB86], and [SiS88] we require at least one equality condition in the closure predicate.

The query languages G+ [CMW86, CMW88, MeW89] and GraphLog [CoM90] support the formulation of queries using a regular expression of the labels and nodes along the path. As an example, the set of flights that have at least one connection with KLM can be found using the regular expression Path-Labels = * KLM *. These languages can express transitive closure queries including label computation. However, conditions on consecutive arcs or selections on path and path-set labels cannot be expressed directly. In addition, a major motivation for our design of a query model was that it should map naturally into a relational query language, and in particular SQL (see Chapter 2). While there are selections that can be expressed more succinctly using regular expressions, we feel that such a notation is far from the SQL spirit.

Cruz and Novrell presented an *aggregative closure* operator [CrN89] that supports the formulation of closed semiring (path algebra) queries. Selections are not discussed in [CrN89]. Cruz and Novrell also defined the classes of absorptive and maximizing semirings. However, no relationship was established between the two. In particular, it was an open question whether every absorptive path algebra is also maximizing. We have shown that the answer to this question is negative by giving a counter example (Example A5). We have also defined the class of ordered path algebras, and showed that every absorptive path algebra (by the definition of [CrN89]) is ordered. An absorptive semiring inspires an ordering in which θ is the least element and ε is the greatest element. In a maximizing path algebra, the ordering $\preceq$ is required to be total. For example, Figure 2.5 shows part of the Hasse diagram [3] for path algebra A5, with the ordering $x \preceq y \iff x$ *divides* y. For clarity, Figure 2.5 only includes numbers that are multiples of 2 and 3. As Figure 2.5 demonstrates, the order $\preceq$ is not total; for example, 2 and 3 are not in the order $\preceq$.

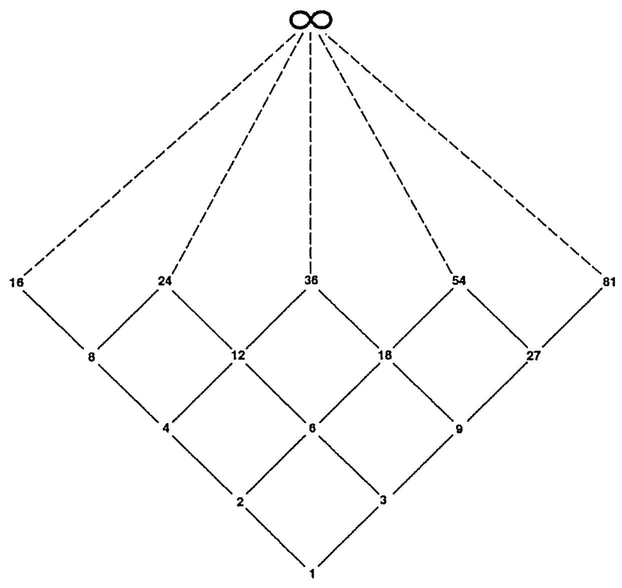


Figure 2.5. Hasse Diagram for Path Algebra A5, $x \preceq y \iff x$ *divides* y

3. In a Hasse diagram depicting an ordering of set S, each element in S is represented by a point, placed such that if $x \prec y$ then x lies below y. Lines connect elements x, y such that x *covers* y, i.e. $x \prec y$ and there is no element z such that $x \prec z \prec y$ [Car78].

2.5.1 Generalized Transitive Closure vs. General Recursion

A large body of work has been done in recent years on the development of logic-based programming languages, such as Datalog, which are capable of expressing recursive queries more general than transitive closure, including non-linear recursion [RSS92, Ull89]. In the remainder of this section, we review the use of Datalog to express generalized transitive closure queries. In subsequent chapters, we will contrast our proposed generalized transitive closure operator with Datalog in more detail.

The transitive closure operator can be expressed using the following Datalog program. We follow the usual Datalog conventions [Ull89]: PA1 and PA2 are rules, which together form a program PA. Lower case letters denote predicates, and upper case letters denote variables.

```
PA1:  path(X,Y) :- arc(X,Y)

PA2:  path(X,Y) :- path(X,Z), arc(Z,Y)
```

In program PA the (extensional, or stored) predicate arc denotes the arc relation, and the (intentional, or derived) predicate path denotes the path relation.

A query to find all nodes reachable from node x would be expressed by

```
QA: path(x,?)
```

were " ? " denotes a free variable whose possible set of values is to be filled in by the program.

If the graph is labeled, the arc and path predicates have additional arguments representing the labels. A label concatenation function may be expressed incrementally by using arithmetics to denote the binary form of the function (i.e. *BCON*). [4] For example, assume each arc has an associated cost label, and we wish to compute the (total) cost of each path. This can be done by program PB below.

```
PB1:  path(X,Y,C) :- arc(X,Y,C)

PB2:  path(X,Y,C) :- path(X,Z,C1), arc(Z,Y,C2), C = C1 + C2
```

A query to find all nodes reachable from node x by a path no longer then 1000 would be expressed by

```
QB: path(x,?,C), C <= 1000
```

To express label aggregation functions as well, we need to extend Datalog with a predefined (interpreted) set of aggregate functions such as sum and min. The shortest path problem can be expressed by program PC below. [5]

4. Assuming that Datalog is extended with a predefined set of aggregate functions such as sum and min, we still cannot express the set form of the label concatenation function (i.e. CON) directly, unless the set of composite arcs is accumulated for each path, e.g., using a list.

5. Our notation is borrowed from LDL [NaT89]. For simplicity, we assume multiset semantics, as in [MPR90]. Otherwise, the path predicate must include another argument, such as the last hop, to prevent erroneous duplicate elimination (see e.g., [RoS92]).

```
PC1:  path(X,Y,C)  :- arc(X,Y,C)

PC2:  path(X,Y,C)  :- path(X,Z,C1), arc(Z,Y,C2), C = C1 + C2

PC3:  s_path(X,Y,min(<C>)) :- path(X,Y,C)
```

The predicate `s_path` denotes the shortest path relation. The notation `min(<C>)` denotes that for a particular instantiation of (grouping) variables X and Y, all C values are collected in a set, and then the `min` aggregate function is applied to that set. Suppose X and Y are instantiated by *x* and *y*, and the minimum of the corresponding C values is *m*, then we get a new `s_path` fact <*x*,*y*,*m*>.

A query to find all nodes reachable from node *x* and the length of the shortest path to each, provided it is no longer then 1000, would be expressed by

```
QC: s_path(x,?,MIN_C), MIN_C <= 1000
```

In subsequent chapters we compare our work with the work on more general recursion along several dimensions: ease of expressing generalized transitive closure queries, optimization, and performance. In our opinion, many practical recursive queries are indeed generalized transitive closures. Therefore, the ability to recognize this restricted type of recursion and to employ specialized optimization techniques and evaluation algorithms is of importance. We believe that significant performance gains can be made in the evaluation of generalized transitive closure queries when such information is available to the query processor.

Chapter 3

Language Extensions

This Chapter presents SQL/TC, an extension of SQL to allow the expression of generalized transitive closure queries [DaA]. The extension permits the user to pose queries that compute paths between two points and information associated with these paths. Such queries may specify selections on arcs, paths or sets of paths. The answer to a query may include the start and end nodes of a path and the sequence of arcs in the path. It may also include the aggregation of information for different paths between the same endpoints. Our notation preserves the spirit of SQL, and allows a declarative and concise formulation of transitive closure queries.

We have chosen the SQL standard as defined by the ANSI Standardization Committee [Mel92] as the platform for our work since SQL is the most popular query language used in relational databases. In Sections 3.1 — 3.3 we describe the syntax and semantics of our extensions to ANSI/SQL. We discuss related work in detail in Section 3.4, and conclude in Section 3.5.

3.1 SQL Extensions

We introduce the SQL/TC syntax with an example. Consider the Flight relation and example query of Section 2.2. This query may be expressed in SQL/TC as follows:

```
1.  SELECT TC.Src, MIN( Acc_Price ) FROM
2.     ( CLOSURE Dest = NEXT Src AND A_Time < NEXT D_Time OF Flights
3.     WITH Acc_Price = SUM ( PATH.Price WHERE Airline <> 'KLM' )) AS TC
4.  WHERE TC.Dest = 'Tasmania'
5.  AND TC.Src IN ('JFK', 'LGA', 'EWR' )
6.  GROUP BY TC.Src
7.  HAVING MIN( Acc_Price ) < 2500
```

Intuitively, the parenthesis in lines 2 and 3 define the closure relation, named TC. The closure is formed by joining Flights tuples, such that the Dest value of each tuple is equal to the Src value of the matching tuple, and the A_Time value of each tuple is less than the D_Time value of the matching tuple. Line 3 specifies a new attribute for the TC relation, called Acc_Price. For a given TC tuple, the value of this attribute is the sum of the Price values of the tuples in the associated PATH relation, ignoring tuples whose Airline value is KLM. Lines 4 and 5 select from the TC relation those tuples whose Dest value is Tasmania, and whose Src value is one of the three New York airports. The MIN aggregate in line 1, together with line 6, partitions the selected TC tuples according to their Src values, and for each partition computes the minimum of the Acc_Price attribute values. Line 7 eliminates those partitions whose minimum Acc_Price value is less than 2500 dollars. Finally, the projection in line 1 projects the resulting tuples on their Src and minimum Acc_Price values.

In the rest of this section, we describe formally the SQL/TC syntax and its mapping to the query model of Chapter 2. The next section illustrates the use of SQL/TC through many examples.

In ANSI/SQL [Mel92], the basic construct used for retrieval is a "query specification". Its BNF syntax is:

```
<query specification> ::=
  SELECT [ ALL | DISTINCT ] <select list> <table expression>
```

The <select list> is a list of attributes the result should be projected on. It may also include aggregate functions. The <table expression> is the query itself:

```
<table expression> ::=
  <from clause>
  [ <where clause> ]
  .
  .
  .
```

The <from clause> specifies the relations that are the input to the query, while the <where clause> specifies the retrieval criteria. The operand of the <from clause> can be a stored relation or a derived relation. A range variable (correlation name) may be associated with the relation:

```
<from clause> ::=
  FROM <table reference> [ { , <table reference> } ... ]

<table reference> ::=
  <table name> [ [ AS ] <correlation name> [ ( <derived column list> ) ] ]
  | <derived table> [ AS ] <correlation name> [ ( <derived column list> ) ]
  | <joined table>
```

The CLOSURE construct we propose is added as a new form of a derived relation:

```
<derived table> ::=
  ( <query expression> ) 6
  | ( <closure expression> )
  .
  .
  .
```

6. For the purpose of this section, one can equate <query expression> with <query specification> presented above. In general, a query expression is a combination of UNION, INTERSECTION, and DIFFERENCE operations on some query specifications.

```
<closure expression> ::=
   <closure clause>
   [ <path label clause> ]
   [ <path selection> ]

<closure clause> ::=
   CLOSURE <closure predicate> OF <closure table>

<closure table> ::= <table reference>
```

The <table reference> specifies the relation the transitive closure is applied to. The <closure clause> corresponds to the **PATHS** operator. The <closure predicate> corresponds to the λ predicate. It is a conjunction of boolean constraints:

```
<closure predicate> ::=
   <closure condition> [ { AND <closure condition> } ... ]
```

Two types of conditions can appear in a closure predicate:

```
<closure condition> ::=
   <column expression> <comp op> NEXT <column expression>
   | <column expression> <comp op> <literal>
```

<column expression> is an expression involving exactly one attribute specification. <comp op> is a comparison operator, such as "=" or "<".

The first type of condition specifies a constraint on consecutive arcs. The keyword NEXT is used to distinguish the *second* of a pair of consecutive arcs. NEXT makes use of the order of arcs in a path, reflected in the order of tuples in the PATH relation. Let the two column expressions specified in the condition be $col\text{-}exp_1$ and $col\text{-}exp_2$ respectively. For every two successive tuples r_1 and r_2 in a PATH relation, $col\text{-}exp_1(r_1)$ <comp op> $col\text{-}exp_2(r_2)$ must hold.

The closure predicate must include at least one equality constraint on consecutive arcs, i.e., a closure condition of the form

```
<column expression> = NEXT <column expression>
```

The conjunction of constraints of this form constitute the *primary closure condition.*

The second type of condition specifies a condition on all arcs or nodes. Let the specified column expression be *col-exp* and the designated literal be *value.* For every tuple r in a PATH relation, $col\text{-}exp(r)$ <comp op> *value* must hold.

The <path label clause> corresponds to the **CON** operator. It specifies label concatenation functions and path labels:

```
<path label clause> ::=
  WITH <path label specification list>

<path label specification list> ::=
  <path label specification> [ { , <path label specification> } ... ]

<path label specification> ::=
  <path label definition> [ <arc selection> ]

<path label definition> ::=
  <column name> = <value expression>

<arc selection> ::= <where clause>

<path selection> ::= <where clause>
```

Each <path label specification> defines a new attribute in the resulting closure relation. The <arc selection> clause corresponds to the ξ arc selection predicate. It selects those tuples in the corresponding PATH relation that satisfy the specified selection. The <path label definition> clause designates a new attribute name and specifies that the value of this attribute is derived from the label values of the selected arcs by using a label concatenation function in the <value expression>. The <path selection> clause in the closure expression corresponds to the path selection predicate σ. It is used for selection of paths by applying a selection criterion on the result relation of the closure operation.

Note that we do not require a new language construct for specifying *AGG* functions, since these can be specified using existing SQL aggregate function syntax. Thus the SELECT clause provides the function of both the π and **AGG** operators. Similarly, SQL's HAVING clause is used to specify the path-set selection operator δ, as will be illustrated in the next section.

Table 3.1 summarizes the mapping between the model presented in Section 2, the SQL/TC constructs, and the syntax of these constructs. A full definition of the portion of the SQL syntax relevant to our extensions is given in Appendix 1. The next section demonstrates the utility of SQL/TC.

3.2 Example Queries

We now give a number of example queries to illustrate SQL/TC. Appendix 2 contains the algebraic formulation of these queries, using the query model of Chapter 2.

The first set of queries determine reachability between nodes in a graph.

3.2.1 The Closure Predicate

Example 1: Given relation Map (Src, Dest), find all connected pairs in Map:

```
SELECT Src, Dest FROM
  ( CLOSURE Dest = NEXT Src OF Map )
```

Model	SQL/TC Construct	SQL/TC Syntax
R	<closure table>	OF R
$PATHS$ λ	<closure predicate>	CLOSURE λ
CON $_{X,\xi}$	<path label clause>	WITH $PL = CON(X)$ WHERE ξ
σ	<path selection>	WHERE σ
AGG $_{Y,Z}$	<select list> + <group by clause>	$AGG(Y)$ ... GROUP BY Z
δ	<having clause>	HAVING δ
π_A	<select list>	SELECT A

TABLE 3.1. Mapping Between Model and SQL/TC

The CLOSURE clause applies the generalized transitive closure operator to Map, based on the Src and Dest attributes. The unqualified Src and Dest attribute names in the select list reference the only relation in the FROM clause, which is the generalized transitive closure result.

Alternatively, using full qualification, we could have written this query as:

```
SELECT TC.Src, TC.Dest FROM
  ( CLOSURE Dest = NEXT Src OF Map ) AS TC
```

In addition to specifying the primary closure condition, the closure predicate is used for specifying conditions on all arcs or nodes, and conditions on consecutive arcs. These are illustrated in the following two queries.

3.2.1.1 Selection on All Arcs/Nodes

Example 2: Given relation Map (Src, Dest), find all cities connected through paths that do not go through Chicago.

```
SELECT Src, Dest FROM
  ( CLOSURE Dest = NEXT Src AND Dest <> 'Chicago' OF Map )
```

3.2.1.2 Selection on Consecutive Arcs

Example 3: Given an airline relation Flights (Src, Dest, D_Time, A_Time), find all cities connected by flights such that the destination of each flight is the source of the next flight, and the arrival time of each flight is at least one hour before the departure time of the next flight [AhU79].

```
SELECT Src, Dest, D_Time, A_Time FROM
  ( CLOSURE Dest = NEXT Src AND A_Time ≤ NEXT D_Time – 1 OF Flights )
```

3.2.1.3 Selection on End Points

Example 4: Given relation Map (Src, Dest), find all points in Map reachable from New York.

```
SELECT Src, Dest FROM
  ( CLOSURE Dest = NEXT Src OF Map ) AS TC
WHERE TC.Src = 'New York'
```

We emphasize that we do not expect the execution of this query to follow the steps implied in the query expression, that is, first compute the whole closure and then select out the paths that start at New York. Yet, we believe the user should be free to formulate queries in the most convenient and natural way, and it should be the responsibility of the optimizer to choose an efficient execution order. Note that all the previous queries could have endpoint-selection specified as well.

Recall that in SQL/TC the set of arcs that constitutes a path is stored in a PATH relation associated with each closure tuple. The PATH relation can be referenced within the closure expression in order to specify path label computations or to specify path-selections on the set of arcs in the path. The following subsections illustrate the use of the PATH relation.

3.2.2 Path Labels

A path label is computed by applying a *CON* aggregate function to the ordered set of arcs in each path. The *CON* function is specified using a path label (WITH) clause.

Example 5: Given relation Trains (Src, Dest, Price), find all cities reachable from Paris by trains and the total cost of each trip.

```
SELECT Dest, Cost FROM
  ( CLOSURE Dest = NEXT Src OF Trains
    WITH Cost = SUM ( PATH.Price ) ) AS TC
WHERE TC.Src = 'Paris'
```

In the path label clause, we define the Cost value of a TC tuple as the SUM of the Price values of the tuples in the corresponding PATH relation.

3.2.2.1 Arc Selection

We can associate an arc specification predicate with each path label. The specified aggregate function will only be applied to the (sub)set of arcs that satisfy the arc selection.

Example 6: Given relation Trains (Src, Dest, Dist, Kind), find all cities reachable from Paris by train. Calculate separately the distances traveled by express trains and regular trains.

```
SELECT Dest, E_Dist, R_Dist FROM
  ( CLOSURE Dest = NEXT Src OF Trains
    WITH E_Dist = SUM ( PATH.Dist ) WHERE Kind = 'Express',
         R_Dist = SUM ( PATH.Dist ) WHERE Kind = 'Regular' ) AS TC
WHERE TC.Src = 'Paris'
```

Note that two path labels are generated for each path.

3.2.3 Path-Set Labels

A path-set label is computed by applying an *AGG* aggregate function to the set of paths that satisfy the specified path selection. The *AGG* function is specified using regular SQL aggregate function syntax.

Example 7 (Shortest Distance, Maximum Capacity): Given Roads (Src, Dest, Dist, Cap), find the shortest distance and maximum capacity of paths from *a* to *b*.

```
SELECT MIN ( TC.Tot_Dist ), MAX ( TC.Min_Cap ) FROM
  ( CLOSURE Dest = NEXT Src OF Roads
  WITH Tot_Dist = SUM ( PATH.Dist ),
     Min_Cap = MIN ( PATH.Cap ) ) AS TC
WHERE TC.Src = 'a' AND TC.Dest = 'b'
```

In the WITH clause, we define the Tot_Dist value of a TC tuple as the SUM of the Dist values, and the Min_Cap value of a TC tuple as the MIN of the Cap values, of the tuples in the corresponding PATH relation. In the SELECT list we compute the MIN of the Tot_Dist values and the MAX of the Min_Cap values of the TC tuples that have Src *a* and Dest *b*.

3.2.3.1 Path-Set Partitioning

The set of paths that satisfy the specified path selection can be partitioned using a group-by clause.[7]

Example 8 (Bill of Materials): Given Assembly (Part, Subpart, Qty), find all subparts used to construct part *a*, and the quantity required of each subpart (assume that an aggregate function PRODUCT exists).[8]

7. GROUP BY is the aggregate formation operator in SQL. It partitions a relation into subrelations having identical values for the given expression. It then applies any aggregate function specified to the tuples of each partition.

8. Alternatively, we can devise syntax, similar to the syntax for the closure condition, for defining a *CON* function *incrementally*, e.g.

```
SELECT Subpart, SUM ( Sub_Qty ) FROM
  ( CLOSURE Subpart = NEXT Part OF Assembly
  WITH Sub_Qty = PRODUCT ( PATH.Qty ) ) AS TC
WHERE TC.Part = 'a'
GROUP BY TC.Subpart
```

3.2.3.2 Path-Set Selection

We can select a subset of the path-sets after partitioning using a having clause.[9]

Example 9 (Most Reliable Path): Given Circuit (Src, Dest, Reliability), find all points reachable from *a* and the most reliable connection to each point, where that reliability is greater than 90%.

```
SELECT Dest, MAX ( Acc_Rel ) FROM
  ( CLOSURE Dest = NEXT Src OF Circuit
  WITH Acc_Rel = PRODUCT ( PATH.Reliability )) AS TC
WHERE TC.Src = 'a'
GROUP BY TC.Dest
HAVING MAX(Acc_Rel) > 0.9
```

3.2.4 Using the Path Information in the Query

A path label can also be introduced in order to specify some path selection:

Example 10: Given relation Flights (Src, Dest, Airline), find all flights from Paris to Vancouver where at least one connection is by KLM:

```
SELECT TC.PATH FROM
  ( CLOSURE Dest = NEXT Src OF Flights
  WITH KLM_connections = COUNT( PATH ) WHERE PATH.Airline = 'KLM' )
  AS TC
WHERE TC.Src = 'Paris' AND TC.Dest = 'Vancouver' AND KLM_connections > 0
```

The path label counts the number of KLM connections, and the associated path selection guarantees that the count is positive. The inclusion of TC.PATH in the select list results in the retrieval of the complete path information, as we explain in Section 3.3.

This query can be formulated more conveniently using a subquery on the PATH relation:

9. HAVING is used to specify selection of partitions in SQL, much like WHERE is used to specify selection of tuples.

```
SELECT TC.PATH FROM
  ( CLOSURE Dest = NEXT Src OF Flights
  WHERE EXISTS ( SELECT * FROM PATH WHERE PATH.Airline = 'KLM' ) )
  AS TC
WHERE TC.Src = 'Paris' AND TC.Dest = 'Vancouver'
```

Example 11 (similar to Example 10): Given relation Flights (Src, Dest, Airline), find all flights from Paris to Vancouver that go through Toronto:

```
SELECT TC.PATH FROM
  ( CLOSURE Dest = NEXT Src OF Flights
  WHERE EXISTS ( SELECT * FROM PATH WHERE PATH.Dest = 'Toronto' ) )
  AS TC
WHERE TC.Src = 'Paris' AND TC.Dest = 'Vancouver'
```

3.2.5 Interaction with Other Relational Operators

The transitive closure operator may be mixed with other relational operators, as illustrated by the following two examples.

3.2.5.1 Closure of a Derived Relation

Example 12: Given relations Flights (Src, Dest) and Trains (Src, Dest, Kind), find all cities reachable from Paris by some sequence of flight and express train connections.

```
SELECT Dest FROM
  ( CLOSURE Dest = NEXT Src OF
    ( SELECT Src, Dest FROM Flights ) UNION
    ( SELECT Src, Dest FROM Trains WHERE Kind = 'Express') ) AS TC
WHERE TC.Src = 'Paris'
```

Note that the closure is applied to the result of a subquery, rather than to a base relation.

3.2.5.2 Join of the PATH Relation

Example 13: Given Flights (Src, Dest, Airline) and City (Name, Country), find all flights from Paris to Vancouver with at least two stops in the USA.

```
SELECT TC.PATH FROM
  ( CLOSURE Dest = NEXT Src OF Flights
  WHERE ( SELECT COUNT (*) FROM PATH, City
      WHERE PATH.Dest = City.Name AND City.Country = 'USA') >= 2 ) AS TC
WHERE TC.Src = 'Paris' AND TC.Dest = 'Vancouver'
```

In this query, we select a path based on attributes of nodes in the path. This requires a join between the PATH relation and the relation holding these node attributes.

3.3 Retrieving the Result

Often the user is interested not only in the fact that two points are connected, but in the actual path (or paths) that leads from the first to the second. For example, consider a shortest path query between points *a* and *b*. An answer that does not specify the whole sequence of arcs in the shortest path(s) from *a* to *b* may well be insufficient. To get the actual path we need to retrieve tuples from the PATH relation associated with a TC tuple.

When retrieving PATH as well as other attributes, one alternative is to make the nested structure of the TC relation explicit in the answer by having a PATH attribute which is itself a relation, as in [RKB87]. For example, consider the relation Assembly and its underlying graph, given in Figure 3.1

Assembly		
Part	Subpart	Qty
a	b	3
a	d	7
b	c	2
c	d	5
e	b	4

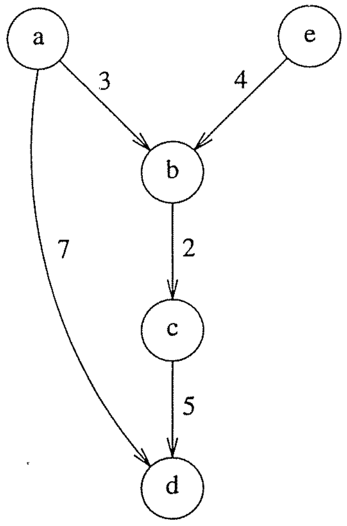


Figure 3.1. Relation Assembly and its Underlying Graph

The following SQL/TC query selects all subparts needed to build part *a* and the quantities required of each subpart, and retrieves all attributes including PATH.

```
SELECT Part, Subpart, Sub_Qty, PATH FROM
  ( CLOSURE Subpart = NEXT Part OF Assembly )
  WITH Sub_Qty = PRODUCT ( PATH.Qty ) AS TC
WHERE TC.Part = 'a'
```

This query will result in the answer displayed in Figure 3.2.

If the underlying system does not support non-normal form relations, the result can be ''unnested''; the PATH information is then stored in a separate relation connected to the TC relation through a system generated identifier, as shown in Figure 3.3.

TC					
Part	Subpart	Sub Qty	PATH		
			Part	Subpart	Qty
a	b	3	a	b	3
a	c	6	a	b	3
			b	c	2
a	d	7	a	d	7
a	d	30	a	b	3
			b	c	2
			c	d	5

Figure 3.2. Displaying Path Information Using a Nested Relation

TC			
Part	Subpart	Sub Qty	Path Id
a	b	3	p1
a	c	6	p2
a	d	7	p3
a	d	30	p4

PATH			
Path Id	Part	Subpart	Qty
p1	a	b	3
p2	a	b	3
p2	b	c	2
p3	a	d	7
p4	a	b	3
p4	b	c	2
p4	c	d	5

Figure 3.3. Displaying Path Information in a Separate Relation

Another alternative is to record just a ''last hop'' value for each path, since the rest of the path is redundantly stored in the PATH table. In this case the storage overhead is much lower, but significant work must be done to recursively reconstruct the complete PATH information.

3.4 Related Work

First, we compare our proposal with the proposal under consideration in the ANSI/SQL standards committee, and then discuss other related work.

3.4.1 Comparison with Proposal to ANSI/SQL

[Eng87, Sha88, Sha87, Sul87] is a series of proposals to the ANSI/SQL standards committee. The proposed SQL extension can be viewed as embedding Datalog-style recursive rules in SQL. Appendix 3 shows the formulation of queries 1-13 in the ANSI/SQL proposed syntax. In this section, we illustrate the differences between the ANSI/SQL proposal and SQL/TC using some queries on the relation Trains (Src, Dest, Dist, Kind), given in Table 3.2. The comparison is limited to generalized transitive closure queries; however the reader should keep in mind that the ANSI/SQL proposal supports more powerful recursive queries, including non-linear recursion.
The SQL/TC formulation is more compact than the ANSI/SQL formulation. We also note the following deficiencies in the ANSI/SQL proposal:

1. The query is specified by an initial ''subquery'' and a recursive ''subquery'', where the latter is executed iteratively until no new tuples are derived (Query I in Table 3.2). The user must specify arbitrarily the

SQL/TC	ANSI/SQL Proposal
(Query I) All Connections	

SELECT Src, Dest FROM
(CLOSURE Dest = NEXT Src OF Trains)
	SELECT Src, Dest FROM
	SELECT Trains.Src, Trains.Dest FROM Trains
	RECURSIVE UNION TC
	SELECT TC.Src, Trains.Dest FROM TC, Trains
	WHERE TC.Dest = Trains.Src

(Query II) Cities Reachable From Paris	

SELECT Dest FROM
(CLOSURE Dest = NEXT Src OF Trains) AS TC
WHERE TC.Src = 'Paris'
	SELECT Dest FROM
	SELECT Trains.Src, Trains.Dest FROM Trains
	WHERE Trains.Src = 'Paris'
	RECURSIVE UNION TC
	SELECT TC.Src, Trains.Dest FROM TC, Trains
	WHERE TC.Dest = Trains.Src

(Query III) Paths From Paris To Moscow	

SELECT * FROM
(CLOSURE Dest = NEXT Src OF Trains) AS TC
WHERE TC.Src = 'Paris'
AND TC.Dest = 'Moscow'
	SELECT * FROM
	SELECT Trains.Src, Trains.Dest FROM Trains
	WHERE Trains.Src = 'Paris'
	RECURSIVE UNION TC
	SELECT TC.Src, Trains.Dest FROM TC, Trains
	WHERE TC.Dest = Trains.Src
	WHERE TC.Dest = 'Moscow'

(Query IV) All Paths Over 1000 Miles	

SELECT Src, Dest, Tot_Dist FROM
(CLOSURE Dest = NEXT Src OF Trains
 	
WITH Tot_Dist = SUM(PATH.Dist)) AS TC
WHERE Tot_Dist > 1000
	SELECT Src, Dest, Tot_Dist FROM
	SELECT Trains.Src, Trains.Dest, Trains.Dist
	FROM Trains
	RECURSIVE UNION TC(Src, Dest, Tot_Dist)
	SELECT TC.Src, Trains.Dest,
	Trains.Dist + TC.Dist FROM TC, Trains
	WHERE TC.Dest = Trains.Src
	WHERE TC.Tot_Dist > 1000

(Query V) Express Train Connections	

SELECT Src, Dest FROM
(CLOSURE Dest = NEXT Src
AND Kind = 'Express')) AS TC
	SELECT Src, Dest FROM
	SELECT Src, Dest FROM Trains
	WHERE Kind = 'Express'
	RECURSIVE UNION TC
	SELECT TC.Src, Trains.Dest FROM TC, Trains
	WHERE TC.Dest = Trains.Src
	AND Trains.Kind = 'Express'

TABLE 3.2. Comparison of SQL/TC and ANSI/SQL

direction of the closure computation. Selection on one end point can be specified in the initial subquery (Query II), but selection on both endpoints (Query III) or on a path label (Query IV) must be specified on the outer WHERE clause. In contrast SQL/TC has a symmetric closure predicate and all path selections are specified in one place, the path selection clause.

2. In the recursive subquery, the label concatenation function *CON* is specified incrementally, using "+", instead of using an aggregate function (Query IV). While this works for SUM, it may be difficult or impossible to do

so for other aggregate functions. Computation of the MINIMUM or MAXIMUM of the labels, such as the *Min_Cap* path label in Example 8, would require using an *IF* ··· *ELSE* ··· in the select list, or introduction of binary min and max functions. Computation of the AVERAGE of the labels would require the introduction of a "LEVEL" or "COUNT" attribute to count the number of arcs in the path, and a formula like:

SELECT ... , (TC.AVG * TC.LEVEL + R.LABEL) / (TC.LEVEL + 1)

In our opinion we cannot expect the user to derive and use such formulae. Further, consider what the user would have to do had he or she wanted to compute the VARIANCE or the MEDIAN of the labels.

3. Arc selection cannot be expressed in the ANSI/SQL proposal, because it amounts to performing different computations in different iterations of the recursive clause. Thus a query such as Example 6 cannot be expressed directly.[10]

4. Since arc selection cannot be expressed, and a subquery on the PATH relation cannot be used, queries on the path information using the EXIST or COUNT operators, such as Examples 10, 11 and 13, cannot be expressed directly.[10]

5. Selections on all arcs or nodes, as in Example 2, must be repeated in the initial and recursive subqueries of a closure query (Query V). This is because the input to the closure operation is referenced in both subqueries.

6. In the ANSI/SQL proposal it is not possible to retrieve the complete path (Query III).

Another extension of SQL that supports recursive queries is SBSQL, developed in the Starburst project [MFP90]. The formulation of transitive closure queries in SBSQL is similar to the ANSI/SQL proposal, and in our opinion suffers from similar deficiencies.

3.4.1.1 Termination

In [Sha87], cycle detection to avoid infinite recursion is an option that can be ''turned on'' by the user. If this option is turned off, then given a relation representing a cyclic graph, it is possible for a user to specify an ill-formed query whose evaluation results in infinite recursion.

To remedy this problem, [Sha87] proposes a LIMIT clause to limit the number of tuples in the result. We feel that this parameter is arbitrary and difficult to set intelligently. It will be better to limit the depth of the recursion, that is, the maximum length of any path. Indeed, [Eng87] proposes a current-depth (''current-level'') register for that purpose. Our approach does not require the introduction of any new constructs. It takes advantage of the fact that PATH is a relation, and its cardinality is the length of the corresponding path. Thus, to stop the recursion at depth D we can limit the maximum cardinality of any PATH to *D*.

10. It might be possible to express such a query using an IF ... ELSE in the select list, but this is an awkward and confusing mechanism.

Example 14: Given relation Map (Src, Dest), find all paths from *a* to *b* that take at most *D* hops.

```
SELECT PATH FROM
  ( CLOSURE Dest = NEXT Src OF Map
    WHERE ( SELECT COUNT (PATH.Dest) FROM PATH ) <= D ) AS TC
  WHERE TC.Src = 'a' AND TC.Dest = 'b'
```

The limiting condition could of course be more general, e.g.,

Example 15: Given relation Map (Src, Dest, Dist), find all paths from *a* to *b* that are no longer than LIMIT.

```
SELECT PATH FROM
  ( CLOSURE Dest = NEXT Src OF Map
    WHERE ( SELECT SUM (PATH.Dist) FROM PATH ) <= LIMIT ) AS TC
  WHERE TC.Src = 'a' AND TC.Dest = 'b'
```

If the purpose of the LIMIT clause was to prevent the query from taking ''too long'', then a limit on the execution time, rather than the size of the result, seems more appropriate. Such a TIMEOUT mechanism, however, should be available for any query and not limited just to CLOSURE queries.

3.4.1.2 Search Order

In [Eng87] and [Sha88], a SEARCH clause is proposed that allows the user to specify the search order as either depth-first, breadth-first or user-defined. The presumed purpose of the search order specification is to allow the user to extract structural information about the underlying graph. This is done by introducing generated ''structural'' attributes — 'TREE', 'NODE' and 'LEVEL' in [Eng87], and 'PARENT' and 'NODE' in [Sha88].

The 'LEVEL' attribute specifies the distance, in number of arcs, between the given node and the root node for this query. Its value is independent of the search order. The 'TREE', 'PARENT' and 'NODE' attributes assign numbers to nodes according to the search strategy indicated in the search clause. For example, a node can be identified as the 4th node in the 2nd tree in [Eng87], or as node 12 with parent node 5 in [Sha88].

So-called ''structural'' information, by its nature, should presumably be independent of the order in which the underlying graph was traversed. But the above ''structural'' attributes have different values when different search orders are used. Therefore, we doubt the value of the 'TREE', 'PARENT', and 'NODE' attributes. In contrast, the structural information in the PATH relation is independent of the traversal order of the underlying graph.

We believe that the choice of search order should be made by the implementation. [Sha88] suggests using a search-clause followed by an order-clause (sort) to allow the closure result to be presented in an ''indented'' way, e.g., print a type hierarchy in breadth-first order. Yet, it seems that a graphical interface such as G+ [CMW88] would be more effective for that purpose. In our extension, the user can specify an ORDER BY clause to get result tuples sorted by the length of their corresponding path, for example.

3.4.2 Other Work

Due to the PATH attribute, the result of a closure query in our model is no longer in the first normal form. The generalization of the relational model to non-first normal form attributes has received wide attention (see, for example, [ScS86]). Extensions of SQL for a nested relation model have been presented, among others, in [PiA86, RKB87]. While it is true that in an SQL extended with non-first normal form relations, the implementation of PATH would be easier, this is not a necessary requirement. We identify the following differences between our model and the nested relation model:

- We do not need nesting to an arbitrary level. The PATH relation represents a nesting level of one.
- The PATH relation is ordered, while relations in the nested relation model are not ordered.
- The nested relation model requires NEST and UNNEST operations to restructure nested relations. These operations are not needed in SQL/TC because tuples in the PATH relation can be explicitly accessed only for the purpose of retrieval, and this can be accomplished by including the PATH attribute in the selection list. Furthermore, since paths share arcs, unnesting PATH would create duplicates, and the semantics of handling these duplicates would be difficult to define.

The formulation of recursive queries in an NF^2 SQL has been studied in [Lin87]. The proposed extension can express selection on end nodes, label concatenation, and retrieval of path information. However, we find the query formulation quite complicated. One reason is that type constructors for tuples, sets, and lists must be specified in the query, as in [PiA86]. Another reason is that the initial clause, iteration clause, selection on the start nodes and termination condition are specified separately. In the proposal of [Lin87] the user must specify termination conditions explicitly, for example, by including in the query a check for acyclicity.

In [LZH90], the relational model is extended with a *path structure*, similar to our PATH relation. To express a generalized transitive closure query, a *path view*, similar to our transitive closure relation, must first be defined, using the CREATE PATH-VIEW statement. As in [Eng87, MFP90, Sha88, Sha87, Sul87], this statement comprises an initial subquery and a recursive subquery. Queries with specific selection criteria can then be formulated against the path view. To create a path view, [LZH90] introduce several new keywords (PATH-VIEW, PATH-STRUCTURE, EDGE, PATH) and built-in functions (CREATE-PATH(edge), APPEND(path, edge), CONCATENATE(path, path)). Equality (''='') is used in the WHERE clause to perform *assignments*, which is a significant deviation from normal (side-effect free) SQL semantics. Queries against the path view utilize the special *cursor functions* FIRST, LAST, SOME and ALL-IN-BETWEEN to access the embedded path structure. Compared to this proposal, SQL/TC is a simpler extension to SQL and allows queries to be expressed more compactly.

In [HSS87], an extended projection operator was introduced. Its functionality is similar to transitive closure, but limited to traversal of a disjoint hierarchical structure with instantiated roots (selection on start nodes). A transitive closure operator was also needed to traverse a shared (overlapping) hierarchy of objects. A QUEL extension utilizing both operators was presented. A later paper, [SHS87], generalized the transitive closure to an *Alpha* operator based on α [Agr87]. An SQL extension was illustrated by queries on different types of complex objects. However, neither the selections (ρ operator) nor the label computation capabilities (μ list) of α are used. Thus, the expressive power of the operator is very limited. Also, since the extended projection and the transitive closure operators overlap in functionality, queries involving both are confusing, as it is not clear what part of the recursion is handled by each. Formulation of transitive closure queries is also possible in MQL, an extension of SQL to support the Molecule-Atom data model (MAD) [HMM87, Mit89]. The closure relationship is embedded in the data model, using the concept of association. Thus a recursive query looks like a regular query on a previously defined recursive structure. Conceivably, label computation could be added to the model as part of the definition of a recursive structure.

3.4.3 Generalized Transitive Closure vs. General Recursion

The ANSI/SQL proposal [Mel92] and SBSQL [MFP90] are capable of expressing recursive queries that cannot be expressed in SQL/TC. However, the formulation of generalized transitive closure queries is more natural and direct in SQL/TC than it is in those proposals. In addition, because our proposal uses syntax that is tailored for generalized transitive closure, identifying the closure predicate, label computations, and selections specified in an SQL/TC query is straightforward. This allows the query optimizer to employ specialized optimizations and evaluation algorithms, as discussed in the following chapters. An alternative approach is to provide general recursion in the language and to attempt to recognize when a recursive query is actually a transitive closure of some kind. This approach was explored, among others, in [MuP91, NRS89, Sar89, ZYT90].

3.5 Summary

We have extended SQL to allow the expression of recursive queries based on the generalized transitive closure paradigm. The extension, SQL/TC, permits the user to pose queries that compute paths between two nodes and information associated with these paths. The queries may involve selections on arcs, paths and sets of paths. The answer to a query may include the start and end nodes of a path and the sequence of arcs in the path. It may also include the aggregation of information for different paths between the same endpoints. Our notation preserves the spirit of SQL, and allows a declarative and concise formulation of transitive closure queries.

Chapter 4

Query Optimization

In this chapter we develop a framework for optimization of generalized transitive closure queries [DAJ91]. The framework demonstrates, in an algorithm -independent fashion, how the various selections and label computations presented in Chapter 2 may be efficiently evaluated. The chapter is organized as follows. In Section 4.1, we state the optimization problem: reducing the time and space requirement of evaluating a generalized transitive closure query. We give some notation in Section 4.2. In Section 4.3 we suggest techniques for ''pushing'' path selections. In Section 4.4 we discuss the efficient evaluation of the label computation functions. We also show how arc selections and path-set selections can be optimized. We discuss related work in Section 4.5, and give our conclusions in Section 4.6.

4.1 Optimization Problem

A straightforward evaluation of a query using the model of Section 2.2 can take exponential time, since **PATHS** can take exponential time to enumerate even all simple paths. It is especially expensive considering that the user is usually only interested in a small subset of those paths.

We are interested in whether it is possible to optimize the evaluation of the selection criteria and label computations so that:

1. Paths that are not in the result, and not needed to compute the result, are pruned as early as possible.

2. Paths that are in the result, or are needed for the computation of the result, are represented in as compact a way as possible.

In Section 4.3 we discuss the early evaluation of selection criteria. In Section 4.4 we discusses query evaluation with condensed representations.

4.2 Notation:

A *concatenation* of two non-empty paths $P(s,t) = \{E_1, \cdots, E_n\}$, $P(t,w) = \{E'_1, \cdots, E'_m\}$, is the path $P(s,w) = \{E_1, \cdots, E_n, E'_1, \cdots, E'_m\}$ (see Figure 4.1). We use the notation $P(s,w) = P(s,t) \circ P(t,w)$.

An *extension* of a path $P(s,t)$ is a path $P(s,w)$ such that there exists a non-empty path $P(t,w)$, and $P(s,w) = P(s,t) \circ P(t,w)$ (see Figure 4.2). We denote an extension of a path P by $P^{\rightarrow}$.

A *superpath* of a path $P(s,t)$ is a path $P(w,z)$ such that there exist paths $P(w,s)$, $P(t,z)$, and $P(w,z) = P(w,s) \circ P(s,t) \circ P(t,z)$, and at most one of $P(w,s)$, $P(t,z)$ is empty (see Figure 4.3). We denote a superpath of a path P by P^{+}.

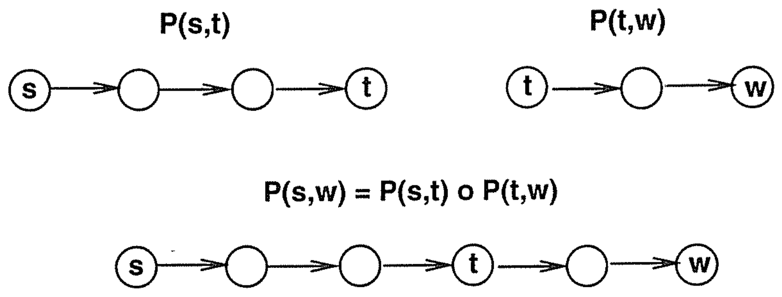


Figure 4.1. Concatenation

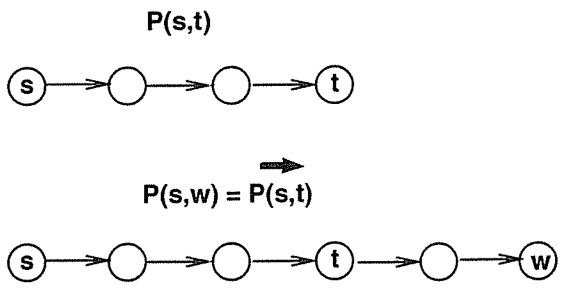


Figure 4.2. Extension

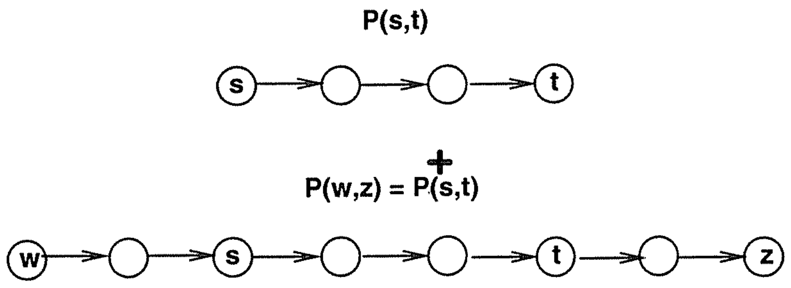


Figure 4.3. Superpath

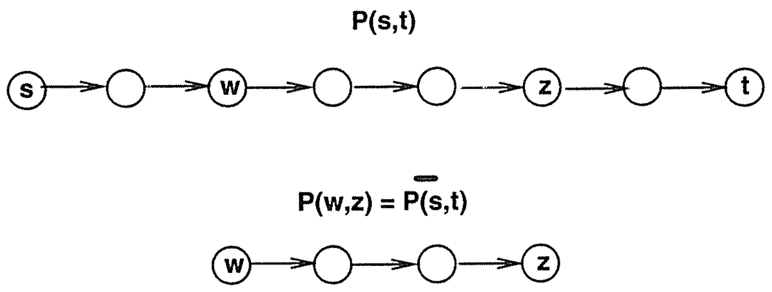


Figure 4.4. Subpath

A *subpath* of a path $P(s,t)$ is a path $P(w,z)$ such that there exist paths $P_1(s,w)$, $P_2(z,t)$, and $P(s,t) = P_1(s,w) \circ P(w,z) \circ P_2(z,t)$, and at most one of $P_1(s,w)$, $P_2(z,t)$ is empty (see Figure 4.4). We denote a subpath of a path P by P^-.

Note that path extension is defined relative to the end of the path, as implied by the arrow, while the definitions of superpath and subpath are symmetric with respect to both endpoints of the path.

Given a path P and a selection criterion C, we say that P satisfies C if evaluating C on P yields TRUE. We say that P violates C if evaluating C on P yields FALSE. We employ the notation $C(P)$ to denote the evaluation of C on P.

4.3 Evaluating Path Selections (σ)

We start by considering path enumeration queries, where label aggregation is not specified. We first state a straightforward algorithm for computing a generalized transitive closure that is derived directly from the query model described in Chapter 2. Given some selection criteria for the desired paths, the algorithm first computes all paths in the closure, and then applies the specified constraints to select only the desired paths. Later, this algorithm is modified to better exploit the selection criteria present in the query.

4.3.1 Algorithm 0

Input: A generalized transitive closure query Q over graph G, with selection criteria C on the desired paths.

Output: The set of paths in G that satisfy C.

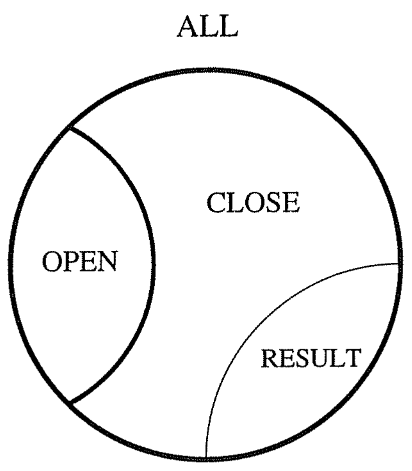


Figure 4.5. Sets of Paths

The algorithm makes use of four sets of paths: *ALL*, *OPEN*, *CLOSE* and *RESULT*. *ALL* holds all known paths at some point in the computation. *OPEN*, *CLOSE* and *RESULT* are subsets of *ALL*. *OPEN* holds paths that might be further extended. *CLOSE* holds paths that should not be extended. *RESULT* accumulates the answer set of the query. It is a subset of *CLOSE*. The algorithm terminates when *OPEN* becomes empty, at which point *RESULT* holds the answer set of the query. The logical relationship between these sets at any point in the computation is depicted in Figure 4.5.

```
1. Initialization:
   $ALL = OPEN = \{ E \mid E \textit{ is an } \text{arc} \textit{ in } G \}$.
   $CLOSE = RESULT = \varnothing$.

2. Do steps 3 through 5, while $OPEN \neq \varnothing$

3. Choose a path $P \in OPEN$. Extend this path by concatenating to it some path $P_{extend} \in ALL$.
   Let the resulting path be $P_{new} = P \circ P_{extend}$.
   Compute the path labels of $P_{new}$ by applying the specified $CON$ functions.

4. If $P_{new} \notin ALL$ then
      $ALL = ALL + P_{new}$, $OPEN = OPEN + P_{new}$

5. If $P$ has been extended in all possible ways then
      $OPEN = OPEN - P$ , $CLOSE = CLOSE + P$
      Apply $C$ to $P$.
      If $C(P) = TRUE$ then $RESULT = RESULT + P$
```

Note that algorithm 0 has been presented in a general way. In particular, we have left open the question of which path from *OPEN* should be extended next. Specific algorithms can be instantiated from algorithm 0 by choosing a specific priority policy. For example, the seminaive algorithm [Ban85] employs a BFS policy, the graph algorithms of [IRW] employ a DFS policy, and the direct algorithms of [ADJ90] employ a matrix ordering policy based on some numbering of the nodes in a graph. Algorithm 0 is meant to illustrate the optimization techniques that follow. Its generality is preserved in the refinements T, T_{pc}, and T_{psc} that are presented later in this chapter. The refinements, and all optimizations discussed with respect to them, are equally applicable to all specific instantiations.

In Section 4.4.4 we define two correctness criteria, *completeness* and *irredundancy*, for generalized transitive closure algorithms [ADJ90]. An algorithm is complete if it generates every path in the graph (implicitly or explicitly) at least once. An algorithm is irredundant if it generates every path in the graph at most once. Algorithm 0 is complete since it directly follows our query model: it generates every path in the graph, and tests whether it satisfies the specified selections and, therefore, should be in the result. The fact that this test is done during the generation of paths, rather than after all paths have been generated, does not affect its completeness. As stated, algorithm 0 may be redundant, because it does not ensure that the same path will not be generated in more than one way. However, we believe that all generalized transitive closure algorithms of interest represented by algorithm 0 and its refinements are irredundant. The way in which these algorithms avoid redundancy depends on the order in which paths are expanded, as we demonstrate in Chapter 5. To keep the generality of the presentation, we ignore the problem of irredundancy for the rest of this Chapter.

Algorithm 0 is very inefficient. It always computes all possible paths first, and applies the selection at the end. Is it possible to apply the selections earlier, as paths are being generated?

4.3.2 Utilizing Selections During the Transitive Closure Computation

We observe that there are four possible places in which selections can be applied during the computation of a transitive closure:

1. As a preprocessing step, to restrict the input graph G (initial value of ALL). We denote such selection criteria by $C_{preprocess}$.
2. To restrict the set of arcs the algorithm should start traversing from (initial value of $OPEN$). We denote such selection criteria by $C_{initial}$.
3. During the recursive enumeration of paths, to restrict the set of known paths that need to be further extended (next value of $OPEN$). We denote such selection criteria by $C_{intermediate}$.
4. After paths have been enumerated, to select the paths that should be in the result (final value of $RESULT$). We denote such selection criteria by C_{final}.

Let us now present a modification of Algorithm 0 that takes advantage of these selections:

Algorithm Template T:

```
1. Given $C$, generate $C_{preprocess}$, $C_{initial}$, $C_{intermediate}$, $C_{final}$.

2. Initialization:
   $ALL = \{E \mid E \text{ is an arc in } G \wedge C_{preprocess}(E) = TRUE\}$
   $OPEN = \{E \mid E \in ALL \wedge C_{initial}(E) = TRUE\}$
   $CLOSE = ALL - OPEN$
   $RESULT = \emptyset$.

3. Do steps 4 through 6, while $OPEN \neq \emptyset$

4. Choose a path $P \in OPEN$. Extend this path by concatenating to it some path $P_{extend} \in ALL$.
   Let the resulting path be $P_{new} = P \circ P_{extend}$.
   Compute the path labels of $P_{new}$ by applying the specified $CON$ functions.

5. Apply $C_{intermediate}$ to $P_{new}$.
   If $C_{intermediate}(P_{new}) = TRUE \wedge P_{new} \notin ALL$ then
      $ALL = ALL + P_{new}$, $OPEN = OPEN + P_{new}$

6. If $P$ has been extended in all possible ways then
      $OPEN = OPEN - P$ , $CLOSE = CLOSE + P$
      Apply $C_{final}$ to $P$.
      If $C_{final}(P) = TRUE$ then $RESULT = RESULT + P$
```

The following question should be asked at this stage: "Given a constraint C, how can the constraints $C_{preprocess}$, $C_{initial}$, $C_{intermediate}$, and C_{final} be derived?"

Different choices of these constraints in step 1 result in different algorithms, as we shall shortly see. For the present, note that if $C_{preprocess} = C_{initial} = C_{intermediate} = \emptyset$, $C_{final} = C$, we obtain Algorithm 0.

A constraint C is called *simple* if it is free of conjunctions. We first demonstrate the transposition of C to $C_{preprocess}$, $C_{initial}$, $C_{intermediate}$, or C_{final} assuming that C is a simple constraint.

4.3.3 Classification of Selection Criteria

Examples in the following discussion refer to path queries over an airline database holding information about the source, destination, departure time, arrival time, ticket price and airline of each flight.

Definition (Monotonically Negative Constraint):

A constraint C is **monotonically negative** if for every graph G, for every path P, if P satisfies C then every subpath P^- of P satisfies C. Conversely, if there exists a subpath P^- of P that violates C, then P violates C.

Examples:

For every pair of consecutive connections, layover time
between 1 and 3 hours.
Total ticket price less than 2000.

Definition (Extension-Monotonic Constraint):

A constraint C is **extension-monotonic** if for every graph G, for every path P, for every extension $P^{\rightarrow}$ of P, $P^{\rightarrow}$ satisfies C if and only if P satisfies C.

Examples:

Source node is New York.
First connection with KLM.

Definition (Non-Monotonic Constraint):

A constraint C is **non-monotonic** if C is not monotonically negative or extension-monotonic.

Examples:

Average connection price less than 300.
Destination node is Paris (assuming the source node
is fixed, e.g. New York).

Definition (Decomposable Constraint):

A constraint C is **decomposable** if for every graph G, for every path P, P satisfies C if and only if every subpath P^- of P satisfies C.

Examples:

Maximum connection price less than 500.
No flights with 'KLM'.

A decomposable constraint by definition is monotonically negative, but the inverse is not true. For example, the total ticket price of a flight sequence may be more than 2000 even if the total cost of each subsequence is less than 2000. Note that the definition of extension-monotonic constraints holds for paths that grow in one direction, starting at the source node of the path. However, when computing the closure, we can choose either end of a path as its source node.

4.3.4 Selection Transposition Algorithms

We now state three algorithms that apply selections as early as possible. The algorithms are derived from the template algorithm T, through appropriate choice of selection conditions. In Table 4.1, Algorithm 0 has been restated for reference purposes.

Algorithm	$C_{preprocess}$	$C_{initial}$	$C_{intermediate}$	C_{final}
Alg. 0	$\varnothing$	$\varnothing$	$\varnothing$	C
Alg. 1	C	$\varnothing$	C	$\varnothing$
Alg. 2	$\varnothing$	C	$\varnothing$	$\varnothing$
Alg. 3	C	$\varnothing$	$\varnothing$	$\varnothing$

TABLE 4.1. Selection Transposition Algorithms

For example, the fourth entry states that if a constraint C is decomposable, it should be applied as $C_{preprocess}$. Algorithm T with this transformation will produce exactly the same result as it would if C is naively applied as C_{final}.

Theorem 4.1 (Transposition of Monotonically Negative Constraints)

Algorithm 1 produces the same result as Algorithm 0 for every input graph if and only if C is monotonically negative.

Proof

Paths are added to *ALL* either in step 2, or in step 5. In both cases they are guaranteed to satisfy C, applied as $C_{preprocess}$ in step 2, and as $C_{intermediate}$ in step 5. Every path in *RESULT* has previously been added to *ALL*, and therefore satisfies C. Hence, Algorithm 1 is sound.

A path P is not added to *RESULT* only if it violates C, or it is a superpath of a path P^- that violates C, and was discarded in steps 2 or 5. In the second case P must also violate C, since C is monotonically negative. Algorithm 1 does not miss any paths that satisfy C. Thus it is complete.

If C is not monotonically negative then there exist a graph G and a path P in G and a superpath P^+ of P in G such that P violates C and P^+ satisfies C. When running Algorithm 1 on G, P will not be added to *ALL*, since it violates $C_{intermediate}$. Therefore, superpaths of P may not be generated, and in particular P^+ may not be in the result. ■

Theorem 4.2 (Transposition of Extension-Monotonic Constraints)

Algorithm 2 produces the same result as Algorithm 0 for every input graph if and only if C is extension-monotonic.

Proof

Assume C is extension-monotonic. Given a path P, because C is extension-monotonic it follows that P satisfies C if and only if its first arc satisfies C. Algorithm 2 considers only and all paths whose first arc satisfies C, and is therefore sound and complete.

If C is not extension-monotonic then there exist a graph G and a path P in G and an extension $P^{\rightarrow}$ of P in G such that either (i) P violates C and $P^{\rightarrow}$ satisfies C, or (ii) P satisfies C and $P^{\rightarrow}$ violates C. The only selection applied by Algorithm 2 is the application of $C_{initial}$ to initialize *OPEN*. Every path in *OPEN* is later added to *RESULT*. In case (i) P will not be inserted to *OPEN*. Therefore, $P^{\rightarrow}$ may never be generated. In case (ii) P will be inserted to *OPEN* at some point. Therefore, $P^{\rightarrow}$ may be generated and inserted to *OPEN* at a later point. In case (i) neither P nor $P^{\rightarrow}$ may be in *RESULT* when the algorithm terminates. In case (ii) both of them may be in *RESULT*. In either case, the result may not be identical to Algorithm 0. ■

Theorem 4.3 (Transposition of Decomposable Constraints)

Algorithm 3 produces the same result as Algorithm 0 for every input graph if and only if C is decomposable.

Proof

Assume C is decomposable. Given a path P, its easy to see by induction on the number of arcs in P that P satisfies C if and only if P is comprised solely of arcs in G that satisfy C. Since *ALL* is initialized to only and all arcs in G that satisfy C, paths comprised of these arcs are guaranteed to satisfy C, and no other paths can satisfy C. Thus Algorithm 3 is sound and complete.

The proof of the only-if direction is similar to Theorem 4.2. If C is not decomposable then there exist a graph G and a path P in G and a subpath P^{-} of P in G such that either (i) P^{-} violates C and P satisfies C, or (ii) P^{-} satisfies C and P violates C. The only selection applied by Algorithm 3 is the application of $C_{preprocess}$ to initialize *ALL*. Every path in *ALL* is later added to *RESULT*. In case (i) P^{-} will not be inserted to *OPEN*. Therefore, P may never be generated. In case (ii) P^{-} will be inserted to *ALL* at some point. Therefore, P may be generated and inserted to *ALL* at a later point. In case (i) neither P^{-} nor P may be in *RESULT* when the algorithm terminates. In case (ii) both of them may be in *RESULT*. In either case, the result may not be identical to Algorithm 0. ■

4.3.5 Conjunction of Simple Constraints

We have so far assumed that C is a simple constraint. We now consider the case where C is a conjunction of simple constraints. Given a conjunction of simple constraints. $C = C_1 \wedge C_2 \wedge \cdots \wedge C_n$, and a path-set ψ, let $\hat{\sigma}_C(\psi)$ stand for the optimized evaluation of $\sigma_C(\psi)$ by transposing independently each of the simple constraints C_i according to Theorems 4.1, 4.2, 4.3.

Theorem 4.4 (Transposition of a Conjunction of Constraints)

Let G be a directed graph, and let ψ be the set of paths in G. Then: $\sigma_C(\psi) = \hat{\sigma}_C(\psi)$

Proof

We show that $\sigma_C(\psi) = \hat{\sigma}_C(\psi)$ by induction on n the number of conjuncts. Theorems 4.1, 4.2, 4.3 cover the case $n = 1$. For $n > 1$:

$$\begin{aligned}
\sigma_C(\psi) &= \sigma_{C_1 \wedge C_2 \wedge \cdots \wedge C_n}(\psi) \\
&= \sigma_{C_1 \wedge C_2 \wedge \cdots \wedge C_{n-1}}(\sigma_{C_n}(\psi)) && \textit{by property of conjunction} \\
&= \sigma_{C_1 \wedge C_2 \wedge \cdots \wedge C_{n-1}}(\hat{\sigma}_{C_n}(\psi)) && \textit{by Theorems 4.1, 4.2, 4.3 on simple constraints} \\
&= \hat{\sigma}_{C_1 \wedge C_2 \wedge \cdots \wedge C_{n-1}}(\hat{\sigma}_{C_n}(\psi)) && \textit{by induction assumption} \\
&= \hat{\sigma}_{C_1 \wedge C_2 \wedge \cdots \wedge C_n}(\psi) = \hat{\sigma}_C(\psi) && \textit{by property of conjunction}
\end{aligned}$$

■

4.3.6 Examples

Example 1:

Given relation Flights (Src, Dest, Time, Airline, Price, Veg_Meal), find all flights from Los Angeles to Jerusalem where the first connection is with United Airlines, the total ticket price is less than 1500 dollars, each layover time is between one and two hours, and all connections offer a vegetarian meal.

Assuming we choose to treat the start of the flight sequence as the source node, the constraints "start node is Los Angeles" and "first connection is with United Airlines" are extension-monotonic and can be used to restrict the initial set. The constraints "layover time is between one and two hours" and "total ticket price less than 1500 dollars" are monotonically negative, and can be used to control the generation of new paths. The latter constraint should also be applied to the input relation to eliminate all single flights that cost more than 1500 dollars. The constraint "all connections offer a vegetarian meal" is decomposable. It can be applied to the input relation to eliminate connections on which a vegetarian meal is not offered. Once this has been done, it can be ignored. The constraint "end node is Jerusalem" is non-monotonic, and must be evaluated for every path. We could have chosen to treat the end of the flight sequence as the source node, in which case the constraint on the end node could be utilized, but the constraints on the first node and first connection could not be transposed. As a general heuristic, if constraints are specified on both endpoints, the endpoint with fewer possible values should be selected as source. The decision depends on the expected selectivity of the specified constraints considering the distribution of values for the corresponding attributes.

Example 2:

Given relation Network (Src, Dest, Delay, Capacity), find all paths emanating from nodes in set START-NODES that do not go through nodes in set BAD-NODES, with at most 5 arcs, where the maximum delay is less than 5 seconds and the average delay more than 1 second, and where the capacity of each arc is at most 20% different than the previous arc.

The constraint "paths from START-NODES" is extension-monotonic and can be used to restrict the initial set. The constraints "at most 5 arcs" and "capacity of each arc at most 20% different from previous arc" are monotonically negative. They can be used to control the generation of new paths. The constraints "do not go through nodes in BAD-NODES" and "maximum delay less than 5 seconds" are decomposable and can be applied to the input relation. The constraint "average delay more than 1 second" is non-monotonic, and must be evaluated for every path.

In presenting these examples, we have assumed that the correct classification of the constraints specified in the query was known. In general, the query optimizer is required to perform such a classification in order to use the selection transposition techniques presented above. One table-driven approach to accomplish this classification is to let the semantic analyzer recognize what type of constraints are present in the query by consulting a prestored table. The optimizer then chooses the correct optimization method for each constraint, which may involve rewriting the constraint and moving it to another place in internal representation of the query. In Appendix 4, we present a constraint classification table for the constraints admitted by SQL/TC.

Our classification of selection criteria, and the selection transposition optimizations that use this classification, are not limited to generalized transitive closure queries. They can be applied to any SQL query involving aggregation, where a constraint is specified on the aggregated value. To the best of our knowledge, such optimization of aggregation queries is not employed in current SQL implementations (see also [MPR90]).

4.4 Representing Paths and Path-Sets

In this section we consider the storage requirement for representing paths and path-sets in the transitive closure. If paths and path-sets are represented by the complete sequence of nodes and arcs, the closure computation would require storage exponential in the size of the input graph. We approach this problem by introducing condensed representations for paths and path-sets. We show that in general the condensed representation is insufficient since it may not be possible to compute the path or path-set label and to evaluate the specified selections with the condensation alone. We then identify classes of queries for which the condensation does suffice.

4.4.1 Representing Paths

We first discuss the case where label concatenation is specified, but label aggregation is not specified.

Definition (Condensation of a Path)

Let $P(s,t) = \{ E_k \}, k = 1, \ldots, n$ be a path from s to t. Let L_i be the label of E_i. Let $P(s,t)$ have a path label defined by $L(s,t) = CON(L_1, L_2, \cdots, L_n)$. The *path condensation* of $P(s,t)$ is the triple <s, t, $L(s,t)$>. We denote the condensation of a path P by $\bar{P}$. In general, several *CON* functions may be specified, in which case the path label $L(s,t)$ is a vector of values. Figure 4.6 shows a path and its associated condensation. The *distance* label was calculated as the sum of the arc lengths.

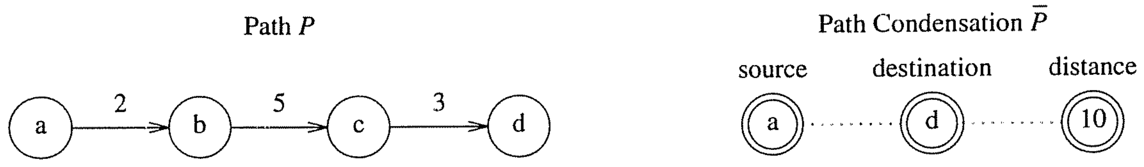


Figure 4.6. Path Condensation

The path condensation of a path P serves as a compact representation of P, given by the projection of P on its source, destination, and label attributes. Given two paths $P(s,t)$ and $P(t,w)$ represented by $\bar{P}(s,t)$ and $\bar{P}(t,w)$, the concatenation of $P(s,t)$ and $P(t,w)$ is achieved by *composition* of $\bar{P}(s,t)$ and $\bar{P}(t,w)$:

$<s, t, L(s,t)> \circ <t, w, L(t,w)> = <s, w, CON(L(s,t), L(t,w))>$

A Template Transitive Closure Algorithm Using Path Condensation

The template algorithm T of Section 4.2.3.2 can be modified to make use of path condensation. *ALL, OPEN, RESULT* and *CLOSE* become sets of path condensations. Every "P" term in the algorithm is replaced by a "$\bar{P}$" term. In Step 4, the label of the new path condensation $\bar{P}_{new}$ is computed from the path condensations of $\bar{P}$ and $\bar{P}_{extend}$. Step 5 simplifies to checking for the existence of alternative paths between the same endpoints with identical label values:

```
4. ...
  Compute the path labels of $\bar{P}_{new}$ from $\bar{P}$ and $\bar{P}_{new}$.
5. Apply $C_{intermediate}$ to $\bar{P}_{new}$.
  If $C_{intermediate}(\overline{P}_{new}) = TRUE \wedge \bar{P}_{new} \notin ALL$ then
    $ALL = ALL + \bar{P}_{new}$, $OPEN = OPEN + \bar{P}_{new}$
```

Note that if label aggregation is also specified, the duplicate elimination step could be erroneous (see Section 4.4.3). We refer to the modification of T to use path condensation as T_{pc}.

Implicitly, most existing generalized transitive closure algorithms (for example, [AgJ90, ADJ90, CrN89, IRW, Jia90, UlY90]) assume that paths can suitably be represented by their condensation.

However, in general, path condensation may not be used for two reasons:

1. The label concatenation function *CON* may not be computable using the path condensation.
2. Selection criteria which cannot be evaluated against the path condensation may be specified on the desired paths.

We first examine the problem of label computation. We then look at evaluation of selection criteria.

4.4.1.1 Label Concatenation Using Path Condensation

Assume that we wish to compute the MEDIAN of the labels along every path in the graph. When concatenating two paths P_1 and P_2, we have no way of establishing the median of the resulting path from the medians of P_1 and P_2. In fact, we require the whole sequence of arc labels in P_1 and P_2 (or an encoding of similar complexity) to compute the path label.

Definition (Incrementally Computable Path Label)

A path label L is *incrementally computable* if for every two paths $P(s,t)$ and $P(t,w)$, the path label of $P(s,w) = P(s,t) \circ P(t,w)$, can be computed directly from $\bar{P}(s,t)$ and $\bar{P}(t,w)$ without referencing individual arcs in $P(s,t)$ or $P(t,w)$.

Theorem 4.5

Given a query Q that includes no label aggregation functions or selections, algorithm T_{pc} computes the correct result for any input graph if and only if every path label specified in Q is incrementally computable.

Proof

Algorithm T_{pc} is a straightforward refinement of T where every path is represented by its condensation, and concatenation is performed by composition. T_{pc} is correct if and only if whenever two path condensations $\overline{P}(s,t)$ and $\overline{P}(t,w)$ are composed to form a path condensation $\overline{P}(s,w)$, we can compute every label of $\overline{P}(s,w)$ from $\overline{P}(s,t)$ and $\overline{P}(t,w)$, which is true if and only if all the path labels of $P(s,w)$ are incrementally computable. ■

Theorem 4.6

A path label L is incrementally computable if, for every path P, either of the following two conditions hold:

1. Let $S = AL_1, AL_2, \cdots, AL_n$ be the sequence of arc labels in P. There exists a binary function $BCON$ such that:

(a) $L(P) = BCON(...BCON(BCON(AL_1, AL_2), AL_3),..., AL_n)$

and

(b) $BCON$ is associative:
$\forall\ x, y, z\ BCON(BCON(x, y), z) = BCON(x, (BCON(y, z))$

2. $L(P)$ can be computed as function f of path labels $L_1, \cdots, L_k$, all of which are incrementally computable.

Proof

Let $P(s,t)$ and $P(t,w)$ be two paths and let $P(s,w) = P(s,t) \circ P(t,w)$.

1. Condition (a) implies that we can replace a sequence of path labels by the result of applying $BCON$ to the subsequence. Condition (b) allows us to choose subsequences in any order. Therefore:

   $L(P(s,w)) = L(P(s,t) \circ P(t,w)) = BCON(L(P(s,t)), L(P(t,w)))$

2. Given path condensations $\overline{P}(s,t)$, $\overline{P}(t,w)$, we can compute $L_1(P(s,w)), L_2(P(s,w)), \cdots, L_n(P(s,w))$ since these are incrementally computable. Therefore, we can also compute $L(P(s,w))$, by: $L(P(s,w)) = f(L_1(P(s,w)), L_2(P(s,w)), \cdots, L_n(P(s,w)))$. ■

The second condition of Theorem 4.6 allows us to define a CON function in terms of other CON functions.

Examples

1. Define path label AVG by AVG = $\frac{\text{SUM}}{\text{COUNT}}$. Since both SUM and COUNT are incrementally computable path labels by the first condition (using addition as the binary functions in both cases), their ratio is incrementally computable by the second condition.

2. Given a path $P = \{ E_1, E_2, \cdots, E_k \}$, let L_i be the label of E_i, i=1,2,...,k. We wish to compute the path label SUB defined by

   $\mathrm{SUB} = L_1 - L_2 - \cdots - L_n$

   SUB can be computed by defining it in terms of two incrementally computable labels, FIRST, whose value is the first label of a path, and SUM, defined as usual:

   $\mathrm{SUB}(P) = \mathrm{FIRST} - [\ \mathrm{SUM}(P) - \mathrm{FIRST}(P)\ ] = 2{*}\mathrm{FIRST}(P) - \mathrm{SUM}(P)$

4.4.2 Arc Selection (ξ)

Assume that an arc selection predicate ξ was associated with the label concatenation function *CON*. The *CON* function should therefore be applied not to the complete set of arcs in a path, but only to those arcs that satisfy ξ. Let $L(P)$ be the path label computed by applying *CON* to all arcs in path *P*, and let $\overline{P}$ be the corresponding path condensation of *P*. Let $L'(P)$ be the path label of *P*, and $\overline{P'}$ be the path condensation of *P*, computed by applying *CON* only to the selected arcs in *P*.

Theorem 4.7

If L is incrementally computable then L' is incrementally computable.

Proof

Let *G* be a graph. Let $P(s,t)$ and $P(t,w)$ be two paths in *G*, and let $P(s,w) = P(s,t) \circ P(t,w)$. We need to show that $L'(P(s,w))$ can be computed directly from $\overline{P'}(s,t)$ and $\overline{P'}(t,w)$ without referencing individual arcs in $P(s,t)$ or $P(t,w)$.

The computation of $L'(P(s,w))$ on the graph *G* is equivalent to the computation of $L(P(s,w))$ on a graph G', which is identical to *G*, except that the label of any arc that violates ξ is set to ε, the unit of the function *CON* (see Section 2.4). Because L is incrementally computable, $L(P(s,w))$ can be computed from $\overline{P}(s,t)$ and $\overline{P}(t,w)$ in G'. Since $\overline{P}(s,t)$ and $\overline{P}(t,w)$ in G' are equal to $\overline{P}(s,t)$ and $\overline{P}(t,w)$ in G', it follows that $L'(P(s,w))$ can be computed from $\overline{P'}(s,t)$ and $\overline{P'}(t,w)$ in *G*. ■

Theorems 4.5 and 4.7 together imply that Algorithm T_{pc} may be used for path enumerations involving incrementally computable path labels and arc selections. Step 4. of algorithm T_{pc} is modified trivially to compute the path labels of the new path condensation $\overline{P'}_{new}$ from the path condensations $\overline{P'}$ and $\overline{P'}_{extend}$.

4. ...
   Compute $\overline{P'}_{new}$ from $\overline{P'}$ and $\overline{P'}_{new}$.

Reconsidering the examples above, computing the SUM or AVG of selected arcs in a path is no more difficult then computing the SUM or AVG of all the arcs in a path. It can be done using Algorithm T_{pc}, where the SUM, COUNT, and AVG path labels are computed with respect to qualifying arcs only.

Theorem 4.8

Given a path label $L(P)$ defined by an arbitrary *CON* function on attribute *Lab*, we can compute $L(P)$ by introducing a path label that requires at most p times the storage of a *Lab* value, where p is the length of the longest path in the graph.

Proof

The label can simply accumulate (for example, by concatenation) the arc labels along each path. When the path is fully extended we can apply *CON* to compute the path label. ■

In the remainder of this section we assume that every path label is incrementally computable, either by some binary label concatenation function *BCON*, or as a function of other path labels.

4.4.2.1 Evaluation of Selection Criteria Using Path Condensation

When selection criteria are specified in the query, even if every path label is incrementally computable, we may not be able to use path condensation since the selection requires knowledge of individual arcs. For example, assume that we need to evaluate a selection on arcs that are k ($k > 1$) apart. In particular, consider an airline database, in which we wish to find a flight sequence where a meal is served on every third connection. When concatenating two paths P_1 and P_2, we need to know at least the last two arcs of P_1 and first two arcs of P_2 before it can be determined if the resulting path satisfies the constraint. Thus, if P_1 and P_2 are represented by their condensation, we cannot evaluate this selection criterion.

What are the selection criteria for which the path condensation holds sufficient information?

Definition (Incrementally Computable Selections)

Let C be a selection criterion, P a path generated by concatenating paths P_1 and P_2, and $C(P)$ the evaluation of C on P. C is *incrementally computable* if $C(P)$ can be determined directly from $\overline{P}_1$ and $\overline{P}_2$ without referencing individual arcs in P_1 or P_2.

Incrementally computable selections cover many useful selections that arise in practice. Let C be a selection criterion, and P be a path generated by concatenating paths P_1 and P_2. We demonstrate some incrementally computable constraints by showing how $C(P)$ can be evaluated from $\overline{P}_1$ and $\overline{P}_2$. The Flight relation is used as a running example.

1. Selection on **start arcs or nodes** (e.g., all cities reachable from Paris): $C(P) = C(\overline{P}_1)$.

2. Selection on **end arcs or nodes** (e.g., all cities that can reach Monaco): $C(P) = C(\overline{P}_2)$.

3. Selection on **all arcs or nodes** in the path (e.g., all flights by European airlines, no flights through Chicago): $C(P) = C(\overline{P}_1)$ AND $C(\overline{P}_2)$.

4. Selection on **some arcs or nodes** in the path (e.g., at least one stop in the midwest): $C(P) = C(\overline{P}_1)$ OR $C(\overline{P}_2)$.

5. Selection on **consecutive arcs** (e.g., arrival time of each flight between three hours and one hour before the departure time of the next flight): such a constraint is incrementally computable by definition, since it directly specifies how $C(P)$ is computed from labels of $\bar{P}_1$ and $\bar{P}_2$ when $\bar{P}_1$ and $\bar{P}_2$ are concatenated.

6. Selection on the **path label** (e.g., total ticket price less than 1000 dollars): such a constraint can be evaluated incrementally, if the specified path label is incrementally computable. The example given here uses the SUM path label, which is incrementally computable.

Theorem 4.9

Given a query Q that includes no label aggregation functions, Algorithm T_{pc} computes the correct result for any input graph if and only if every selection specified in Q is incrementally computable.

Proof is similar to that of Theorem 4.5.

4.4.3 Representing Path-Sets

We now discuss the case of path aggregation, where both label concatenation and label aggregation are specified in the query.

Theorem 4.10[11]

Given a path-set label $L(\psi)$ defined by an arbitrary *CON* and *AGG* functions on attribute *Lab*, if $L(\psi)$ can take a number of values that is bounded by *K*, then $L(\psi)$ can be computed by introducing a path label that requires at most $O(K)$ times the storage of a *Lab* value.

Proof

We prove the theorem by construction. We introduce a path-set label that accumulates the arc labels along each path in the path-set by keeping a list of <*Lab* value, counter> pairs. To apply aggregation, the lists of the respective path-sets are unioned, merging items with identical *Lab* values by adding their counters. To apply concatenation, a pair-wise concatenation of list items is performed, by applying the *CON* function on the values and multiplying the counters in each pair. Conceptually, this is equivalent to breaking each <*L*, *n*> item into the *n* <*L*, 1> items it represents. Finally, the *AGG* function is applied to complete path-sets. Because of the bound *K* on $L(\psi)$ values, such a list cannot exceed *K* items. ■

As an example, consider the case where the labels are drawn from the boolean domain. Regardless of the definition of *CON* and *AGG*, we can evaluate this query using path set labels of size $O(2)$.

We now introduce a compact representation for sets of path, allowing paths and their associated path labels to be merged into a single "path-set condensation", with a resulting saving in storage. We then identify classes of

11. Discussions with M. Yannakakis helped us crystalize this theorem.

aggregation queries that can be evaluated using this representation.

Definition (Condensation of a Path-Set)

Let $\psi(s,t) = \{ P_k(s,t) \} \; k = 1,...,m$ be a path-set from s to t. Let L_i be the label of P_i. Let $\psi(s,t)$ have a path-set label defined by $L(s,t) = AGG(L_1, L_2, \cdots, L_m)$. The *path-set condensation* of ψ is the triple <s, t, $L(s,t)$>. We denote the condensation of a path-set $\psi(s,t)$ by $\overline{\psi}(s,t)$. In general, several *AGG* functions may be specified, in which case the path-set label $L(s,t)$ is a vector of values. Figure 4.7 shows a path-set and its associated condensation. The *min.distance* path-set label was calculated as the minimum of sums of the arc lengths.

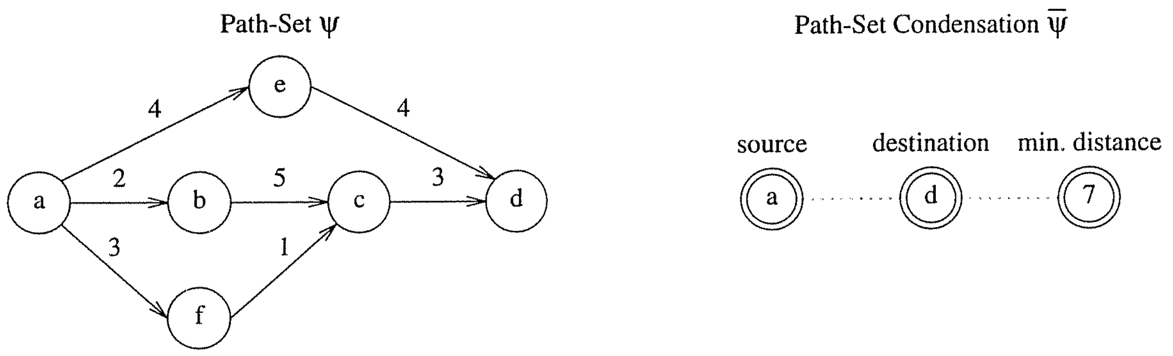


Figure 4.7. Path-Set Condensation

A Template Transitive Closure Algorithm Using Path-Set Condensation

The template algorithm T of Section 4.3.2 can be modified to make use of path-set condensation. Every "P" term in the algorithm is replaced by a "$\overline{\psi}$" term. *ALL*, *OPEN*, *RESULT* and *CLOSE* become sets of path-set condensations. We extend the definition of *concatenation* to path-sets as follows:

$$\psi(s,t) \circ \psi(t,w) = \{ P(s,t) \circ P(t,w) \mid P(s,t) \in \psi(s,t),\ P(t,w) \in \psi(t,w) \}$$

The duplicate test in step 5 is replaced by aggregation, as follows:

```
5. Let $\overline{\psi}_{new}$ be the newly generated path-set condensation.  Apply $C_{intermediate}$ to $\overline{\psi}_{new}$.
  If $C_{intermediate}(\overline{\psi}_{new}) = TRUE$ then
    If there is no path-set condensation between the endpoints of $\overline{\psi}_{new}$
      then $ALL = ALL + \overline{\psi}_{new}$, $OPEN = OPEN + \overline{\psi}_{new}$
    Else
      Let $\overline{\psi'}_{new}$ be the current path-set condensation between the endpoints of $\overline{\psi}_{new}$.
      Let $L'_{new}$ be the label of $\overline{\psi'}_{new}$, and $L_{new}$ be the label of $\overline{\psi}_{new}$.
      Set $L'_{new} = AGG(L'_{new}, L_{new})$ /* merge $\overline{\psi'}_{new}$ and $\overline{\psi}_{new}$ */
```

We refer to the modification of T to use path-set condensation as T_{psc}.

In general, path-set condensation may not be used for two reasons:

1. The label concatenation and aggregation functions *CON* and *AGG* may not be computable using the path-set condensation.

2. Selection criteria may be specified on the desired paths, which cannot be evaluated against the path-set condensation.

4.4.3.1 Label Computation Using Path-Set Condensation

Assume a path-set label was specified in a query, using label concatenation and aggregation functions *CON* and *AGG* respectively. Can we compute that label using path-set condensations?

Definition (Independently Computable Path-Set Label)

A path-set label L is *independently computable* if the following two conditions hold:

1. For every two path-sets $\psi(s,t)$ and $\psi(t,w)$, the path-set label of $\psi(s,w) = \psi(s,t) \circ \psi(t,w)$, can be computed directly from $\overline{\psi}(s,t)$ and $\overline{\psi}(t,w)$ without referencing individual paths or arcs in $\psi(s,t)$ or $\psi(t,w)$.

2. For every two path-sets $\psi^1(s,t)$ and $\psi^2(s,t)$ the path-set label of $\psi(s,t) = \psi^1(s,t) \cup \psi^2(s,t)$, can be computed directly from $\overline{\psi^1}(s,t)$ and $\overline{\psi^2}(s,t)$ without referencing individual paths or arcs in $\psi^1(s,t)$ or $\psi^2(s,t)$.

Theorem 4.11

Given a query Q with no selections specified, algorithm T_{psc} computes the correct result for any input graph if and only if each path-set label specified in Q is independently computable.

Proof

Algorithm T_{psc} is a refinement of T and is therefore complete. Assume path-set label L was specified using functions *CON* and *AGG*. For every path-set $\psi(s,w)$, let the label computed for $\psi(s,w)$ by algorithm T_{psc} be $L_{psc}(s,w)$. Let the correct label for $\psi(s,w)$ defined according to our model be $L(s,w) = AGG\ CON\ (\psi(s,w))$.

First, assume that every path-set label specified in Q is independently computable. We need to show that $L_{psc}(s,w) = L(s,w)$.

There are two ways in which $\psi(s,w)$ could have been generated by algorithm T_{psc}. We show that in either way $L_{psc}(s,w) = L(s,w)$. The proof relies on induction on the number of applications of the binary concatenation and aggregation functions *BCON* and *BAGG* that generated $L_{psc}(s,w)$:

1. If $\psi(s,w)$ was generated by concatenation, that is, $\psi(s,w) = \psi(s,t) \circ \psi(t,w)$, then $L_{psc}(s,w) = BCON(L_{psc}(s,t), L_{psc}(t,w))$.

   By the induction assumption, it follows that:

   $L_{psc}(s,t) = L(s,t) = AGG\ CON\ (\psi(s,t))$, and $L_{psc}(t,w) = L(t,w) = AGG\ CON\ (\psi(t,w))$.

   Because L is independently computable:

$L(s,w) = BCON(L(s,t), L(t,w)) = BCON(L_{psc}(s,t), L_{psc}(t,w)) = L_{psc}(s,w)$

2. If $\psi(s,w)$ was generated by union, that is, $\psi(s,w) = \psi^1(s,w) \cup \psi^2(s,w)$, then $L_{psc}(s,w) = BAGG(L^1_{psc}(s,w), L^2_{psc}(s,w))$

   By the induction assumption, it follows that:

   $L^1_{psc}(s,w) = L_1(s,w) = AGG\ CON\ (\psi^1(s,w)$, and $L^2_{psc}(s,w) = L_2(s,w) = AGG\ CON\ (\psi^2(s,w))$.

   Because L is independently computable:

   $L(s,w) = BAGG(L^1(s,w), L^2(s,w)) = BAGG(L^1_{psc}(s,w), L^2_{psc}(s,w)) = L_{psc}(s,w)$

Assume Q specifies a path-set label L' that is not independently computable. Then, there exists a graph G such that one of the following is true:

1. There exist two path-sets $\psi(s,t)$ and $\psi(t,w)$ in G, such that the path-set label of $\psi(s,w) = \psi(s,t) \circ \psi(t,w)$, cannot be computed directly from $\overline{\psi}(s,t)$ and $\overline{\psi}(t,w)$.

2. There exist two path-sets $\psi^1(s,t)$ and $\psi^2(s,t)$ in G, such that the path-set label of $\psi(s,t) = \psi^1(s,t) \cup \psi^2(s,t)$, cannot be computed directly from $\overline{\psi^1}(s,t)$ and $\overline{\psi^2}(s,t)$.

In either case algorithm T_{psc} may fail to compute $L(s,t)$. ■

Theorem 4.12

A path-set label L is independently computable if either of the following two conditions hold:

1. There exist binary functions $BCON$ and $BAGG$ such that for every path-set ψ, $L(\psi)$ can be computed by $BCON$ and $BAGG$, and $BCON$ and $BAGG$ form a path algebra (see Section 2.4, [AHU75], [Car78]).

2. For every path-set $\psi(s,t)$, the path-set label of $\psi(s,t)$, can be computed as a function f of path-set labels $L_1, \cdots, L_n$, which are all independently computable.

Proof

1. The proof of the first condition follows from [AHU75].

2. Let $\psi(s,t)$ and $\psi(t,w)$ be two path-sets, and let $\psi(s,w) = \psi(s,t) \circ \psi(t,w)$. Given path-set condensations $\overline{\psi}(s,t)$ and $\overline{\psi}(t,w)$, we can compute $L_1(\ \psi(s,t)\ ), L_2(\ \psi(s,t)\ ), \cdots, L_n(\ \psi(s,t)\ )$ since these are independently computable. Therefore, we can also compute $L(\psi(s,t))$, by $L(\psi(s,t)) = f(L_1(\psi(s,t)), L_2(\psi(s,t)), \cdots, L_n(\psi(s,t)))$

   Let $\psi^1(s,t)$ and $\psi^2(s,t)$ be two path-sets, and let $\psi(s,t) = (\psi^1(s,t) \cup \psi^2(s,t))$. Given path-set condensations $\overline{\psi^1}(s,t)$ and $\overline{\psi^2}(s,t)$, we can compute $L_1(\psi(s,t)\ ), L_2(\psi(s,t)\ ), \cdots, L_n(\psi(s,t)\ )$ since these are independently computable. Therefore, we can also compute $L(\psi(s,t))$, by: $L(\psi(s,t)) = f(L_1(\psi(s,t)), L_2(\psi(s,t)), \cdots, L_n(\psi(s,t)))$.

■

Example

Given a part hierarchy, we wish to compute the following variation of the bill of materials: for every part we wish to know the average number of any subparts required for its assembly, averaged across all the paths from the part to that subpart. We can do so by defining *CON* to be MULT (product), and *AGG* to be AVG. But we cannot use the binary forms of MULT and AVG as *BCON* and *BAGG*, since this will not give us the correct result. Instead we can define the required path-set label as $\frac{\text{SUM}(\psi)}{\text{COUNT}(\psi)}$. The new definition requires two path-set labels, the first computed by CON_1 which is MULT, and AGG_1 which is SUM, and the second computed by CON_2 which is the constant '1', and AGG_2 which is COUNT. Both *CON* and *AGG* combinations form a path algebra, and are independently computable by the first condition of Theorem 4.12. Therefore the ratio of the corresponding path-set labels is independently computable by the second condition.

4.4.3.2 Selecting *BCON* and *BAGG*

For specified *CON* and *AGG* functions, we need to choose binary *BCON* and *BAGG* functions to compute the corresponding path-set label. Typically, there are ''trivial'' binary functions that can be used to compute *CON* and *AGG* respectively. For example, for the bill of materials problem, the path-set label is defined by CON = MULT, AGG = SUM, and can be computed using $BCON$ = "*", $BAGG$ = "+". However, if the trivial *BCON* and *BAGG* functions do not form a path algebra, do we have to keep the complete set of path labels in every path-set? Sometimes it is possible to find ''non-trivial'' *BCON* and *BAGG* functions that correctly compute the desired path-set label and do form a path algebra.

Example

Consider the paint problem, example A9 in Table 2.1. Given a graph where there is a label $L(s,t)$ on every arc $E(s,t)$ from node s to node t representing the 'effort' required to traverse this arc, we wish to find the total effort required to traverse the graph through all possible paths. To do so, we could use $BCON$ = "+", $BAGG$ = "+". However, since addition does not distribute over addition, the resulting path problem is not a path algebra.

To fix the problem we have to keep track of the number of paths in every path-set, and use that value to correctly handle the concatenation of two path-sets. Therefore we introduce a composite label $L'(s,t) = <L(s,t), CTR(s,t)>$. For every arc (i, j) in the graph, $CTR(i,j)$ is set to 1. The modified *BCON* and *BAGG* functions are:

Aggregation:

$$\text{BAGG}'(<L^1(i,j), CTR^1(i,j)>, <L^2(i,j), CTR^2(i,j)>) =$$
$$<L^1(i,j) + L^2(i,j), CTR^1(i,j) + CTR^2(i,j)>$$

Concatenation:

$$\text{BCON}'(<L(i,j), CTR(i,j)>, <L(j,k), CTR(j,k)>) =$$
$$<CTR(j,k)L(i,j) + CTR(i,j)L(j,k), CTR(i,j)\,CTR(j,k)>$$

We need to show that *BCON′* and *BAGG′* are equivalent to (that is, compute the same result as) *BCON* and *BAGG*, and that they form a path algebra (in particular, that *BCON′* distributes over *BAGG′*).

First, let us verify that this transformed problem indeed results in the same arc label values as the original problem. According to our model, for every two nodes s and t, for every path $P(s,t)$, we first apply *BCON′* to the arc labels of P. This results in a path label $L'_p(s,t) = <L_p(s,t),\ CTR_p(s,t)>$, where $L_p(s,t)$ is the sum of the arc labels along P, and $CTR_p(s,t)$ is 1. We then apply *BAGG′* to the set of paths between s and t. This results in a path-set label $L'(s,t) = <L(s,t),\ CTR(s,t)>$, where $L(s,t)$ is the sum of the $L_p(s,t)$ path labels, and $CTR(s,t)$ is the number of different paths between s and t. The resulting $L(s,t)$ label is exactly what was specified by the original *CON* and *AGG* functions (and what we wished to compute with the *BCON* and *BAGG* functions).

Next we need to show that *BCON′* distributes over *BAGG′*:

$$BAGG'(BCON'(L'(i,j),L'(j,k)),\ BCON'(L'(i,j),L'(j,l))) = BCON'(L'(i,j),\ BAGG'(L'(j,k),L'(j,l)))$$

Beginning with the left-hand side of the above equation, we can write:

$$\begin{aligned}
&BAGG'(BCON'(L'(i,j),L'(j,k)),\ BCON'(L'(i,j),L'(j,l)))\\
&= BAGG'(BCON'(<L(i,j),CTR(i,j)>,<L(j,k),CTR(j,k)>),\\
&\qquad BCON'(<L(i,j),CTR(i,j)>,<L(j,l),CTR(j,l)>))\\
&= BAGG'(<L(i,j)\,CTR(j,k)+L(j,k)\,CTR(i,j),CTR(i,j)\,CTR(j,k)>,\\
&\qquad <L(i,j)\,CTR(j,l)+L(j,l)\,CTR(i,j),CTR(i,j)\,CTR(j,l)>)\\
&= <L(i,j)\,CTR(j,k)+L(j,k)\,CTR(i,j)+L(i,j)\,CTR(j,l)+L(j,l)\,CTR(i,j),\\
&\quad CTR(i,j)\,CTR(j,k)+CTR(i,j)\,CTR(j,l)>\\
&= <(CTR(j,k)+CTR(j,l))\,L(i,j)+CTR(i,j)(L(j,k)+L(j,l)),\ CTR(i,j)(CTR(j,k)+CTR(j,l))>\\
&= BCON'(<L(i,j),CTR(i,j)>,\ <(L(j,k)+L(j,l)),(CTR(j,k)+CTR(j,l))>)\\
&= BCON'(<L(i,j),CTR(i,j)>,\ BAGG'(<L(j,k),CTR(j,k)>,<L(j,l),CTR(j,l)>))\\
&= BCON'(L'(i,j),BAGG'(L'(j,k),L'(j,l)))
\end{aligned}$$

■

For the rest of this section, we assume that every path-set label is incrementally computable.

4.4.4 Properties of Generalized Transitive Closure Algorithms

The presentation of template algorithm P_{psc} in Section 4.4.3 might suggest that every transitive closure algorithm can be trivially generalized to solve path problems by adding to it some label computation operations. In particular, the *CON* function can be applied when a new arc $E(i,k)$ is created by joining two arcs $E(i,j)$ and $E(j,k)$. That is:

$$(i,j,L_{ij}) \circ (j,k,L_{jk}) = (i,k,L_{ik}) \quad \text{where} \quad L_{ik} = CON(L_{ij},L_{jk})$$

Whenever multiple arcs get created between the same pair of nodes i,k, these arcs can be aggregated into a single arc with the *AGG* function:

$$(i,k,L^1_{ik}) \cup (i,k,L^2_{ik}) = (i,k,L_{ik}) \quad \text{where} \quad L_{ik} = AGG(L^1_{ik},L^2_{ik})$$

In this section we show that extending transitive closure algorithms to perform label computation is not straightforward, and we identify necessary and desirable properties of generalized transitive closure algorithms.

4.4.4.1 Necessary Properties — Completeness and Irredundancy

As pointed out in Section 2.4.1, the distribution of *BCON* over *BAGG* in a path algebra allows the aggregation to be moved in — instead of forming all paths first and then aggregating them, partial sets of paths are aggregated as they are generated. Once the aggregation has been moved in, paths are no longer explicitly enumerated, and therefore it must be ensured that each path between every source-destination pair has been considered (implicitly).

Similarly, it must be ensured that paths are not considered more than once. Redundant consideration of paths may lead to incorrect results for non-ordered path problems, where the *AGG* function is non-idempotent, such as bill of materials. Even for ordered problems, where *AGG* is idempotent, considering the same path multiple times results in unnecessary computation.

With these intuitions, we now formalize the notion of ''consideration'' and use it to define correctness criteria for generalized transitive closure computation.

Definition (Consideration Set)

Let t be a tuple generated by a transitive closure algorithm. The *consideration set* of t, denoted $CS(t)$, is a path-set defined recursively as follows:

1. If t is an arc in the original graph, then $CS(t) = t$.
2. If t was generated by concatenating tuples t_1 and t_2, then $CS(t) = CS(t_1) \circ CS(t_2)$.
3. If t was generated by aggregating tuples t_1 and t_2, then $CS(t) = CS(t_1) \cup CS(t_2)$.

The reader will note that the definition of the consideration set is in itself an ordered path problem corresponding to example A7 in Table 2.1.

Given a transitive closure algorithm *TC*, let $t(i,j)$ be the tuple computed by *TC* for nodes i and j. Let $\psi(i,j)$ denote the set of all simple paths between nodes i and j in graph G.

Definition (Completeness)

TC is a *complete* transitive closure algorithm if for every tuple $t(i,j) \in TC(G)$, $\psi(i,j) \subseteq CS(t(i,j))$

This definition guarantees that the final value obtained for each label $L(i,j)$ takes into account all possible paths from i to j that *it is supposed to.* (It must consider all simple paths. However, it may in addition consider some cyclic paths.)

Complete transitive closure algorithms have also been called *standard* algorithms in [UIY90]. Intuitively, a complete transitive closure algorithm never infers the existence of a path without actually constructing that path. An example of an algorithm that is not complete is the Schnorr algorithm [Sch78]. In this algorithm, a successor list is built for each node exactly until it includes half the nodes in the graph and no more. It is evident that this algorithm does not examine every path and hence it is not complete. Consequently, Schnorr's transitive closure algorithm cannot be generalized to perform path computations.

Algorithms that are based on fast matrix-multiplication (e.g. [CoW87]) are also incomplete. An important class of incomplete algorithms that we describe in Section 5.2.3 are those algorithms that determine reachability by collapsing strongly connected components and computing the closure of the resulting condensation graph.

Definition (Irredundancy)

TC is an *irredundant* transitive closure algorithm if for every tuple $t(i,j) \in TC(G)$, for every two paths from i to j $P_1(i,j)$, $P_2(i,j) \in CS(t(i,j))$, $P_1(i,j) \neq P_2(i,j)$.

This definition guarantees that the final value obtained for each label $L(i,j)$ does not take into account the same path multiple times.

An example of a redundant algorithm is the naive algorithm [AhU79, Ban85]. In this algorithm, in each iteration, every path so far discovered is joined with the original relation, rather than using the paths newly discovered as in semi-naive (see Section 5.1.1 below). In consequence, short paths that are discovered early are created again and again since the join that created them is repeated in each iteration.

4.4.4.2 Desirable Properties — Termination and Discrimination

As discussed in Section 1.4.2, not all path problems are well-defined for any graph, in the sense of being cycle-indifferent. For example, one cannot compute a bill-of-materials on a graph with cycles, or shortest path on a graph with negative cycles. It is desirable that a transitive closure algorithm handle such errors gracefully. At the very least, the algorithm should terminate. Ideally, though, the algorithms should not just terminate, but should detect the error and produce an error message[12]. This motivates the following definitions:

Definition (Termination)

A transitive closure algorithm that halts on any given path problem and input graph is called *terminating*.

Definition (Discrimination):

A transitive closure algorithm that detects when the specified path problem is not-well defined for the input graph is called *discriminating*.

One can view the above two properties as desirable or necessary depending on whether or not the input to the transitive closure algorithm is separately checked for inconsistency.

4.4.5 Maximizing Path Algebras

The case where the specified *CON* and *AGG* functions form a maximizing path algebra with an ordering $\preceq$ offers opportunity for more efficient evaluation. For each pair of nodes x and y, it is no longer necessary to consider all

12. Detecting that there is *some* cycle in a graph is not a hard problem. This problem should not be confused with the *np*-hard problem of performing a computation involving exactly *all* the acyclic paths in a graph that may contain cycles.

paths from x to y. Paths known to be suboptimal may be ignored, since the solution to the path problem depends only on the optimal paths between each pair of nodes (see Section 2.4.3). Hence, it is only necessary to consider at least one path between x and y that is optimal with respect to $\preceq$. This motivates a relaxation of the completeness requirement for transitive closure algorithms that may be used to evaluate a maximizing path problem.

Definition (Maximal Completeness)

TC is a *maximally complete* transitive closure algorithm if for every tuple $t(i,j) \in TC(G)$,
$\exists\, P_{\max}(i,j) \in CS(t(i,j))$ *s.t.* $\forall\, P(i,j) \in \psi(i,j)\ P(i,j) \preceq P_{\max}(i,j)$

This definition guarantees that the final value obtained for each label $L(i,j)$ takes into account at least one maximal path from i to j.

In particular, this definition allows us to look at the simple reachability problem in a new light (see Example A1 in Table 2.1). Reachability is a maximal path algebra where *every* path is maximizing. Therefore, reachability algorithms need only find *any* path between each pair of nodes to be maximally complete.

A Template Transitive Closure Algorithm for Maximizing Path Algebras

The template algorithm for path-set condensation, T_{psc} of Section 4.2.2, can be modified to only extend optimal paths by extending paths in *best-first* order. This is achieved by using *OPEN* as a priority queue (see e.g. [Knu73]) which stores path-sets according to the ordering $\preceq$. The insertion of new path-sets to *OPEN* in steps 2 and 5 will maintain the priority queue ordering. Each concatenation in step 4 will extend the best path-set condensation $\overline{\psi} \in OPEN$. We modify step 4 of Algorithm T_{psc} as follows.

```
4. Let $\overline{\psi}$ be the path-set condensation on the top of the OPEN priority queue.
   Extend $\overline{\psi}$ by concatenating to it some path-set condensation $\overline{\psi}_{extend} \in ALL$.
   ...
```

We refer to the modification of T_{psc} for maximizing path algebras as $T_{\max}$.

Theorem 4.13

Given a query Q and a path-set label L computed by path algebra A = (*D*, *BCON*, *BAGG*), if A is maximizing, then algorithm $T_{\max}$ computes the correct result for any input graph.

Proof

For every path-set $\psi(s,w)$, let the label computed for $\psi(s,w)$ at any point in algorithm T_{psc} be $L_{psc}(s,w)$. Let the correct label for $\psi(s,w)$ defined according to our model be $L(s,w) = AGG\ CON\ (\psi(s,w))$.

Because the path algebra is maximizing, there must exist at least one best path $P(s,w)$ such that the label of $P(s,w)$ is $L(s,w)$.

Lemma

If $P(s,w)$ is a best path from s to w with label $L(s,w)$, then at some point $\overline{\psi}(s,w)$ will be the best path-set condensation in *OPEN* chosen for expansion in step 4 of algorithm $T_{\max}$, with label $L_{psc}(s,w) = L(s,w)$.

We prove this Lemma by induction on the number of arcs in $P(s,w)$. If it is 1, then $P(s,w)$ is an arc $E(s,w)$. $E(s,w)$ is inserted into *OPEN* at step 2, at which point $\overline{\psi}(s,w)$ is set to $E(s,w)$, and $L_{psc}(s,w)$ is set to $L(s,w)$.

Otherwise, let the last hop in $P(s,w)$ be t, i.e., $P(s,w)$ is $P(s,...,t,w)$. By Bellman's principle of optimality [Bel58], the path $P(s,...,t)$ is a best path from s to t, and $L(s,w) = BCON(L(s,t),L(t,w))$. By the induction hypothesis, $\overline{\psi}(s,t)$ will be the best path-set condensation in *OPEN* chosen for expansion at some point, with label $L(s,t)$. Therefore, a path $P(s,w)$ would be established with label $L_{psc}(s,w) = L(s,w)$ unless an alternative path which is as good was already found.

The proof of the algorithm follows directly from the Lemma. ■

As an example, algorithm $T_{\max}$ can be used to compute the all-pairs shortest path problem. However, the algorithm is applicable also for computing the shortest path from a given set S of source nodes. As discussed in Section 4.3, such a selection is extension-monotonic, and will be applied in step 1 to restrict the initial path-set condensations in *OPEN*. Therefore, *OPEN* will only contain path-set condensations emanating from the source nodes in S. In particular, for the special case of single-source shortest path, algorithm $T_{\max}$ is essentially Dijkstra's algorithm [Dij59].

4.4.6 Path-Set Selection (δ)

Assume that associated with the label aggregation function *AGG* is a partitioning expression *Z*, and a path-set selection predicate δ is used to select the desired partitions. Let the aggregate (path-set) label attribute defined by *AGG* be *PSL*. Let the aggregated (path) label attribute associated with *AGG* be *Y*, and let the partitioning expression *Z* be a set of attributes $Z_1, \cdots, Z_p$ such that $\forall\ i\ Z_i \neq Y$. Let the set of paths being aggregated (i.e., those that satisfied the path selections specified in the query) be ψ', and let the path-set partitions according to the values of the composite attribute *Z* be $\psi'_1, \psi'_2, \cdots, \psi'_l$ (see Section 2.2). How should we optimize the evaluation of the δ predicate?

Let us assume that the δ predicate is a simple constraint on attribute *X*. By definition of δ in Section 2.2, $X \in \{PSL, Z_1, \cdots, Z_p\}$. We distinguish between two cases.

First consider the case where $X = Z_i$. Because in each partition ψ'_j the value of the Z_i attribute must be the same for any path $P_k \in \psi'_j$, the δ path-set selection can be pushed through the aggregation, and be applied as a path selection predicate σ on all paths in ψ'. That path selection may be optimized according to the selection transposition algorithms of Section 4.3.4.

Now consider the case where $X = PSL$. Let the aggregated attribute *Y* be defined by applying a label concatenation function *CON* over attribute *W*. What path selections can we derive on values of the path label *Y* for paths in ψ'?

Example A

Let *CON* be SUM, and assume that that the domain of attribute W is the non-negative reals. Let *AGG* be MAX, and the selection δ be $\text{MAX}(\text{SUM}(W)) < 1000$. As discussed in Section 4.3.3, the constraint $\text{MAX}(S) < 1000$ for a set S of positive reals is decomposable, and implies the constraint $\forall\ i\ S_i < 1000$. Therefore, we get the constraint $\forall\ P \in \psi'\ \ \text{SUM}(P) < 1000$. This constraint, in turn, is monotonically negative, and implies the constraint $\forall\ P \in \psi',\ \forall\ P^-\ \ \text{SUM}(P^-) < 1000$, allowing the elimination of subpaths that violate the constraint.

Example B

Consider Example A with *AGG* being MIN instead of MAX, and let the selection δ be $\text{MIN}(\text{SUM}(W)) < 1000$. The constraint $\text{MIN}(S) < 1000$ for a set S of positive reals is not decomposable, and does not in itself imply that $\forall\ i\ S_i < 1000$. Yet, the constraint $\forall\ P \in \psi'\ \text{SUM}(P) < 1000$ holds in this case as well, and may be utilized as in Example A. To see that this is the case, consider a path $P \in \psi'$ such that $\text{SUM}(P) \geq 1000$. If P is not the minimal path in its partition, it can be ignored, since we are only interested in minimal paths in each partition that satisfy the constraint. But if P is the minimal path in its partition, then it and all other paths in its partition violate the constraint, and can again be ignored. In either case, if $\text{SUM}(P) \geq 1000$, then P may be ignored.

If the δ predicate is a conjunction of constraints on attributes $X_1, \cdots, X_l$, we can optimize each constraint separately as outlined above. A constraint on a group-by attribute Z_i is pushed without modification and becomes a path selection predicate σ on values of attribute Z_i. A constraint on the path-set label *PSL* defined by applying an *AGG* function to attribute Y, may be used to derive a constraint on values of Y, and it is transformed to a path selection predicate σ on the *CON* function that computes the Y path label.

4.4.6.1 Path-Set Condensation in the Presence of Path Selection

Even if *CON* and *AGG* are independently computable, we may not be able to use path-set condensation if selections on the desired paths have been specified in the query. For example, assume we wish to find the shortest path between each pair of nodes in a graph such that the path has at most four arcs in it. Suppose at some point we find two paths between points s and t; $P(s,t)$ comprises three arcs and has label $L(s,t)$ and $P'(s,t)$ comprises two arcs and has label $L'(s,t)$, and $L(s,t) \leq L'(s,t)$. The situation is illustrated in Figure 4.8.

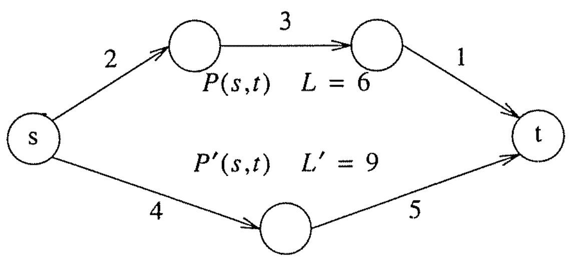


Figure 4.8. Two Paths from s to t

We could merge $P(s,t)$ and $P'(s,t)$ by aggregating their labels. This would result in a path-set condensation $\overline{\psi}(s,t)$ with a label of 6. However, it may turn out that some superpaths of $P'(s,t)$ satisfy the constraint on the number of arcs, while corresponding superpaths of $P(s,t)$ violate this constraint. In the graph of Figure 4.9, the path from w to z via $P'(s,t)$ should be in $\psi(w,z)$, since it comprises only 4 arcs, but the path from w to z via $P(s,t)$ should not be in $\psi(w,z)$ since it comprises 5 arcs. Therefore the path-set label of $\psi(w,z)$ should be 13 and not 10. But the path-set condensation $\overline{\psi}(s,t)$ cannot be used to compute the path label of $\psi(w,z)$, and the individual labels of $P(s,t)$ and $P'(s,t)$ are lost at this point.

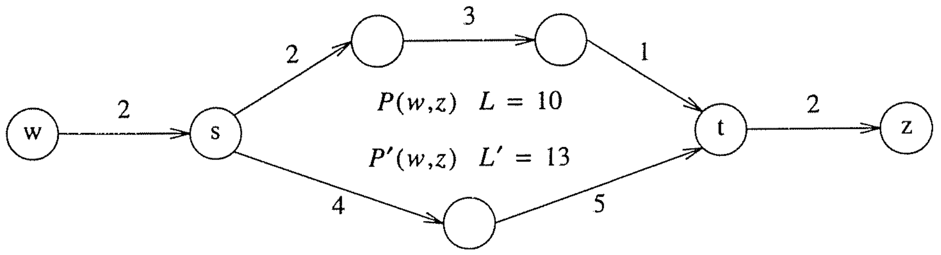


Figure 4.9. Two Paths from w to z

The example motivates the following theorem.

Theorem 4.14

Let Q be a query with a path-set label L and a selection criteria C on the desired paths. We can represent a path-set $\psi(s,t)$ by its condensation $\overline{\psi}(s,t) = <s,\ t,\ L(s,t)>$ iff for every two paths $P(w,s)$, $P(t,z)$, either

(1) $\forall\ P(s,t) \in \psi(s,t),\ P(w,s) \circ P(s,t) \circ P(t,z)$ violates C, or

(2) $\forall\ P(s,t) \in \psi(s,t),\ P(w,s) \circ P(s,t) \circ P(t,z)$ satisfies C.

Proof

Let $P^1(s,t)$ and $P^2(s,t)$ be two paths in $\psi(s,t)$, and let $P^1(w,z) = P(w,s) \circ P^1(s,t) \circ P(t,z)$, $P^2(w,z) = P(w,s) \circ P^2(s,t) \circ P(t,z)$. Assume $C(P^1(w,z)) = TRUE$ and $C(P^2(w,z)) = FALSE$. Then $\psi(w,z)$ should include $P^1(w,z)$ but not $P^2(w,z)$. If $\psi(s,t)$ was represented by $\overline{\psi}(s,t)$, then $L^1(s,t)$ and $L^2(s,t)$, the individual path labels of $P^1(s,t)$ and $P^2(s,t)$, are no longer available, so we cannot compute $L^1(w,z)$ and $L^2(w,z)$. Therefore, we cannot compute the path-set label for $\psi(w,z)$.

If (1) holds for a given pair of paths $P(w,s)$, $P(t,z)$, then $\psi(w,z)$ is empty. If (2) holds for a given pair of paths $P(w,s)$, $P(t,z)$, then $\psi(w,z)$ can be represented by
$\overline{\psi}(w,z) = <w,\ z,\ CON(L(w,s),\ L(s,t),\ L(t,z))>$. ■

Corollary

Path-set condensation can be used in the presence of selections on the source or destination nodes.

4.5 Related Work

In the context of generalized transitive closure, optimization ideas have been suggested, among others, in [LZH90, RHD86]. Rosenthal et al [RHD86] have proposed the pushing of monotonic constraints and showed how monotonicity constraints on functions can be identified. Luk et al [LZH90] considered algorithms in which a path and an arc are joined to form new paths (*sparse-standard* algorithms, in the terminology of [UlY90]). They identified different points in the closure computation where selections can be applied. Compared to these, our work is more general. We suggest optimizations for arc, path, and path set selections, as well as path and path-set label computations.

The idea of pushing selections into a recursive evaluation has received a lot of attention in the framework of Datalog optimization. Most of the work has been devoted to pushing *bindings* of variables to values. For example, in a Datalog program for transitive closure, bindings may correspond to equality selections on the endpoints, on all arcs/nodes, or on consecutive arcs in the graph. For least fixed point recursions, Aho and Ullman [AhU79] gave a rule for moving selections into the iterative process of evaluating the solution. The types of selections that can be pushed into a recursion have been characterized by Kifer and Lozinskii [KiL86], by Beeri et al [BKB90] and by Agrawal and Devanbu [AgD89]. A large body of knowledge was accumulated in recent years on general techniques for rewriting a collection of Datalog rules into an equivalent yet more efficient form. In the rewritten program, predicates are adorned, e.g. with "b or f" (bound or free) adornments, to indicate the binding of their arguments (see [RSS92, Ull89]).

More recent work has extended magic optimization techniques to handle conditions other than equality (see e.g. [BKM89, MFP90, SrRar]) and aggregate functions (see e.g. [GGZ91, MPR90, RoS92, SuR91]). We describe this work in the remainder of this section.

Let us first look at path enumeration in Datalog. As an example, consider program PB from Section 2.5, which computes paths in the graph and their associated length. We assume that there are no negative arcs in the graph.

```
PB1:  path(X,Y,C) :- arc(X,Y,C)

PB2:  path(X,Y,C) :- path(X,Z,C1), arc(Z,Y,C2), C = C1 + C2
```

Consider the query to find paths from node x no longer then 1000:

```
QB: path(x,?,C), C <= 1000
```

In the terminology of Section 4.3.3, the constraint on the value of C is *monotonically negative*. Hence, for this query, we would like to propagate the constraint on the variable C from the query rule QB to the program rules QB1 and QB2. Such an optimization requires the ability to propagate non-equality conditions on variables, e.g. `C <= 1000`, and the recognition of monotonicity of derivations, e.g. that a fact path(x,y,c) where $c > 1000$, cannot give rise, via rule QB2, to a new fact path(x',y',c'), such that $c' \leq 1000$.

Constraint propagation has been studied, among others, in [BKM89, MFP90, SrRar]. Balbin et al [BKM89] push conditions into recursive rules by adding a fold/unfold preprocessing phase to the magic set rewriting algorithm. Mumick et al [MFP90] introduce the "c" (condition) adornment class, and develop an extension of the magic templates algorithm [Ram88], called GMT (ground magic templates), which ensures that all rules are range-restricted. Srivastava and Ramakrishnan [SrRar] consider the propagation of arithmetic constraints, and develop a rewriting algorithm that computes the minimal set of facts necessary to answer the query given the constraints.

Monotonicity has beed studied in [GGZ91, MPR90, RoS92, Sri93, SuR91]. In [MPR90] a rule is said to be *monotonic* if adding new instantiations to any subgoal in its body (in particular one involving aggregation), can only generate new facts for the head if the rule, but cannot invalidate previous derivations. A program is monotonic if it comprises only monotonic rules. Ganguly et al. [GGZ91] generalized Datalog semantics for programs involving *min* or *max* aggregation by stipulating that each fact has an associated cost given by the value of the aggregated variable (in our terminology, its label). They define a class of *cost-monotonic min* or *max* programs and provide semantics for it by transforming each subgoal involving aggregation into an equivalent one using negation. Ross and Sagiv [RoS92] provide a more general definition of monotonicity that combines the monotonicity of [MPR90] with the cost-monotonicity of [GGZ91]: a rule is *monotonic* if adding new facts to a subgoal, or increasing the cost of previously derived facts, can only add new facts to the head of the rule, or increase the cost of previously derived facts. Srivastava [Sri93] develops a general model of *constraint projection* that subsumes most of the previous work on monotonicity and constraint propagation. For example, a constraint such as $\mathrm{MAX}(X_1,...,X_n) < 1000$ can be projected to dimension $n-1$ to derive the constraint $\mathrm{MAX}(X_1,...,X_{n-1}) < 1000$, or to dimension 1 to derive the constraint $\forall\ i\ X_i < 1000$. In the terminology of Section 4.3.3, the former derivation captures *monotonically negative* constraints, while the latter captures *decomposable* constraints.

Let us now look at path aggregation in Datalog. As an example, consider program PC from Section 2.5, which computes the all-pairs shortest path problem.

```
PC1:  path(X,Y,C)  :- arc(X,Y,C)

PC2:  path(X,Y,C)  :- path(X,Z,C1), arc(Z,Y,C2), C = C1 + C2

PC3:  s_path(X,Y,min(<C>))  :- path(X,Y,C)
```

The standard bottom-up evaluation [Ull89] of this program is similar to Algorithm 0 of Section 4.3.1 — it first constructs all the paths in the graph, and then computes the shortest paths between pairs of nodes. As Theorems 4.11 and 4.12 and the discussion of path algebras in Section 2.4 should make clear, the optimization of this query should take advantage of the distributive property of the shortest path problem, rewriting program PC into program PC' below.

```
PC'1:  path(X,Y,C)  :- arc(X,Y,C)

PC'2:  path(X,Y,C)  :- s_path(X,Z,C1), arc(Z,Y,C2), C = C1 + C2

PC'3:  s_path(X,Y,min(<C>))  :- path(X,Y,C)
```

Program `PC'` is no longer *stratified* [Ull89]: the evaluation of predicate `path` in rule `PC'2`, required for instantiating the set <C> and evaluating predicate `s_path` in rule `PC'3`, depends itself on the predicate `s_path`. Yet it is possible to give semantics to program `PC'` based on monotonicity, as we discussed above [GGZ91, RoS92, SuR91].

However, the idea behind the transformation from program PC to program PC' is applicable to more than just maximizing path problems. Consider program PD below for computing the bill of materials of an acyclic graph. assembly(x,y,q), contains(x,y,q), and bom(x,y,q) state that part *x* contains *q* *y* parts either directly, through one path of inheritance, or through multiple paths, respectively.

```
PD1:  contains(X,Y,Q)  :- assembly(X,Y,Q)

PD2:  contains(X,Y,Q)  :- contains(X,Z,Q1), assembly(Z,Y,Q2), Q = Q1 * Q2

PD3:  bom(X,Y,sum(<Q>)) :- contains(X,Y,Q)
```

Program PD can be transformed into program PD' below:

```
PD1':  contains(X,Y,Q)  :- assembly(X,Y,Q)

PD2':  contains(X,Y,Q)  :- bom(X,Z,Q1), assembly(Z,Y,Q2), Q = Q1 * Q2

PD3':  bom(X,Y,sum(<Q>)) :- contains(X,Y,Q)
```

Program PD' is not stratified. But, assuming that the underlying graph represented by relation assembly is acyclic, it is possible to give semantics to program PD' based on *modular stratification* (see e.g. [Gel92, Ros90, SSR93]) since the derivation of a specific fact for predicate contains will not depend on the same fact.

In addition to the observation that the logical semantics that have been suggested for programs PC' and PD' are based on different notions, we are not aware of an optimization technique that will perform a transformation similar to rewriting program PC into program PC' or program PD into program PD' for *all* path algebra problems.

Special optimizations for maximizing path algebras have been suggested in [CrN89, GGZ91, SuR91]. Cruz and Norvell [CrN89] gave an algorithm for maximizing semirings based on best-first expansion (see Section 4.4.5). Ganguly et al. [GGZ91] presented a *greedy fixpoint* algorithm that propagates facts in increasing order of their cost. Sudarshan and Ramakrishnan [SuR91] approached extremal aggregates using the notion of *relevance*. Essentially, facts that are non-maximal are not relevant (useful) to the evaluation of the query, and may be ignored. They presented techniques to derive constraints on predicates that avoid generating facts when ''better'' facts are known.

Finally, we consider path-set selection. Consider the following query on program PC (or PC').

```
QC: s_path(X,?,MIN_C), MIN_C <= 1000, X > 5
```

As we suggested in Section 4.4.6, constraints on a group-by variable can be pushed without change through the grouping subgoal. In this example, the constraint X > 5 can be pushed from rule PC3 to rule PC2 (see e.g [MPR90]). Constraints on the aggregate value may be used to derive constraints on the aggregated variable. In this example, the constraint MIN_C <= 1000 on rule PC3 may be used to derive the constraint C <= 1000 for rule PC2 (see [Sri93]).

4.6 Summary

We presented techniques for optimizing Generalized Transitive Closure queries. The following are the main results:

1. **Selections on the desired paths are applied during the closure computation, so that paths that are not in the result, and are not needed to compute the result, are pruned as early as possible.**

    - *Monotonically negative selections* can be used to control the generation of new paths from existing ones.
    - *Extension-monotonic selections* can be used to restrict the initial set of arcs that might be extended.
    - *Decomposable selections* can be used to restrict the initial set of arcs that should be copied from the input relation.
    - These optimizations can also be applied to constraints that are conjunctions of simple constraints.

2. **Paths and path-sets are represented in a condensed form. The condensed representation records the information that is necessary for the specified label computations and selections to be performed.**

    - *Incrementally computable path labels* can be computed using path condensation. Such path labels can be defined using an associative binary function. Alternatively, they can be defined using previously defined incrementally computable path labels. The computation of incrementally computable path labels with associated *arc selections* can also be done using path condensation.
    - *Incrementally computable path selections* can be evaluated using path condensation.
    - *Independently computable path-set labels* can be computed using path-set condensation. Such path-set labels can be defined using binary label concatenation and aggregation functions that form a path algebra (closed semiring). Alternatively, they can be defined using previously defined independently computable path-set labels.
    - Computation of path-set labels may require the use of ''non-trivial'' binary label concatenation and aggregation functions.
    - Maximizing path problems may be evaluated using ''best-first'' expansion strategy.
    - *Path-set* selection may be optimized by pushing it either through the grouping or the aggregation associated with the corresponding path-set label.
    - A path selection may interfere (at least partially) with path-set condensation. However, selections against the endpoints can be evaluated against path-set condensations.

These optimization techniques are independent of the specific algorithm that is used to evaluate the closure. The combined impact of the these methods is that the number of paths maintained during the closure computation and the storage requirement for each such path are both greatly reduced.

Chapter 5

A Survey of Transitive Closure Algorithms

In this chapter, we examine the algorithms suggested in the literature for evaluating transitive closure queries. We focus on algorithms developed in the database context, where I/O cost is at a premium. As a starting point, we describe reachability algorithms that compute the complete transitive closure (CTC) of the graph. We then describe how these algorithms may be adopted to compute the generalized transitive closure. We also discuss how the algorithms may be used to compute the most common type of selection, i.e. finding those nodes in the closure graph that are reachable from a specified set S of source nodes. Following [Jia90], we refer to this problem as *partial* transitive closure (PTC).

Given the plethora of transitive closure algorithms, we classify them into four families according to the data structure that underlies the computation in each algorithm. *Iterative* algorithms, which are based on a relational representation, are discussed in Section 5.1. *Matrix-based* algorithms are discussed in Section 5.2. *Graph-based* algorithms are discussed in Section 5.3. In Section 5.4 we discuss *hybrid* algorithms which combine ideas from more than one of the other families.

We should note that we have been involved in the design of the Direct algorithms, described in [ADJ90], the Single-Sided Composition algorithm, described in [ADJ89], and the Spanning Tree algorithm, described in [DaJ92]. However, for the sake of uniformity, we do not discuss these works separately; rather, we present them as part of an overall survey of transitive closure algorithms for databases.

Notation

Let the input graph be $G = (V,E)$. We denote $|V|$ the number of nodes in G, by n. We denote $|E|$ the number of arcs in G, by e. We denote the average branching factor (outdegree) of G by $b = \frac{e}{n}$. We say that G is *dense* if e is $O(n^2)$ and that G is *sparse* if if e is $O(n)$. We denote by d the length of the longest acyclic path in G.

A large number of transitive closure algorithms have been suggested in graph theory (see e.g., [Yan90]). Most of these algorithms implicitly assume that the transitive closure can be completely computed in memory. In the discussion below, we focus on the case where the computation does not fit in memory, and thus the I/O cost is a primary consideration. Let M be the size of the available memory, in units of number of tuples (arc facts), and let $k = \frac{M}{n}$ be the number of maximal successor lists that can fit in memory at once. We are interested in the case where $n < M < n^2$, i.e., $1 < k < n$. [13]

Considering the computation of partial transitive closure (PTC), let S be the set of source nodes specified in the query. If S contains exactly one node, we say that the PTC is *single-source*. Otherwise we say that the PTC is *multi-source*. Following [Jia90], we distinguish between *weak* and *strong* multi-source PTC computation. In a weak multi-source PTC query we wish to find all nodes in G reachable from some node in S, regardless of which particular node(s) in S they are reachable from. That is, we wish to compute the set

$\{ t \mid \exists\, s \in S \; s.t. \; (s,t) \in TC(G) \}$

In a strong multi-source PTC query we wish to find all nodes in G reachable from some node in S, as well as the particular node(s) in S they are reachable from. That is, we wish to compute the set

$\{ (s,t) \mid s \in S \wedge (s,t) \in TC(G) \}$

A weak multi-source PTC can be cast as a single-source PTC by adding a new ''dummy'' source node to the graph, connected with arcs to each of the nodes in S. Hence, in this Chapter and in Chapter 6 we consider the computation of strong multi-source PTC, and we omit the qualification ''strong multi-source'' for the sake of brevity.

5.1 Iterative Algorithms

In this section we examine the family of iterative algorithms, with the Semi-naive algorithm as the basis of our study.

5.1.1 The Semi-naive Algorithm

Let R_0 represent the source relation and R_f its transitive closure. In terms of the directed graph defined by R_0, the Semi-naive algorithm [BaR87, Ban85, GKB87], starting with paths of length one (i.e., length in number of arcs in the path), finds in each iteration new nodes that can be reached through paths that are one longer than in the previous iteration:

```
$R_f \leftarrow R_0$;
$R_\Delta \leftarrow R_0$;
while ($R_\Delta \neq \phi$) {
        $R_\Delta \leftarrow R_\Delta \cdot R_0 - R_f$;
        $R_f \leftarrow R_f \cup R_\Delta$;
}
```

The symbol $\cdot$ indicates *natural composition* i.e., a natural join followed by a projection on the non-join attributes. Let $R(A,B)$ and $S(B,C)$ be two binary relations. Then

$$R \cdot S \equiv \pi_{A,C}\,(R \bowtie_{R.B=S.B} S)$$

13. If $M \leq n$, we may not be able to fit even one successor list into memory, while if $M \geq n^2$, the computation of the transitive closure can be done completely in memory.

In the worst case, the Semi-naive algorithm requires d iterations, and therefore its complexity is $O(n^3 d)$ [CrN89].

Several variations on this basic iterative algorithm have been proposed in the literature [ADJ89, GKB87, Ioa86, Lu87, VaB86], and are described in the following subsections.

5.1.2 The Logarithmic Algorithm

Valduriez and Boral [VaB86] and Ioannidis [Ioa86] independently suggested a logarithmic algorithm for transitive closure. The algorithm requires just $\sqrt{d}+1$ iterations, but the relations processed at each iteration are larger. Ioannidis [Ioa86] developed algebraically a whole family of logarithmic algorithms, and suggested that the one based on powers of three, the so-called *Smart* algorithm, may be the best one. However, several performance studies indicate that the Semi-naive algorithm usually outperforms the logarithmic algorithms (see for example [KIC92]). In addition, the Semi-naive algorithm can handle selection on source nodes, which the logarithmic algorithms cannot handle efficiently.

5.1.3 Lu's Algorithm

Lu's HYBRIDTC algorithm [Lu87] is essentially an implementation of the Semi-naive algorithm based on hash-join [DKO84]. The algorithm employs two optimizations: (a) at the end of each iteration, tuples in R_0 that will not be needed later on are deleted, and (b) new tuples that fall into the current hash bucket are processed immediately. Lu showed that for some graphs the HYBRIDTC algorithm performs better than the Semi-naive and Smart algorithms. The performance study involved trees only, and duplicate elimination was not considered.

5.1.4 Single-Sided Composition

Iterative algorithms are essentially a sequence of composition operations Each composition operation comprises 3 steps: i) join, ii) projection, and iii) duplicate elimination. Some systems (System R [Moh88] and Ingres [Sto88], for example) combine the first two steps and do projection at the same time as the join is being computed, but all the systems that we know of make a separate pass for removing duplicates. In [ADJ89], Agrawal et al proposed an alternative composition method called *single-sided* composition, which performs join, projection, and duplicate elimination as one unified operation. The essential idea is as follows.

Let $R(A,B)$ and $S(A,B)$ be the two relations to be composed. With the usual hash join, R is hashed into buckets $R_1 \cdots R_n$ based on the $R.B$ attribute value, and S is based into buckets $S_1 \cdots S_n$ based on the $S.A$ attribute value. Let T be the relation holding the join result. Each R_i bucket need only be joined with the corresponding S_i bucket, but the resulting tuples may have arbitrary values for their $T.A$ and $T.B$ attributes. Hence, elimination of duplicates from T can be very expensive. [14]

14. This observation applies to Sort-Merge join as well, where R and S are sorted on their joining attributes ($R.B$ and $S.A$), and then merged using a scan.

With single-sided composition, R is hashed into buckets $R_1 \cdots R_n$ based on the $R.A$ attribute value. S is not partitioned into buckets. The join result relation, T, is also partitioned into buckets $T_i \cdots T_n$. For each bucket R_i, S is read into memory one page at a time. The tuples in R_i are matched with the tuples in the page of S, and the resulting tuples are output. The new tuples are guaranteed to fall into the T_i bucket, and therefore duplicate elimination can be performed in memory.

The tradeoff between the higher join cost and lower duplicate elimination cost is studied in [ADJ89]. Single-sided composition turns out to perform better than standard composition methods when the relations to be joined are such that the join result is many times larger than the source relations and many duplicates must be eliminated after projection over the non-join attributes. This is typically the case when computing transitive closures of dense graphs.

An implementation of Semi-naive based on single-sided composition was found to be better than the standard implementation for most input graphs in [ADJ90]. Implementations of the Semi-naive and logarithmic (Smart) algorithms, based on hash-join and single-sided compositions, were also explored in [KIC92]. They found the hash-join implementations to perform generally better, though the single sided composition method employed was quite different — duplicate elimination was done at the end of each iteration rather than being done on the fly. It is difficult to speculate regarding the performance tradeoff of these two schemes.

Computation of Semi-naive using single-sided composition can take advantage of the fact that result tuples are inserted into a bucket that is currently in memory. These result tuples can be processed in the same iteration, instead of writing them to disk and then reading them again in the next iteration. This sort of ''pushy'' [GKB87] or ''immediate'' processing has been explored in [KIC92]. It was shown to improve the performance of the Semi-naive and logarithmic algorithms by reducing the number of iterations before convergence.

5.1.5 Partial Transitive Closure

The Semi-naive algorithms can be adapted to handle selections on a set S of source nodes in by changing the initialization phase as follows:

```
$R_f \leftarrow \{t \in R_0 | t.src \in S\};$
$R_\Delta \leftarrow \{t \in R_0 | t.src \in S\};$
```

The resulting Semi-naive evaluation has also been called the δ-Wavefront algorithm [QHK89].

5.1.6 Generalized Transitive Closure

To perform label computations, the Semi-naive algorithm can be generalized as follows [ADJ88b, CrN89]:

```
$R_f$ ← $R_0$;                              (* 1 *)
$R_\Delta$ ← $R_0$;
while ($R_f$ changes) {
        $R_\Delta$ ← $AGG(CON(R_\Delta,R_0))$;
        $R_f$ ← $AGG(R_f,R_\Delta)$;
}
```

When the path problem is ordered, we can ignore a newly generated path when a better path has already been established and write [ADJ88b, CrN89]:

```
$R_f$ ← $R_0$;                              (* 2 *)
$R_\Delta$ ← $R_0$;
while ($R_f$ changes) {
        $R_\Delta$ ← $AGG(CON(R_\Delta,R_0))$   -   $R_f$;
        $R_f$ ← $AGG(R_f,R_\Delta)$;
}
```

Where the subtraction – is defined by

$$S-T = \{(x,y,L_S) \in S \text{ s.t. } \forall\ (x,y,L_T) \in T\ L_T \preceq L_S\}$$

A generalization, similar to (* 1 *), of the logarithmic algorithm [Ioa86, VaB86] is given in [CrN89].

5.1.7 Completeness

The Semi-naive algorithm, as generalized in `(* 1 *)`, is complete. Each arc in R_0 represents a path of length 1 and is included in R_f. All possible paths of length k are generated in iteration $k-1$ and aggregated into R_f. Similarly, the generalization of the Semi-naive algorithm in (* 2 *) is maximally complete since only suboptimal paths are discarded.

5.1.8 Irredundancy

In each iteration, the Semi-naive algorithm creates paths that are one longer than in the previous iteration, always adding one arc at the end (or beginning, depending upon the join order of R_0 and R_Δ) of a previously created path. Since, for each path, the last arc is unique and the path obtained by removing the last arc is unique, each path is generated only once. This irredundant property has been proved for the Semi-naive algorithm in [BaR] under the name of optimality.

5.1.9 Error Handling

Given an ill-defined problem, such as a bill-of-materials query on a cyclic graph or a shortest path query on a graph with negative weight cycles, the Semi-naive algorithm (and other iterative algorithms) may not terminate. However, through some additional checking the iterative algorithms can be rendered terminating as well as discriminating [ADJ88a, KIC92]. If there is a (negative weight) cycle in a graph, than for every node on this cycle there must exist a non-null (negative weight) path to itself. Thus, in an iterative algorithm, after creating or updating a tuple, we can check to see whether the tuple indicates a cycle. If such a tuple is generated at any stage in the algorithm, the algorithm can halt and produce an error message.

5.2 Matrix-Based Algorithms

In this section, we study algorithms based on the adjacency (incidence) matrix representation, using Warshall's algorithm as the basis for our study.

Definition (Adjacency Matrix)

The *adjacency matrix* of a n-node graph G is an $n \times n$ boolean matrix with element a_{ij} being 1 if there is an arc from node i to node j, and 0 otherwise.

5.2.1 Warshall's Algorithm

Given an initial $n \times n$ adjacency matrix, Warshall's algorithm [War62] computes the transitive closure of the matrix as follows:

```
For $k$=1 to $n$
        For $i$=1 to $n$
                For $j$=1 to $n$
                        $a_{ij} = a_{ij} \vee (a_{ik} \wedge a_{kj})$;
```

In successor list terms, one can understand the algorithm as follows:

```
For every node $k$
        For every predecessor $i$ of $k$
                For every successor $j$ of $k$
                        Make $j$ a successor of $i$
```

The complexity of Warshall's Algorithm is $O(k^3)$ [War62]. A straightforward implementation of Warshall's algorithm may proceed as follows: For each node k, first fetch its successor list. Then for each predecessor i of k, fetch the successor list of i, and add to the successor list of i the successor list of k (removing duplicates if any). Unfortunately, to determine the predecessors of k, the successor list of all other nodes may need to be scanned to see if k appears in them. An alternative would be to maintain with each node a predecessor list as well. Determination of the predecessors of k would now become trivial, but at the time of updating the successor list of a predecessor i to include in it the successors of k, the predecessor lists associated with the successors of k would also have to be updated to include i.

If the transitive closure cannot fit in main memory all at once, such a ''naive'' implementation of Warshall's algorithm would incur a high I/O penalty. This observation motivated the design of variations on Warshall's algorithm that reduce its I/O requirement by improving the locality of reference. The following subsections describe these variations.

5.2.2 Warren's Algorithm

Warren [War75] observed that processing the matrix elements row-wise is desirable if the matrix is stored as bit vectors formed row-wise with each matrix element represented as a bit, and proposed the following modification to

Warshall's algorithm:

$$\forall_{i=1}^{n} \ \forall_{k=1}^{i-1} \ \forall_{j=1}^{n} \ a_{ij} = a_{ij} \vee (a_{ik} \wedge a_{kj})$$
$$\forall_{i=1}^{n} \ \forall_{k=i+1}^{n} \ \forall_{j=1}^{n} \ a_{ij} = a_{ij} \vee (a_{ik} \wedge a_{kj})$$

Warren's algorithm iterates over the adjacency matrix in two separate passes, first processing the lower diagonal half and then the upper diagonal half. As a result, it only requires successor lists — there is no need to maintain predecessor lists as well. Yet, the locality of reference of the algorithm is still poor. A straightforward implementation of Warren's algorithm in [LM87] found that the algorithm did well (compared to iterative algorithms) only when the memory size was not much smaller than the final transitive closure size.

5.2.3 Warshall-Derived Algorithms

In [ADJ90, AgJ87] the following observation was made. The order in which matrix elements are processed in Warshall's algorithm may be modified, provided that the following two constraints are satisfied:

1. For all i,j,k, processing of element (i,k) precedes processing of element (i,j) iff $k<j$.

2. For all i,j,k, processing of element (j,k) precedes processing of element (i,j) iff $k<j$.

Algorithms that maintain these constraints are called *Warshall-derived* algorithms. As an example, it is easy to verify that Warren's algorithm is a Warshall-derived algorithm.

The correctness of the Warshall-derived algorithms is established by the following lemma.

Lemma

Every Warshall-derived algorithm computes the same result as Warshall's algorithm.

Proof

Consider an element (l,k) that is processed before element (i,j) in the Warshall algorithm. One may switch the order, and process (i,j) before (l,k), without affecting the result unless either the successor list of i or j is affected by the processing of (l,k) or the successor list of k or l is affected by the processing of (i,j). But such a problem can arise if and only if $l=i$, $l=j$, or $k=i$.

We can therefore write the following three constraints:

1. Two elements of the form (i,k), (i,j) must be processed in the same order as before, i.e. (i,k) before (i,j) iff $k<j$.

2. Two elements of the form (j,k), (i,j) must be processed in the same order as before, i.e. (j,k) before (i,j) iff $k<j$.[15]

3. Two elements of the form (l,i), (i,j) must be processed in the same order as before, i.e. (l,i) before (i,j) iff $i<j$.

The third constraint can be rewritten: (i,j) before (l,i) iff $j<i$. But this constraint is the same as constraint 2 with appropriate substitution of variables. Therefore we need impose only the first two constraints. ■

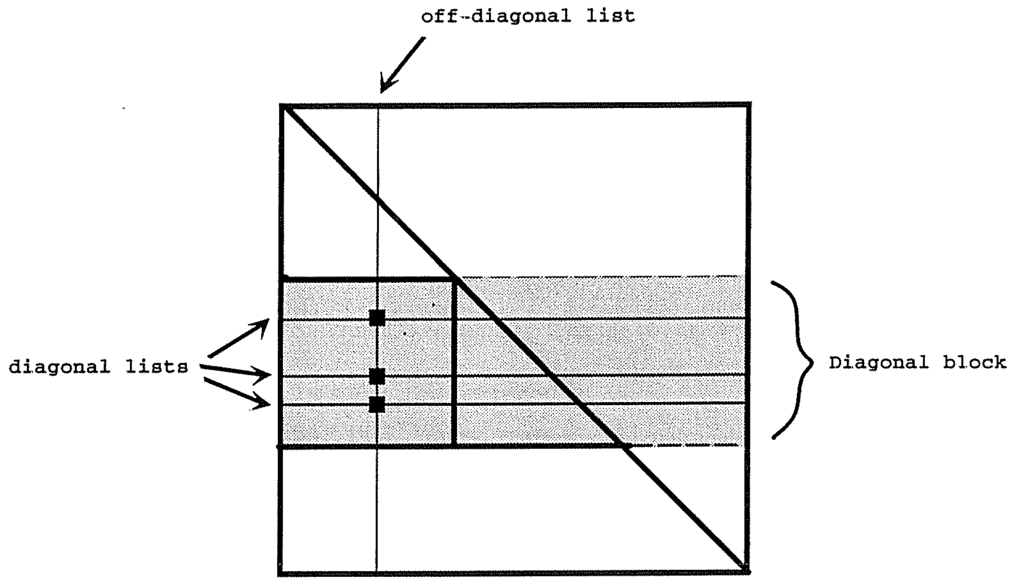


Figure 5.1. Blocking

Two Warshall-derived algorithms are presented in [ADJ90]. These are variations of the Warshall and Warren algorithms that use *blocking* to improve locality and reduce the I/O cost of computing the transitive closure. The basic idea is illustrated in Figure 5.1. In the blocking algorithms, successor lists are read into memory and expanded one block at a time. The block of successor lists in memory, shaded in figure 5.1, is called the *diagonal block.* A successor list is called a *diagonal list* if it is part of the diagonal block, and an *off-diagonal list* otherwise. The union of two diagonal lists requires no I/O since both lists are in memory. On the other hand, the union of a diagonal list with an off-diagonal list requires I/O to bring the off-diagonal list into memory. However, when an off-diagonal list j is brought into memory (displacing a previous off-diagonal list), it is joined with all diagonal lists i_k such that j is a successor of i_k. Figure 5.1 shows schematically an off-diagonal list that may be unioned with three diagonal lists. Thus, blocking allows several successor list unions to be performed at the cost of a single I/O.

The performance studies in [ADJ90, KIC92] showed that the blocking algorithms generally performed much better than the iterative algorithms for the computation of the complete closure over a wide range of input graphs.

15. The case where $j=k$ is not important since processing of (j,j) does not affect anything. As such, we have assumed $j \neq k$ in writing this constraint.

5.2.4 Partial Transitive Closure

Matrix-based algorithms cannot handle selections on the source nodes efficiently [ADJ90, KIC92] since paths are extended at both ends. A comparison of the Blocked Warren algorithm with the Seminaive algorithm for PTC computation in [KIC92] demonstrated that the performance of the Seminaive algorithm improved relative to Blocked Warren as the percentage of source nodes was decreased (i.e. the selectivity of the query was increased). Furthermore, the crossover point at which the Seminaive algorithm performed better than the Blocked Warren algorithm was above 1/3 for all the graph studied.

5.2.5 Generalized Transitive Closure

To perform label computations, the innermost step of Warshall's algorithm can be rewritten as:

$$L_{ij} = AGG(L_{ij}, CON(L_{ik}, L_{kj}))$$

With a different notation, this is Kleene's algorithm for generalized transitive closure [Kle56]. Floyd's shortest path algorithm [Flo62] is a special case of the above generalization of Warshall's algorithm for the shortest path problem, with CON being *addition* and AGG being the *min* function.

Warshall-derived algorithms can be generalized in a similar fashion. A generalization of Warren's algorithm (in the relational form of [LM87]) is given in [CrN89], and generalizations of the blocking algorithms are given in [ADJ90].

5.2.6 Completeness

We establish the completeness of Warshall's algorithm using the sufficient condition given in the lemma below.

Definition (Order-Invariant)

A transitive closure algorithm is *order-invariant* if for every n node graph the computation order is the same, independent of the existence (or otherwise) of arcs between nodes in the graph. By computation order we mean the order in which an attempt is made to induce the existence of a path $P(i,j)$ by concatenating paths $P(i,k)$ and $P(k,j)$ if they exist. Note that the order is defined on the *attempt* made, and not on the actual creation of a tuple (i,j), the timing of which could vary depending on which intermediate node resulted in this path.

Lemma

If a transitive closure algorithm is order-invariant then it is complete.

Proof

Let TC be an order-invariant transitive closure algorithm. We will show by contradiction that if TC is incomplete that it is not a correct transitive closure algorithm, that is, that there exists a graph on which the application of algorithm TC will fail to compute the closure. Let G be a graph with multiple acyclic paths $P_1(i,j), \ldots, P_n(i,j)$ between nodes i and j. Assume that algorithm TC generated the transitive closure of G, and the tuple (i,j) in this transitive closure was generated without ever considering path P_m.

Generate a graph G' by deleting from G all arcs participating in $\{P_i\}$ $i = 1, \ldots, m-1, m+1, \ldots, n$ that do not also participate in P_m. Then execute algorithm TC on G'. Since the order of computation is the same, the computation should not consider any path in G' that was not considered in G. Therefore tuple (i, j) will not be generated. ■

In Section 4.4.4.1 we have given the Schnorr algorithm [Sch78] as an example of an algorithm that is not complete. Since the stopping condition in this algorithm depends on the number of successors found for a particular node, we see that the algorithm is also not order-invariant.

Theorem 5.1

Warshall's algorithm is complete.

Proof

Warshall's algorithm is order-invariant. Apply above Lemma.

5.2.7 Irredundancy

Before showing that Warshall's algorithm is irredundant, we give a few definitions.

Definition (Processing an Element)

Processing an element (i,j) is the procedure of determining whether an arc (i,j) exists, and if it does, making all successors of j into successors of i by generating (and aggregating) arcs from i to each one of them.

Definition (The Touch-Once Property)

A transitive closure algorithm has the *touch-once* property if for every two nodes, i and j, element (i,j) is only processed once.

Definition (Generator)

If a path $P(i,j)$ is obtained through the concatenation of $P(i,k)$ and $P(k,j)$, then node k is called the *generator* of the path $P(i,j)$.

If a transitive closure algorithm has the touch-once property, then it generates no tuple twice with the same generator.[16] However, the same path could be found more than once with different generators.

Theorem 5.2

Warshall's algorithm is irredundant.

16. The touch-once property corresponds to *duplicate free* execution in [Ban85].

Proof

Suppose that Warshall's algorithm generates the same path $P(i,j)$ twice. Assume k and l were the two generators. Clearly Warshall's algorithm has the touch-once property since each matrix element is processed exactly once. Therefore it must be the case that $k \neq l$. Assume, without loss of generality, that k is a predecessor of l, so that the nodes appear in the order:

i ----> k ----> l ----> j

The path $P(i,j)$ would be generated as follows:

1. Through concatenating (i,k) with (k,j) when (i,k) is processed, and
2. Through concatenating (i,l) with (l,j) when (i,l) is processed.

Assume (1) occurred before (2); then, the existence of the path from k to j through l implies that column l has already been processed. Therefore, it is impossible that element (i,l) will be processed later on. Assume (2) occurred before (1); then, the existence of the path from i to l through k implies that column k has already been processed. Therefore, it is impossible that element (i,k) will be processed later on. ■

Theorem 5.3

Every Warshall-derived algorithm is both complete and irredundant.

Proof

The results computed by a Warshall-derived algorithm are identical to those computed by Warshall, irrespective of the AGG and CON functions specified or the underlying relation. Since Warshall is complete and irredundant, every Warshall-derived algorithm must also be complete and irredundant. ■

5.2.8 Error Handling

Matrix-based algorithms always come to a halt, even if the path problem is not well-posed, since every matrix element is processed exactly once. Yet,, the ''answer'' computed for an ill-defined path problem is meaningless, and in fact could be different for different processing orders of the nodes. Therefore, matrix-based algorithms are terminating but are not discriminating.

Matrix-based algorithms can be made discriminating, too, at the cost of some additional processing, as in the case of iterative algorithms. The idea is to check the diagonal elements of the incidence matrix, which reflect paths from nodes to themselves, to detect the occurrence of a cycle, a negative-weight cycle, etc. This additional processing can be done as a post-processing step after the algorithm has terminated, or it can be done on-the-fly as the transitive closure is being computed.

5.2.9 The Grid Algorithm

Ullman and Yannakakis proposed an algorithm for dense graphs that processes the adjacency matrix in squares rather than stripes [UlY90]. The algorithm, referred to as the *Grid algorithm* in [AgJ90], proceeds as follows:

Partition the matrix into $f \times f$ square sub-matrices. The size of each submatrix, $\frac{n}{f}$, is the maximal size that fits into the available memory, i.e., $\frac{n}{f} = \sqrt{M} = \sqrt{\frac{n}{k}}$, and therefore we set $f = \frac{n}{\sqrt{M}} = \sqrt{n \times k}$. Let $M^*_{i,j}$ be the reflexive and transitive closure of submatrix $M_{i,j}$. The Grid algorithm proceeds as follows:

```
For $k = 1$ to $f$
    $M_{k,k} = M^*_{k,k}$ ;
    for $i = 1$ to $f$
        for $j = 1$ to $f$
            $M_{i,j} = M_{i,j} + M_{i,k} \times M_{k,k} \times M_{k,j}$
```

Ullman and Yannakakis show that, if the graph is dense, the closure computation incurs less I/O when the adjacency matrix is divided into squares than when it is divided into stripes, as in the blocking algorithms of [ADJ90].

However, with the Grid algorithm, the block sizes must all be equal, and dynamic block sizing is difficult [AgJ90]. The determination of the size of each submatrix is conservative, and unless the submatrices do fill up, memory is underutilized. The only performance study of this algorithm, conducted in [AgJ90], found it to be inferior to the Hybrid algorithm.

5.3 Graph-Based Algorithms

In this section, we study graph-based algorithms using the Purdom algorithm as the basis for our study.

Definition (Strongly Connected Component)

Given a graph *G*, let the relation *SC* (strongly connected) be defined by $s\ SC\ t$ if both *s* is a successor of *t* and *t* is a successor of *s* in *G*, i.e., $(s,t) \in TC(G)$ and $(t,s) \in TC(G)$. *SC* is reflexive, symmetric and transitive, hence it is an equivalence relation. Each of the equivalence classes (subgraphs) of *G* is called a *strongly connected component* (SCC). For a node *s*, the set of all nodes t_i such that $s\ SC\ t_i$ is denoted by $SCC(s)$.

Definition (Condensation Graph)

Given a graph *G*, the *condensation graph* of *G* is a graph G_C such that for every strongly connected component *SCC* in *G* there is a node in G_C, and for every arc $(x,y) \in G$ such that $SCC(x) \neq SCC(y)$ there is an arc $(SCC(x), SCC(y)) \in G_C$.

Definition (Topological Order)

Given a directed acyclic graph *G*, a *topological order* on the nodes of *G* is an ordering $\prec_t$, such that if (s, t) in *G*, then $s \prec_t t$.

5.3.1 The Purdom Algorithm

Purdom, in [Pur70], made two key observations:

1. During the computation of transitive closure of a directed acyclic graph, if node $s \prec_t t$, then additions to the successor list of node s cannot affect the successor list of node t. One should therefore compute the successor list of t first and then that of s. By processing nodes in reverse topological order, one need add to a node only the successor lists of its immediate successors since the latter would already have been fully expanded. We call this idea the *immediate successor* optimization [AgJ90].

2. All nodes within a strongly connected component (SCC) in a graph have identical reachability properties, and the condensation graph obtained by collapsing all the nodes in each strongly connected component into a single node is acyclic.

Weaving these ideas together, Purdom proposed a four-phase algorithm:

i. Let the original graph be G. Compute the strongly connected components in G, and collapse each one into a single node. Let the resulting acyclic condensation graph be G_C.

ii. Obtain a topological sort on G_C.

iii. Compute the transitive closure of G_C by expanding successor lists in reverse topological order.

iv. Compute the successor lists of the nodes in the original graph G from the successor lists of their respective SCCs: a node y is a successor of node x in G iff $SCC(y) = SCC(x)$ or SCC(y) is a successor of $SCC(x)$ in G_C.

The complexity of Purdom's algorithm is $O(n \times b \times n) = O(ne)$ [Yan90], since for each node, we process on average b arcs leading to its immediate successors, and each one of these may have $O(n)$ successors.

Tarjan [Tar72] independently developed an efficient algorithm for determining strongly connected components of a graph by means of a depth-first search. This algorithm also produces as a by-product a topological sort on the components. Thus it can be used to perform phases 1 and 2 of Purdom's algorithm.

Embellishments on this basic algorithm have been suggested, among others, in [Ebe81, EvK77, GoK79, IRW, Jia90, Sch83] and are described in the following subsections.

5.3.2 The Eve and Kurki-Suonio Algorithm

Eve and Kurki-Suonio [EvK77] observed that it was possible to modify Tarjan's algorithm so that the successor lists are also expanded as the strongly connected components were being determined, in effect combining Purdom's phases 1-2 with 3-4 and producing a single phase algorithm. When an SCC is identified, the successor list of the root of this SCC is built as a union of the (partial) successor lists of the members of the SCC. When the successor list of the root is fully expanded, it can be distributed to all the members of the SCC.

5.3.3 The Ebert Algorithm

Ebert [Ebe81] observed that sometimes there is no need to add to a node i the successors of its child j. Consider the graph in Figure 5.2.

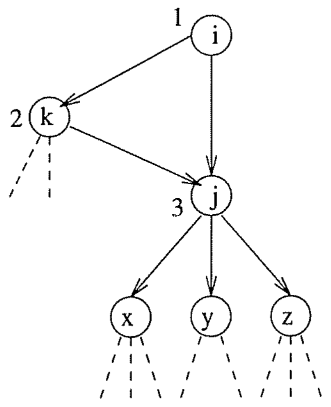


Figure 5.2. The Marking Optimization

If node k is visited before node j, then after the union of S_k with S_i, j and its successors are already in S_i. The union of S_j with S_i is therefore redundant, and Ebert's algorithm avoids this union. This optimization was also utilized in [Sch83] and in several algorithms in [AgJ90, IRW]. The latter algorithms mark those immediate successors of a node i that should not be processed, and we therefore refer to this technique as the *marking optimization* [AgJ90]. [17]

5.3.4 The Goralcikova-Koubek Algorithm

As Figure 5.2 demonstrates, whether the marking optimization may be applied to a particular arc from a node to its child, such as (i,j), depends on the relative order in which the children of the node (e.g., j and k) are visited. However, if the nodes are first topologically sorted, and if the children of a node i are visited in (forward) topological order, then the marking optimization always applies. For example, in Figure 5.2, $i \prec_t j \prec_t k$, as indicated by the topological sort rank next to each node. Therefore, when expanding node i, node k will be considered before node j. This is the basis of the reduct and closure algorithm in [GoK79]. As shown in [GoK79, Jia90], the marking optimization on a topologically sorted graph is equivalent to a transitive reduction of the input graph [AHU75]. In other words, if we remove from the original graph all marked arcs, we end up with the minimal graph that has an identical closure to the original graph. The combination of topological sort and the marking optimization was used, among others, in the Hybrid algorithm [AgJ90] and the BTC algorithm [IRW]. Both algorithms demonstrated that the redundant work avoided by the marking optimization far outweighed the extra cost of the topological sort.

5.3.5 The Schmitz Algorithm

Schmitz [Sch83] noticed that the generation of (partial) successor lists for nodes other than the root node in an SCC is not necessary. He proposed an algorithm that only generates one successor list per SCC, and delays the

17. In general, when a successor list S_j is added to another list S_i where i is a predecessor of j, the list S_j may or may not be complete. In [IRW] such a list union is called a *closed addition* if S_j is complete, and an *open addition* otherwise. Ioannidis et al considered algorithms that use either closed or open additions, and they employed the marking optimization to reduce the cost of both types of algorithms. Our definition of marking is more restrictive than that of [IRW] since we study algorithms that are based on Purdom's algorithm and hence use closed additions only.

generation of that list until the SCC is identified. The SCC successor list contains the nodes of the SCC and the successors of children of the nodes of the SCC that are not members of the same SCC. This optimization was also utilized in [Jia90] and in several algorithms in [IRW], where pointer manipulation was used to generate the successor list of the root of the SCC by merging partial successor lists of members of the SCC. We call this technique the *root optimization* [IRW].

5.3.6 The BTC Algorithm

Ioannidis et al [IRW, Win92] proposed a DFS algorithm called BTC (Basic Transitive Closure) that incorporates most of the ideas presented in the previous subsections. In addition, they developed several implementation techniques to improve the efficiency of storing and expanding successor lists. These techniques include inter- and intra-successor list clustering and page and list replacement policies. We describe these techniques in detail in Chapter 6.

5.3.7 Partial Transitive Closure

Graph-based algorithms can be adapted to handle selections on a set S of source nodes in a straightforward way by searching only the portion of the graph that is relevant to the selection.

Definition (Magic Graph)

Given a graph G and a set S of source nodes in G, the *magic* subgraph of G with respect to S, denoted by G^S, comprises the nodes and arcs in G reachable from some node $s \in S$. [18]

As an example, consider the DAG in Figure 5.3 (a) and the set of source nodes, marked with a circle, $S = \{a,b,e\}$. The magic subgraph for this selection is shown in Figure 5.3 (b).

We can use Purdom's algorithm to compute the partial transitive closure of a graph G with respect to a set S of source nodes by modifying the first and last phases of the algorithm as follows:

i. Compute the strongly connected components in the *magic graph* G^S, and collapse each into a single node. Let the resulting acyclic condensation graph be G^S_C.

iv. Compute the successor lists of the nodes *in* S from the successor lists of their respective SCCs: a node y is a successor of node $s \in S$ iff $SCC(y) = SCC(s)$ or SCC(y) is a successor of $SCC(s)$ in G^S_C.

To see that this modification is correct, observe that that, from the definition of an SCC, either all nodes in an SCC are reachable from some source node $s \in S$, or none of them is. As before, we can use Tarjan's algorithm to compute the reachable strongly connected components of the graph by intializing it with the nodes in S.

18. We refer to this graph as the *magic graph* [Jak92] since it includes exactly the arcs (facts) that would be deduced by magic optimization of the corresponding selection query (see e.g. [Ull89]).

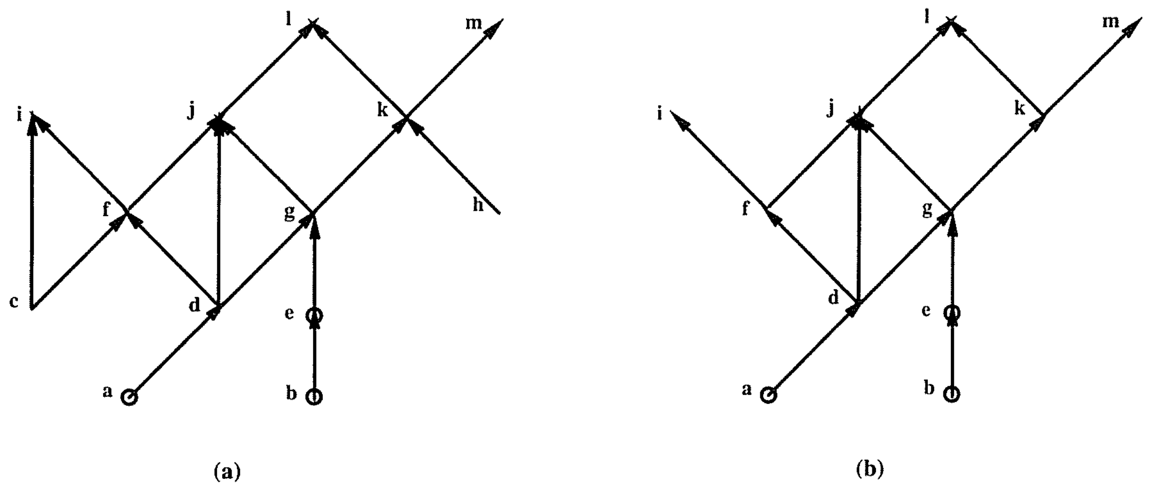


Figure 5.3. Magic (Reachable) Graph

Graph algorithms that compute single-source or multi-source partial transtive closure have been presented in [IRW, Jak92, Jia90].

5.3.8 The Jiang Algorithm

Jiang [Jia90] observed that for multi-source queries, a further reduction of the input graph is possible. If a node *j* has a single parent *i* and *j* is not in *S*, then there is no need to compute the successor list of *j*. (In [Jia90] *j* is called a *single-input* node, but we use the term *single-parent* instead.) Instead, node *j* can be *reduced* to a sink node by making its children (immediate successors) into children of *i* and deleting the outgoing arcs of node *j*. We say that the children of the single-parent node *j* are *adopted* by its parent node *i*, and we call this technique the *single-parent optimization*. Considering the graph of Figure 5.3 (a), we see that node *d* is single-parent and can be reduced. Nodes *f*, *g* and *j*, the children of node *d*, are adopted by node *a*, the parent of *d*, and node *d* becomes a sink node, resulting in the graph shown in Figure 5.4. Note that node *e* is not reduced since it is in *S*.

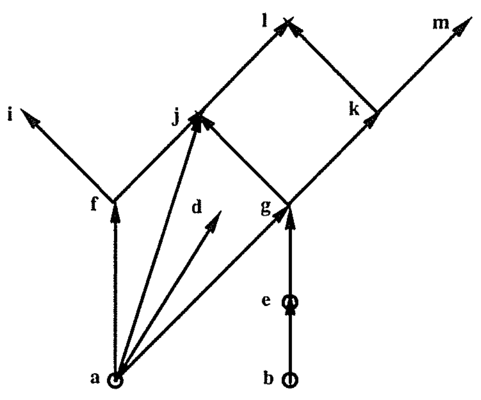


Figure 5.4. Single-Parent Optimization

The application of this optimization to several graph-based algorithms, including those by Ebert and Schmitz, has been studied in detail in [Jia90].

5.3.9 The Search Algorithm

The main advantage of computing successor lists in reverse topological sort order is that the expanded successor list of a node can be used by the parents of this node when their successor lists are being expanded. This sharing of information is useful even when the node is not a source node in the query, but some of its ancestors are sources. However, when the number of source nodes in the query (i.e., the size of S) is small, the extra cost of a topological sort and of expanding non-source nodes in the magic graph may not be justified. A simple search of the graph, starting from the source nodes and expanding only their successor lists, may be cheaper. We call such an algorithm a *search* algorithm (following [Jak91]). With a search algorithm, a multi-source query with k source nodes is essentially treated as k single-source queries.

Given a single-source selection on node s, the successor list of s may be expanded using various graph search algorithms. For reachablity, search algorithms based on BFS and on DFS are given in [Jia90]. For ordered path problems, such as shortest path, Dijkstra's algorithm achieves the optimal search order by expanding the best current successor next [Dij59]. Variations of this algorithm can be found, in [GGZ91, IRW, Jia92, SuR91].

5.3.10 Generalized Transitive Closure

Algorithms that compute transitive closure using the condensation graph are not complete, and cannot be directly adapted for generalized transitive closure computation. Properties of paths from different nodes in the same strongly connected component are in general different even if the reachability property is the same. By agglomerating all of the nodes within a strongly connected component, we lose individual paths from different nodes.

However, it is possible to generalize graph-based algorithms for the special case of an acyclic graph, or to modify the algorithms to avoid collapsing strongly connected components in a cyclic graph. Ioannidis and Ramakrishnan present several such algorithms in [IRW] and prove their correctness, i.e. completeness and irredundancy. They also compare the performance of the algorithms for a set of shortest path queries.

5.4 Hybrid Algorithms

In this section, we study algorithms that use a hybrid or mixed approach. The algorithms presented below combine ideas from both matrix and graph-based algorithms

5.4.1 The Hybrid Algorithm

In [AgJ90] it has been shown that if the nodes of a DAG are first sorted in reverse topological order, then a matrix-based computation of the closure is equivalent to a graph-based computation, and has the advantage that blocking can be used to reduce the I/O cost. Thus, the Hybrid algorithm combines the immediate successor and marking optimizations of graph-based algorithms with the use of blocking for matrix-based algorithms. (illustrated in Figure 5.1). We describe the Hybrid algorithm in more detail in Chapter 6.

5.4.2 Tree-Based Algorithms

All of the matrix and graph-based algorithms mentioned so far use the successor list of a node as the basic unit of storage and manipulation. Jakobsson [Jak91] and Dar and Jagadish [DaJ92] independently observed that the use of successor lists can result in redundant work and in unnecessary I/O, and that this extra work can be avoided if *successor spanning trees* are used instead of ''flat'' successor lists. The structural information held by the tree promotes ''sharing'' of partial computations across multiple nodes and hence leads to more efficient evaluation. Consider the graph in Figure 5.5:

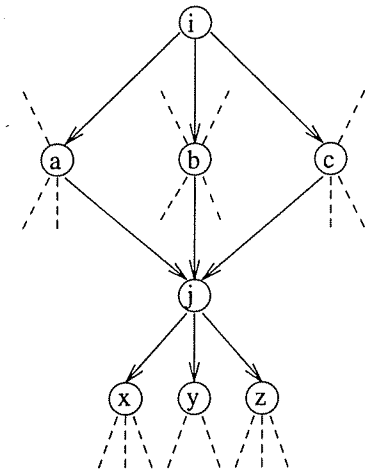


Figure 5.5. Sharing of sub-trees

Suppose we are constructing the successor spanning tree of *i*. First, we merge the spanning tree of *a* with the tree of *i*. As a result, the successor spanning tree of *j* is inserted as a subtree of *i*. Next, the spanning tree of *b* is merged with that of *a*. This merge is done recursively. First, an attempt is made to insert node *j* and its successors into the spanning tree of *i*. At this point the algorithm discovers that *j* is already a successor of *i*. Therefore, there is no need to merge the remainder of the successor spanning subtree of *j* with that of *i*. In fact, there is no need to read any successors of *j* in the successor spanning tree of *b* into memory. In transitive closure algorithms that treat the whole successor list as one unit (e.g. [AgJ90, Ebe81, EvK77, IRW, IoR88, Jia90, Sch83]) , the sucessor list of *b* would be read in its entirety and each of its elements examined for membership in the successor list of *i*, thereby performing redundant work. Finally, the spanning tree of *c* is merged with that of *i*. Once again, the subtree rooted at *j* is not read or processed. See Figure 5.6.

The fact that the children of node *j* are searched in only one tree, namely the tree of node *a*, has been called the *unique assignment property* in [Jak92].

In addition to promoting sharing, a successor spanning tree implicitly records a path between the root of the tree and each of its leaves, and can be used to answer queries that require this path information. The use of successor trees is a general technique that may be applied to different algorithms. The Spanning Tree transitive closure algorithm presented in [DaJ92] is a modification of the hybrid algorithm of [AgJ90]. A modification of the Semi-naive algorithm is presented in [Jak91]. A modification of Warshall's algorithm is developed in [Jak92]. In Chapter 6 we

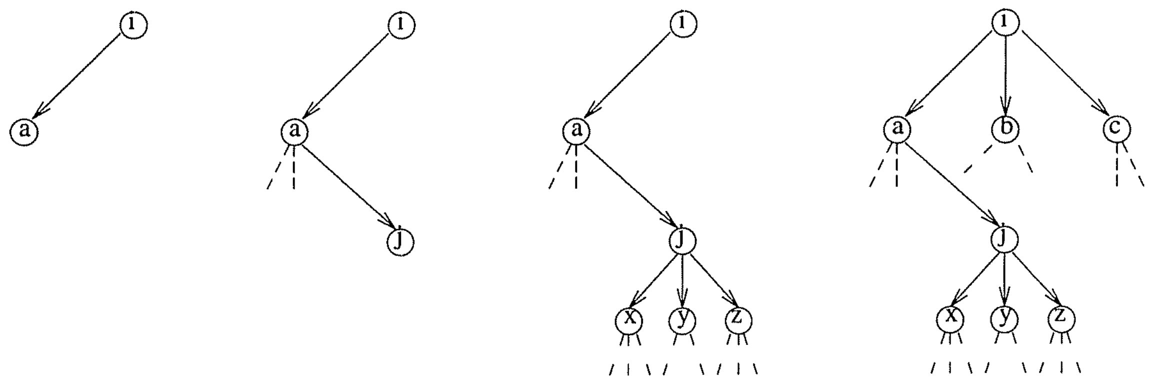


Figure 5.6. Some Stages in the Construction of tree(i)

investigate the performance of two spanning tree algorithms: an adaptation of the BTC algorithm of [IRW], and an algorithm using predecessor trees presented in [Jak92] and described in the next subsection.

5.4.3 Partial Transitive Closure

The spanning tree algorithms presented in [DaJ92] and [Jak91] handle selections in the same fashion as the successor list algorithms that they are based on. In [Jak92], Jakobsson presents a spanning tree algorithm tailored to partial transitive closure. The algorithm, called *Compute_Tree*, differs from the complete closure algorithm of [Jak91] in two important ways:

- The spanning trees are built with respect to the arc-reversed graph G_r, i.e., the algorithm uses predecessor trees rather than successor trees.
- A predecessor tree for a node x contains only a subset of the predecessor nodes of x. Those nodes are called *special* nodes (with respect to x). A node is special if it is either a source node or the nearest common ancestor of at least two source nodes that are unrelated (that is, there is no path between them). Because a predecessor tree only stores special nodes, the size of such a tree is at most twice the size of the set S, i.e., the number of source nodes specified in the query.

As an example, consider the magic graph in Figure 5.3 (b). Four possible special-node predecessor trees, for nodes f, g, j and k, are shown in Figure 5.7 (a), (b), (c) and (d). As in the Spanning Tree algorithm, the tree of a node x is constructed by consulting the trees of x's children (in the arc-reversed graph G_r). A successor tree union of node x with its child y is performed by recursively traversing the tree of y and constructing the tree of x in bottom up fashion from the leaves up to the root. For example, the tree of node j in Figure 5.7 is constructed from the trees of nodes f, g and d. Let us assume that the arc (j,f) is the first to be explored. First node a is considered. This node is special by virtue of being a source node, and is therefore added to the (currently empty) tree of j. Next node f is considered; however it is not special with respect to j and is therefore ignored. Instead, a direct arc (j,a) is added to the tree of j. Let us assume that the arc (j,g) is explored next. As a result, the source nodes b and e and the arc (e,b) are added to the tree of j. Node g is considered next, however it is not special with respect to j since it is not the nearest common ancestor of at least two unrelated source nodes (nodes b and e are related, and node d is already in the tree of j at this point). Therefore a direct arc (j,e) is added to the tree of j, resulting in the tree shown in Figure

5.7 (c). The last arc to be explored, *(j,d)*, does not contribute any new nodes or arcs to the tree of *j*. In fact, the processing of this arc could be avoided by using the marking optimization, as we discuss in detail in Section 6.5.2.3. The predecessor tree of node *k* (Figure 5.7 (d)) is constructed in a similar fashion from the tree of node *g* (Figure 5.7 (b)). Note that node *g* ends up being special with respect to node *k* but not with respect to node *j* (assuming as we did that the arc *(j,f)* is processed before the arc *(j,g)*).

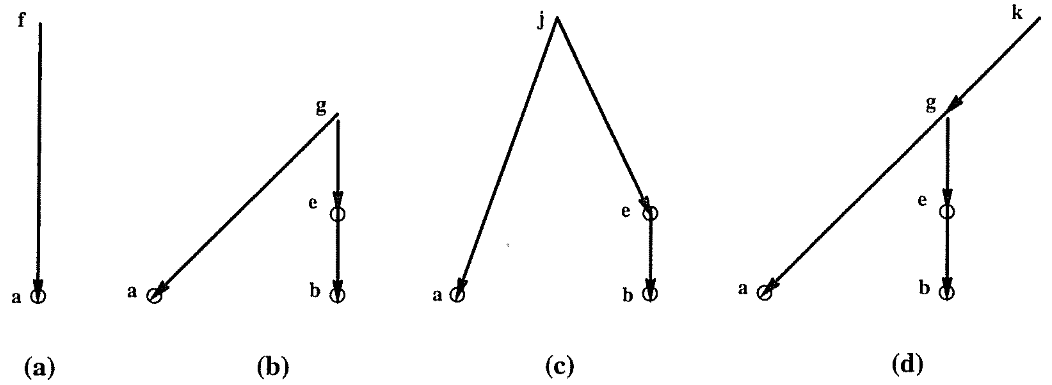


Figure 5.7. Special-Node Predecessor Trees

We will study the Compute_Tree algorithm in more detail in Chapter 6.

5.4.4 Generalized Transitive Closure

In [DaJ92] Dar and Jagadish present a generalization of the Spanning Tree algorithm for path problems. They also present an algorithm for maximizing path problems which establishes a best spanning tree out of each node.

5.5 Summary

We described many transitive closure algorithms, focusing on algorithms developed in the database context for computing the closure of a large graph. We discussed how the algorithms may be used to compute generalized transitive closure, i.e. path problems, and partial transitive closure, i.e. multi-source selection. For exposition purposes, we classified the algorithms into families according to their underlying data respresentation. We tried to illustrate the evolution of algorithms in each family as well as the integration of ideas that lead to the subsequent design of mixed-approach, or hybrid, algorithms.

Chapter 6

Performance Evaluation

This chapter provides a performance study of a restricted class of transitive closure algorithms — those that compute simple *reachability*. We compare the I/O cost of the algorithms for the computation of the full and the strong partial transitive closure on a range of input graphs. We show that significant cost tradeoffs exist between the algorithms in this range of queries, and identify factors that influence the cost of the computation. We also take a critical look at the evaluation methodology for full and partial transitive closure algorithms. We demonstrate that cost metrics based on tuples or successor lists, used in many previous studies of transitive closure, cannot be reliably used to predict the I/O cost of these algorithms.

The candidate algorithms we study are ones that can compute the partial transitive closure as well as the full transitive closure. We have chosen what we consider to be the best set of algorithms, based on previous performance studies, which we review in the related work section. Although the original descriptions of the algorithms were quite dissimilar, we observed that the algorithms can all be restated in terms of successor list manipulation. This observation lead to a uniform implementation framework, which highlights the relationship between the algorithms. The candidate algorithms are the following:

- The BTC algorithm [IRW] (see Section 5.3.6).
- The Hybrid algorithm [AgJ90] (see Section 5.4.1).
- The BFS algorithm [Jia90] (see Section 5.3.8).
- The Search algorithm (see Section 5.4.3).
- The Spanning Tree algorithm [DaJ92] (see Section 5.4.2).
- The Compute_Tree algorithm [Jak92] (see Section 5.4.3).

This chapter is organized as follows. In Section 6.1 we review the cost models that have been used in previous studies of transitive closure and recursion in databases, and motivate our choice of performance metrics. In Section 6.2 we present the implementation framework, which is common to all the algorithms. In Section 6.3 we describe the implementation of the individual algorithms. This description augments the high level description of the algorithms given in Chapter 5. In addition, while implementing the algorithms we came up with several extensions and improvements, which we present in this section. In Section 6.4 we present the experimental setup. In particular, we describe the input graphs used in our study, and identify graph properties that influence the cost of evaluating the transitive closure of the graph. In Section 6.5 we give the results of the experiments involving computation of the complete closure, and in Section 6.6 we give the results of the experiments involving computation of the partial closure, In Section 6.7 we offer our insight on the evaluation methodology for transitive closure algorithms. In

Section 6.8 we discuss related work, and in Section 6.9 we present our conclusions.

6.1 Cost Models

A large number of different cost models have been used in studies of algorithms for transitive closure and recursive query evaluation. As a consequence, a comparison of such algorithms based on a literature survey is quite difficult. Indeed, one of the results of our performance study is that different choices of cost metrics may result in drastically different ratings of the performance of the algorithms for a given query. That is the topic of Section 6.7.

The following cost models have been used in the literature:

1. **Main memory operations** [Ebe81, EvK77, Pur70, Sch83, Sch78, War75, War62]. This is the standard theoretical metric, assuming equal cost operations, e.g. testing or setting of a bit in the adjacency matrix. When established analytically, this metric corresponds to the complexity of the algorithm, and is usually given for the average or worst case. However, the number of operations may also be measured on actual input graphs [Sch83]. In either case, this measure is most useful when the computation is memory resident since it does not take I/O into account.

2. **Number of deductions** [BeR87, Jak91, Jak92, NRS89]. This is a standard metric in recursive query processing. In the case of transitive closure, it corresponds to the number of tuples generated by the algorithm, including duplicates.

3. **Number of distinct tuples derived** [MuP91]. For transitive closure, it corresponds to the number of distinct tuples generated, excluding duplicates.

4. **I/O complexity** [UlY90]. This metric is based on the Kung-Hong model [HoK80]. Essentially, Ullman and Yannakakis assume a main memory of size s ''values'' (e.g., nodes or tuples), where $n < s < e$. One I/O is performed whenever a value is moved between main memory and secondary storage. Complexity results are a function of n, e, and s, and are within an order of magnitude.

   Ullman and Yannakakis state that their results extend in a straightforward manner to a block-oriented I/O model — if a block (page) holds b values, then each expression should be divided by b. We feel that this suggestion is quite simplistic: it does not take into account the memory management policy for pages, the distribution of values on pages (e.g. clustering), or the possibility of reorganization of values across pages.

5. **Successor list I/O** [IoR88, Jia90, Jia90]. This is the number of times a successor list is moved between main memory and secondary storage. In [IoR88], this metric was established analytically for the worst case, assuming main memory of size n.

6. **Number of successor list unions** [Jia90, Jia90]. This is the number of times a union of successor lists is performed.

   The major difficulty with metrics using successor lists is that they are not uniform: successor lists of different nodes can have very different sizes, and the size of each successor list may change dramatically during the computation.

7. **Tuple I/O** [AgJ90, DaJ92]. The tuple I/O metric is similar to the notion of I/O complexity [UlY90], but it involves algorithm-specific assumptions about the way I/O is done, e.g., that successor lists are read and written as a unit. (Unlike the successor list I/O model, though, reading a successor list of size n counts as n ''tuple I/O's'' rather than one ''list I/O''.)

8. **Page I/O** [ADJ90, IRW, KIC92]. This is the number of times a page was moved between main memory and secondary storage.

9. **I/O + CPU time** [ADJ90, IRW, KIC92]. The total elapsed time may be used directly [ADJ90], but it is sensitive to the workload of the system. Instead, the measured CPU time may be combined with an estimate of the I/O time given by the number of I/O's multiplied by the postulated cost of a single I/O [IRW, KIC92].

In studies that measured both I/O cost and CPU time, the I/O cost was clearly the dominant factor [ADJ90, IRW, KIC92]. Furthermore, a close correlation of the CPU and I/O curves was observed in [ADJ90]. This observation suggests that a large portion of the CPU cost was actually spent in processing the I/O subsystem calls, making the I/O cost even more pronouned. Since the number of page I/O's is the most important metric, and it is also architecture independent, we have chosen it as the major performance yardstick for our study. We measured other metrics as well, including the total number of tuples generated, the number of duplicates produced, and the number of successor list unions, in order to be able to compare our results with those of previous studies.

6.2 Implementation Framework

An important goal of our study was to implement and compare the algorithms in a uniform way. We observed that in spite of apparent differences between the candidate algorithms, they can all be naturally formulated and implemented as manipulations of successor lists. Consequently, we implemented the algorithms in the study as variations of one basic algorithm, namely the BTC algorithm. In addition to easing the coding task, this implementation gave us considerable insight into the advantages and disadvantages of each algorithm.

We recast the transitive closure problem in terms of *successor lists* as follows. For a given source node $x \in S$, the transitive closure computation involves *expanding* S_x, the successor list of node x, such that

1. Initially, the *immediate* successor list of x is given by $S_x = \{ y \mid (x,y) \in G \}$.

2. After the computation of the closure, the *complete* successor list of x is given by $S_x = \{ y \mid (x,y) \in TC(G) \}$.

Our implementations of the candidate algorithms are based for the most part on the implementation of BTC and other DFS algorithms in [IRW, Win92]. With the exception of the Search algorithm, all of the algorithms expand successor lists in reverse topological order and utilize the immediate successor and marking optimizations, as well as inter- and intra-successor list clustering. In addition, each algorithm adds an important variation to the processing of successor lists, as we outline below.

- The Hybrid algorithm uses blocking of successor lists. A block of successor lists at a time is read into memory and expanded. With a block size of 1, the algorithm is identical to BTC.

- The BFS algorithm uses the single-parent optimization. The successor lists of single-parent nodes are not expanded. Instead, the children of such nodes are adopted by their parents.

- The Search algorithm does not use the immediate successor optimization. The successor list of a node (x) is unioned with the *immediate* successor list of every node y that is a successor of x. With the other algorithms, which use the immediate successor optimization, the successor list of a node (x) is unioned with the *complete* successor list of every node y that is an *immediate* successor of x.

- The Spanning Tree algorithm uses successor trees, which are essentially successor lists containing additional structural information. The union of successor trees of nodes x and y uses the structural information to reduce the number of arcs fetched from the tree of y and the number of duplicate arcs generated for the tree of x.

- The Compute_Tree algorithm uses predecessor trees. These trees contain only those predecessor nodes that are ''special'' with respect to the root of the tree. The union of the trees of nodes x and y is done in bottom up fashion, copying from the tree of y into the tree of x only those nodes that are special with respect to x.

Our implementations are not identical to the original descriptions of the candidate algorithms; however, we believe that we capture the important ideas underlying these algorithms. Moreover, our implementations are more detailed than any previous performance study we are aware of and take into account issues such as clustering of successor lists on pages as well as page and list replacement policies. In addition, we came up with several improvements to the individual algorithms. In the rest of this section we describe the common aspects of our implementation. Section 6.3 describes the specific implementation of each algorithm.

6.2.1 Common Aspects of the Implementation

The execution of all the algorithms (except the Search algorithm) is divided into two phases.

1. **Restructuring phase**: This phase is common to all of the algorithms. During the restructuring phase, nodes are assigned node identifiers $1..n$, where n is the number of nodes. The nodes (i.e. node identifiers) are topologically sorted. At the same time, the tuples of the input relation are converted into *successor list format* resulting in a more compact representation as we illustrate below. In addition, useful statistics about the graph are collected during this phase (see Section 6.4.3). These statistics may be used to make the computation phase more efficient, as we suggest in Section 6.5. For selection queries, the magic subgraph is identified during this phase, and the node numbering, topological sort, construction of successor lists, and gathering of statistics are done with respect to this graph.

2. **Computation phase**: This phase is different for each algorithm. It involves expanding the successor lists and then writing the expanded lists out to disk. For selection queries, only the expanded lists of the query source nodes are written out.

As an example, the graph of Figure 5.3 (a) is repeated in Figure 6.1 (a). The magic graph, with respect to the circled source nodes a, b, and e, is shown in Figure 6.1 (b), with the node names replaced by node identifiers. Next to each node in the magic graph, in parenthesis, appears its numbering (also called *rank*) in a possible topological ordering.

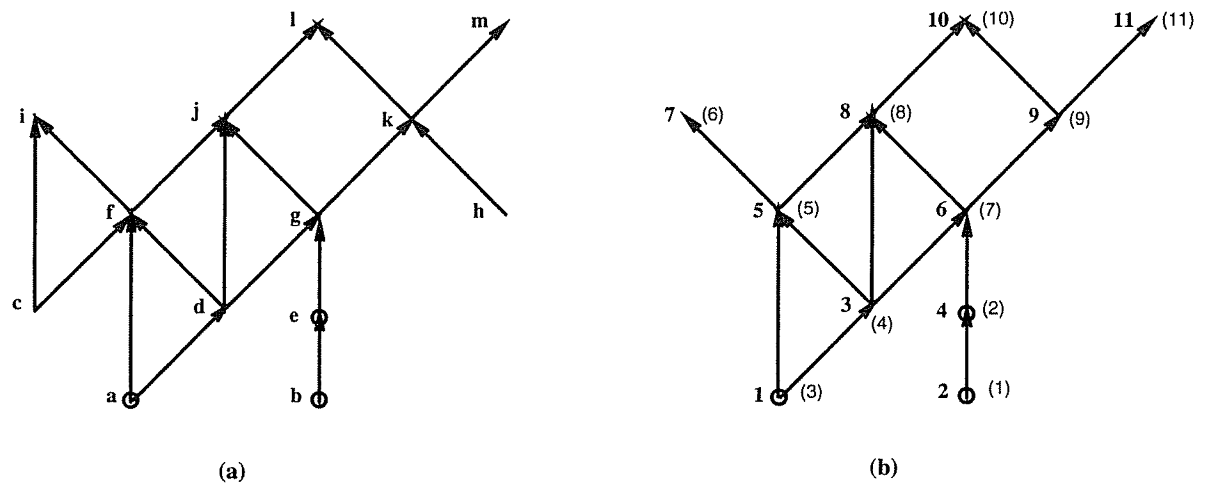


Figure 6.1. An Input Graph

We assume that the corresponding relation is stored on disk as a set of tuples clustered on the source attribute. We also assume the existence of a clustered index on the source attribute. We model the clustered index as a set of <*src*, *tid*> pairs, where the *tid* is modeled by the offset of the tuple from the beginning of the file. Figure 6.2 shows the relation and index corresponding to the graph of Figure 6.1 (a). The dashed arrows are drawn for illustrative purposes.

index

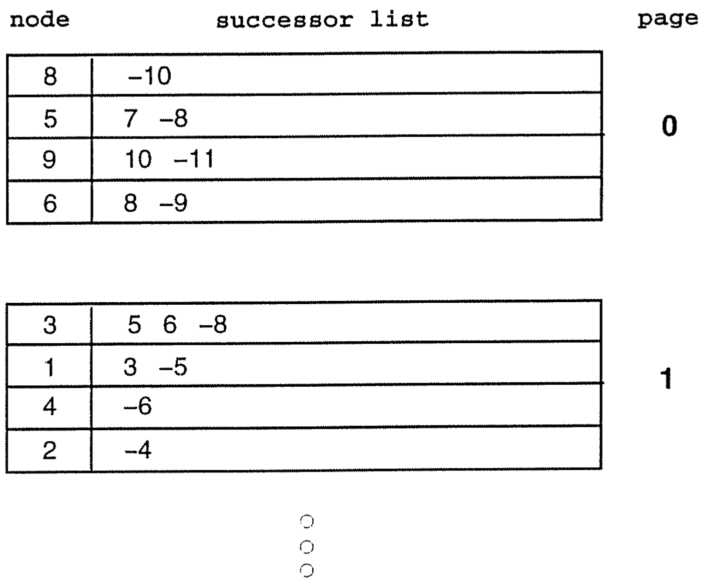
src	offset
a	0
b	8
c	12
d	20
e	32
f	36
g	44
h	52
j	56
k	60

relation

src	dest
a	d
a	f
b	e
c	f
c	i
d	g
d	j
d	f
e	f
f	j
f	i
g	k
g	j
h	k
j	l
k	m
k	l

Figure 6.2. Input Relation and Index

Figure 6.3 shows the representation of the graph of Figure 6.1 (b) in successor list format. Successor nodes are stored in fixed-length blocks, with the last immediate successor in each list designated by negating its value. The advantage of this representation is that if the size of a successor list S_x is k at some point, the successor list format requires k+1 memory slots compared to $2k$ slots required by the relation format. Some space is wasted in the successor list format due to partially filled successor blocks, but the percentage of wasted space becomes small as successor lists are expanded. In our experiments the block size was 15, while the average size of an expanded successor list was one or two orders of magnitude larger.



node	successor list	page
8	-10	0
5	7 -8	
9	10 -11	
6	8 -9	
3	5 6 -8	1
1	3 -5	
4	-6	
2	-4	

Figure 6.3. Successor Lists Format

Both inter- and intra-successor list clustering are utilized in the restructuring phase [IRW], taking advantage of the topological sort order established during this phase. *Inter-list clustering* is accomplished by storing successor lists on a page in a way that matches the order in which they will be accessed, i.e., in reverse topological order. For example, the successor list of node 8 is stored on the same page as the lists of nodes 5 and 6. *Intra-list clustering* is accomplished by storing the nodes in each successor list in topological order. This allows the algorithms to take advantage of the marking optimization: if a child j of a node i is reachable from i through another child k of i, then node k will be considered before j; thus, by the time j is considered, it will be marked and hence ignored. We implement marking by incrementing the marked node identifier by n, the number of nodes in the graph. That is, a node identifier greater than n denotes a marked node. For example, in the successor list of node 3, nodes 5 and 6 precede node 8. When nodes 5 and 6 are processed, node 8 is marked by changing its value to 19 (8+11), and a later union of its successor list with the list of node 3 is thus avoided.

6.3 The Competing Algorithms

6.3.1 The BTC Algorithm

Our implementation of the BTC algorithm follows [IRW]. In the acyclic case, nodes are topologically sorted, and then the successor lists are expanded in reverse topological order. The processing of a node involves reading the successor lists of its children and merging them with its own list (the immediate successor optimization). The children are considered in topological order, to support the marking optimization, ignoring children of the expanded node that are also non-immediate successors of it.

Topological order is not defined for cyclic graphs. To describe the processing of a cyclic graph by the BTC algorithm, we first extend trivially the definition of topological order to a general directed graph. We then use the order to ensure that the root of an SCC is the last member of the SCC to be expanded.

Definitions

1. A numbering of nodes in a (cyclic) graph is an *extended topological sort* if for each two nodes x, y such that x and y are not in the same SCC and x is a predecessor of y, $rank(x) < rank(y)$ (as in the acyclic case).

2. The node with the highest rank (extended topological sort order number) among the members of an SCC is designated as the *root* of the SCC.

For a cyclic graph, the processing of successor lists is done differently for nodes that are not in an SCC, non-root members of an SCC, and root nodes of an SCC.

1. Processing a non-SCC node results in expanding its successor list, as in the acyclic case.

2. During the processing on a non-root member of the SCC, nodes are added to the list of the root of the SCC rather than to the list of the processed node.

3. Processing of the root node involves distributing its successor list among the members of the SCC (which have all been processed already).

An extended topological sort ordering of a cyclic graph such that the root of an SCC has the highest rank in the SCC may be established by the standard algorithm for topological sort and SCC recognition [Tar72]. The combined effect of numbering the nodes by an extended topological sort, expanding nodes in reverse topological order, and handling SCC nodes as described above is the root optimization — only one successor list is built per SCC.

We denote our implementation of the BTC algorithm by *BTC*.

6.3.2 The Hybrid Algorithm

In the Hybrid algorithm [AgJ90], as in the Direct algorithms [ADJ90], successor lists are processed in blocks in order to take advantage of locality and reduce I/O (see Figure 5.1). Block sizes are determined by a parameter called *ILIMIT*. The processing of a block starts by reading successor lists into memory until $ILIMIT \times M$ pages have been read, where M is the size of the memory. Next the successor lists are expanded, and then they are written out to disk and the next block is processed. If the memory is filled up during expansion, the current block size is decreased by discarding one or more pages from the current block (this is called *dynamic blocking* in [AgJ90, ADJ90].)

With a block size of 1, the Hybrid algorithm is essentially identical to the BTC algorithm, in spite of their different representation. Thus, we wanted to see what effect the extra dimension of blocking had on the performance of the algorithm.

The implementation of the Hybrid algorithm in [AgJ90] maintained a copy of each (unexpanded) successor list in the current block, thus reducing the effective size of available memory. This was done in order to distinguish immediate and non-immediate successors to apply the immediate successor optimization. This overhead is not necessary in our implementation — the successor list structure distinguishes the two by negating the last immediate successor, as illustrated in Figure 6.4.

The algorithm presented in [AgJ90] is only applicable to acyclic graphs. After the initial topological sort, all the "1" values in the matrix are concentrated in the lower left half-matrix. The Hybrid algorithm only processed this half-matrix, i.e. it is a one-pass algorithm. Using the similarity between BTC and Hybrid, we were able to extend the algorithm to handle cyclic graphs while still maintaining the one pass property.

Our extension relies on the following observation on the numbering of nodes produced by *extended topological sort*, as defined above. Consider a "1" value that occurs in the the right upper half of the adjacency matrix. Such a "1" represents an arc (x,y), such that $rank(x) < rank(y)$. Therefore, it must be the case that x and y are in the same SCC, and x is not the root of the SCC. Such an arc can be safely ignored due to the root optimization. When node y is processed, its children will be added to the successor list of the root of the SCC, and when the root is processed (later on), the list will be distributed to all the SCC members, including x.

In addition, we had to extend the blocking and reblocking policy to take SCC's into account. Because of the root optimization, the successor list of a member of an SCC is fully expanded (i.e. contains all reachable nodes) only after all other members of the SCC, including the root node, have been processed. We must not allow the partially expanded list of a node to be joined with the successor lists of the current block before the SCC of the node has been fully expanded. We avoid this problem by we making sure that the current block does not contain successor lists from more than one (non-trivial) SCC.

We denote our implementation of the Hybrid algorithm by *HYBRID*.

6.3.3 The BFS Algorithm

Jiang's BFS algorithm [Jia90] has been implemented by extending the BTC algorithm with the *single-parent optimization*. The successor lists of nodes with a single parent (incoming arc) are not expanded. Instead, the children of single-parent nodes are "adopted" by the parent, as though they were children of that parent node.

Algorithms in [Jia90, Jia90] classify nodes as single-input or multi-input based on the input graph, regardless of the query at hand. In our implementation, we consider instead the magic subgraph reachable from the source nodes. By definition, the in-degree of each node in the magic graph is less than or equal to its in-degree in the original graph (the out-degrees are the same). Hence, the magic graph usually contains more single-parent nodes. As an example, consider Figures 5.3 (a) and (b). In the original algorithm of [Jia90], the original graph would be considered (Figure 5.3 (a)), and only node d would be classified as a single parent node and therefore reduced, resulting in the graph shown in Figure 5.4. In our algorithm, the magic graph is considered (Figure 5.3 (b)), and node k is also classified as a single parent node and reduced, resulting in the graph shown in Figure 6.4 (a). The classification of nodes is done during the restructuring phase as the magic graph is traversed.

We have also considered determining single-parent nodes dynamically rather than statically. We observed that an adopted child may become single-parent even if it had more than one parent to start with. For example, in Figure 5.3 (b), after node d is reduced, node f becomes single-parent and can be reduced as well (see Figure 6.4 (a)), resulting in the graph shown in Figure 6.4 (b). However, this optimization requires traversing nodes in topological order, and cannot be done "for free" in an algorithm that expands nodes in reverse topological order. We have

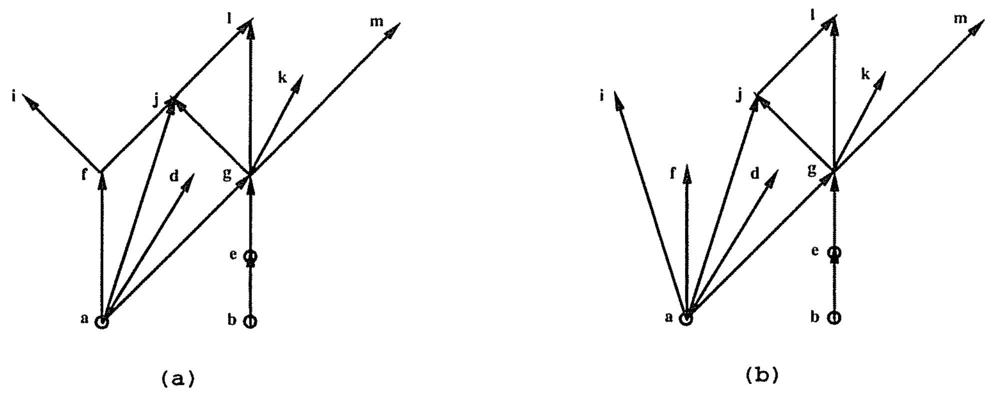


Figure 6.4. Variations of the Single-Parent Optimization

experimented with a version of Jiang's algorithm that first performs a forward scan of the graph, reducing all single-parent nodes, and than expands multi-input nodes in reverse topological order.

When applying Jiang's algorithm to a cyclic graph, we have not applied the single-parent optimization to nodes in an SCC. Because of the root optimization, we do not generate individual successor lists for SCC members anyway, but rather build a single successor list for the SCC.

We denote our two implementations of Jiang's algorithm by *BJ* and *BJ*2.

6.3.4 The Search Algorithm

The Search algorithm has been implemented by extending the preprocessing phase to expand the successor lists of the source nodes specified in the query. The computation phase is no longer needed. The algorithm computes a single-source PTC for each source node (see Section 5.4.3). One page in the buffer pool is reserved for the expanded successor list, while the other pages are used to cache pages of the original relation. As in the preprocessing phase for the other algorithms, the buffer pool is managed using the LRU policy.

We denote our implementation of the Search algorithm by *SRCH*.

6.3.5 The Spanning Tree Algorithm

In [DaJ92], a Spanning Tree algorithm was presented that was based on the Hybrid algorithm. But the idea of using successor spanning trees instead of flat successor lists is general, and can be applied to other algorithms as well. We have implemented the Spanning Tree algorithm as a modification of BTC.

The implementation in [DaJ92] used an extra column to store the derivation information. Thus, a tuple of the form (x, y, z) would represent an arc from y to z in the spanning tree of x (we say that y is the *parent* of z). The extra column was reflected in the performance evaluation of [DaJ92], which penalized the Spanning Tree algorithm by giving it 2/3 of the memory available to the other algorithms. With the restructuring of successor tuples into successor lists in memory, it is no longer necessary to store the value of x. It is possible to store the spanning tree as a list of pairs representing each successor and its parent in the tree. However, we observed that we can get a more compact representation by repeating the "denormalization step" once more. In the successor list, we store each parent (internal node) once, followed by a list of its children. Parent nodes are distinguished by negating them. For example, Figure 6.5 shows a successor spanning tree for node 1, and Figure 6.6 shows the modified successor list representation of this tree. If the average fan-out of a node is F, then this successor tree format adds on the average an overhead of $\frac{F+1}{F}$ to the storage of a flat successor list.

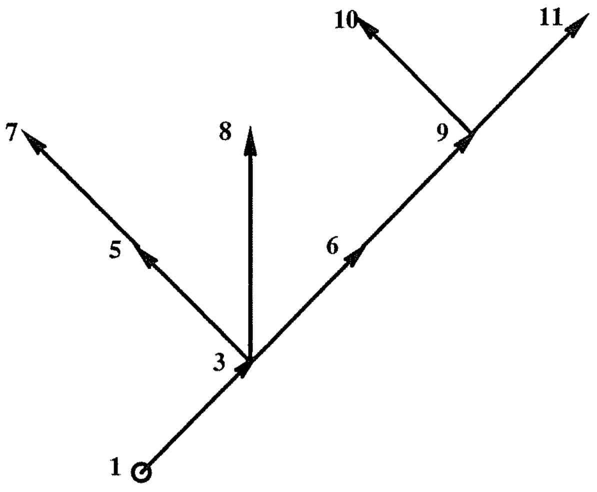


Figure 6.5. A Successor Tree

node	successor list
1	–1 3 –3 5 6 8 –5 7 –6 9 –9 10 11

Figure 6.6. In-Memory Format of Successor Tree

The union of two successor spanning trees S_i and S_j is done in selective fashion, as explained in Section 5.4.2. If a parent (negated) node x in S_j is already found in S_i then all the successors of x in S_j are ignored (see Figure 5.6).

We denote our implementation of the Spanning Tree algorithm by *SPN*.

6.3.6 The Compute_Tree Algorithm

Like the Spanning Tree algorithm, The Compute_Tree Algorithm also makes use of successor trees, but these trees are with respect to the arcs-reversed graph G_r, i.e, they are predecessor trees. The representation of predecessor trees in memory is similar to that of successor trees, described in the previous section. On the other hand, the creation of these trees in the restructuring phase is substantially different.

Given a set of source nodes S, the Compute_Tree algorithm requires the creation of immediate predecessor lists for every node in the magic graph, i.e., every node reachable from a node in S. The identification of the magic graph G_m requires a search forward from the nodes in S. To make this search I/O efficient, we would like the input relation to be clustered and indexed on the source attribute. At the same time, to identify the immediate predecessors of a node x, we need to find all nodes y such that $(y,x) \in G$. To do so in an I/O efficient way, we would like the input relation to be clustered and indexed on the destination attribute. Thus, we have two conflicting demands.

Our initial implementation of the Compute_Tree algorithm was based on the same assumptions as our implementation of the other algorithms, namely that the input relation is clustered and indexed on the source attribute. With this implementation, however, the cost of creating predecessor lists can be very high, especially when the in-degree of the graph is large and the buffer is small. The I/O cost of the restructuring phase, which creates the immediate predecessor lists, can actually exceed the cost of the computation phase, which expands these lists!

There is a more general lesson from this observation. Our study is focused on queries involving selections on source nodes. However, queries involving selections on *destination* nodes are just as important. Previous papers on computation of partial transitive closure often claimed that selections on destination nodes can be solved simply by ''inverting the graph'' and running an algorithm for selections on source nodes on the inverted graph. However, our study indicates that the cost of this transformation can be high. Perhaps the best way to support selections on either source or destination nodes is to have a dual representation of the graph in two relations. The first, the graph relation, should be clustered and indexed on the source attribute. The second, the inverse graph relation, should be clustered and indexed on the destination attribute.

With this in mind, we developed a second implementation of the Compute_Tree algorithm that assumes both representations of the input graphs are available. The graph relation is used to establish the magic graph by searching forward from the source nodes. The inverse graph relation is then used to create an immediate predecessor list for each magic node. Only arcs from other magic nodes are read from this relation into memory; other arcs are ignored. With this implementation, the I/O cost of the restructuring phase for the Compute_Tree algorithm is approximately twice the cost of that phase for the other algorithms, independent of the input graph.

We denote our two implementations of Jakobsson's algorithm by *JKB* and *JKB*2.

6.4 Experimental setup

The parameters of each experiment in our study are described in the following subsections.

6.4.1 System Parameters

The configuration of the system for each experiment is determined by the size of the buffer pool (*M*) and the list and page replacement policies [IRW]. The page size in our experiments is 2048 bytes. The input relation tuples are 8 bytes long (two integers). Hence, in the relation format (Figure 6.2), 256 tuples may be stored on a page. After conversion to successor list format (Figure 6.3) 450 successor may be stored on each page. (A successor list page is divided into 30 blocks, each holding up to 15 successor nodes. See [Win92] for details). A *list replacement policy* is used when a successor list expands to the point where at least one of the other lists on the page must be moved to a new page (i.e., the page must be split). The first policy, NC (no clustering), moves half of the lists on the page to a new page. The second, TC (topological clustering), moves either all the lists on the page that precede the expanded list in the topological order, or the lists that succeed it, depending on which set is larger. The third, DC (degree clustering), moves the lists with the minimum number of unprocessed incoming or outgoing arcs. Essentially, NC tries to maintain balanced utilization of pages, TC tries to maintain some topological clustering on the split pages, and DC tries to prevent future page faults on the new page by moving to the new page those lists that are less likely to be used again. A *page replacement policy* is used to select a page to remove from the buffer pool when it is full. The first policy, LRU (least recently used), selects as victim the page in the buffer pool that has not been accessed for the longest time. The second, LUND (least unprocessed node degree), computes for each page in the buffer pool the total number of unprocessed arcs into and out of this node (called the page's UND). It selects as the victim the page with the smallest UND.

The system parameter space for the experiments is shown in Table 6.1. The results in Section 6.5 reflect the best combination of list and page replacement policies for a given query and buffer size.

Parameter	Symbol	Values
Buffer size (in pages)	M	10,20,50
List replacement policy	–	NC, TC, DC
Page replacement policy	–	LRU, LUND

TABLE 6.1. System Parameters

6.4.2 Query Parameters

Synthetic graphs with varying characteristics are used to explore the performance of the algorithms (see e.g. [AgJ90, ADJ90, AgJ87, IRW, Jia90, Jia90]). The parameters used to guide the graph generation process are the number of nodes in the graph (n), the average out-degree (F), and the graph locality (l). The actual out degree of each node is chosen using a uniform distribution between 0 and $2F$. We experimented with boths DAGS and general (cyclic) graphs. For a DAG, arcs going out of a node are restricted to go to higher numbered nodes. Our usage of the locality parameter is similar to that of [AgJ87, IRW]. In a general graph, children of node i are in the range $[i-l \% n,\ i+l \% n]$. In a DAG, children of node i are in the range [i+1, min(i+l, n)]. This definition of locality makes it a property of the graph generation routine and not a property of the resulting graph itself. Hence, we refer to the parameter l as the *generation locality.* We have also developed other definitions of locality that do characterize a graph, independently of how it is generated, and we present those in Section 6.4.3.

The graph generation routines produce a graph relation that is clustered on the source attribute. In the case of a DAG, the resulting relation is also *topologically clustered* [LaD89]. That is, if node j is a successor of node i, then the immediate successor list of j is stored after the immediate successor list of i. This topological clustering biases the performance results, since it reduces the I/O cost of the preprocessing phase that scans the graph relation. We therefore use a separate ''scrambling'' routine to destroy this topological clustering.

For selection queries, the number of source nodes, i.e., the size of the set S, is used to control the selectivity of the query. We denote this value by s. The query parameter space for the experiments is shown in Table 6.2. We generated 5 graphs of each family (n, F, and l). In addition, for selection queries, we repeated each experiment 5 times, with a different set S of source nodes. The results presented below show the average of these experiments.

Parameter	Symbol	Values
Number of nodes	n	2000
Average out degree	F	2, 5, 20, 50
Generation locality	l	20, 200, 2000
Selectivity	s	2, 5, 20, 200, 500, 1000, 2000

TABLE 6.2. Query Parameters

Unlike most previous studies, we used statistical methods to evaluate our graph generation routines. Specifically, we used the Kolmogorov-Smirnov test [Tri82] to verify that the generated graphs are approximately normally distributed with respect to both the closure size and the total number of I/O's spent in computing the closure (using the BTC algorithm). We then checked that the variance of these variables was within the confidence interval for the confidence level of 95%. In addition, for several families of graphs, we repeated these statistical tests over 30 graphs of each family, and the results were still within the 95% confidence interval. A similar test for the distribution of the closure size in general graphs with respect to the average out degree is reported in [GKS91].

6.4.3 Derived Graph Parameters

Several statistics about the input graph (or, in case of selection, the magic graph) are collected as the graph is being scanned during the restructuring phase. Some of these statistics are trivial and need not be discussed in detail. For example, for a cyclic graph, we compute the number of strongly connected components and the average size (in number of nodes) of a component. In the rest of the section, we propose some novel statistics that may serve as the basis for a characterization of directed acyclic graphs (DAGS). In Section 6.5, we show how this characterization can assist us in making an intelligent choice of which algorithm and system configuration to employ for the computation phase of the closure.

Definitions (Node Level, Arc Locality)

Given a DAG G, we define the *node level* of each node in $i \in G$ as [19]

$$level(i) = \begin{cases} 0 & \text{if i is a sink node} \\ 1 + \max\limits_{(i,j) \in G} level(j) & \text{otherwise} \end{cases}$$

We define the *arc locality* of each arc $(i,j) \in G$ as

$locality(i,j) = level(i) - level(j)$

19. Similar definitions for node level were given in [HuS93, Jak93].

As an example, Figure 6.7 shows the graph of Figure 6.1 (b), rearranged according to its node levels. The locality of each arc is given in parenthesis.

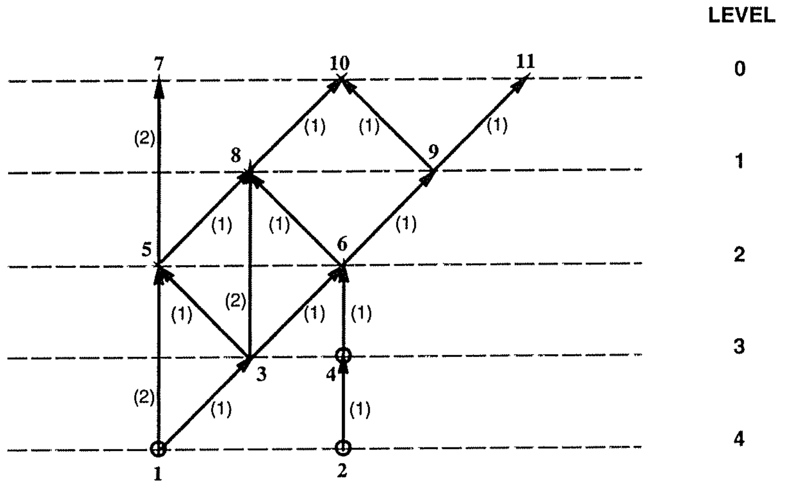


Figure 6.7. Node Levels and Arc Localities of a DAG

The following are a few properties of node levels and arc localities which follow directly from their definitions.

1. A node at level $k > 0$ has out-going arcs to nodes at levels lower than k, with at least one arc to a node at level $k-1$.
2. For each arc $(i,j) \in G$, $locality(i,j) \geq 1$
3. A node at level k has a path of k arcs to some sink node in the graph, where the locality of each arc along the path is 1.
4. The maximum node level in a DAG G is equal to the depth of G, i.e., the length of the longest path in G.

The locality of an arc (i,j) intuitively indicates the "distance" that the arc spans between i and j. This distance effects the cost of processing the arc, i.e., merging S_j with S_i. Because successor lists are expanded in reverse topological order, the probability that S_j is in memory when S_i is expanded is higher if the distance between i and j is small (the arc (i,j) has *high* locality). The processing of an arc (i,j) does not require a union of the two successor lists, however, if the arc is *marked*, which is the case if an alternative path from i to j is found in G. Hence, the locality of *unmarked* arcs is a better indicator than the locality of all arcs of the probability than a union between S_i and S_j will require an I/O operation to bring S_j into memory. As noted in Section 5.3.4, when children are expanded in topological order, an arc (i,j) will be marked during the expansion of S_i if and only if (i,j) is in the *transitive reduction* of G. We assume that G has no duplicate arcs since such arcs are indistinguishable. Because G is a DAG, its transitive reduction is unique [AHU75]. This discussion motivates the following definition.

Definitions (Redundant Arcs)

Let G be a DAG, and let $TR(G)$ be the transitive reduction of G. We say that an arc $(i,j) \in G$ is *redundant* if $(i,j) \notin TR(G)$. Otherwise, we say that (i,j) is *irredundant.*

Given the above definitions, we can now suggest a simple model for representing an arbitrary directed acyclic graph.

Definitions (Height and Width of a DAG)

Given a DAG G, we define the *height* of G, denoted by $H(G)$, as

$$H(G) = \underset{i \in G}{\text{AVG}}\ level(i)$$

Let $|G|$ be the number of arcs in G. We define the *width* of G, denoted by $W(G)$, as

$$W(G) = \frac{|G|}{H(G)}$$

Using these definitions, an arbitrary DAG G is mapped into a *rectangle model*, with height $H(G)$ and width $W(G)$. This is illustrated schematically in Figure 6.8. The lemma below shows an interesting property of this rectangle model.

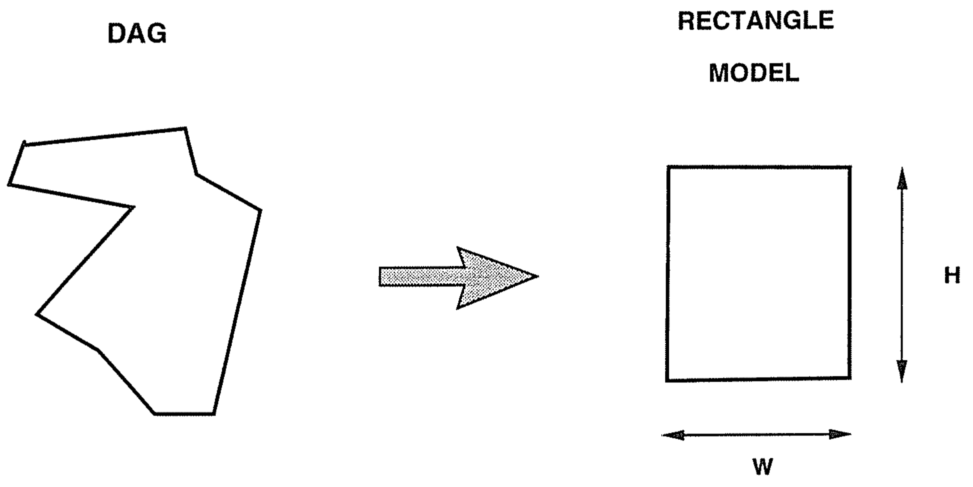


Figure 6.8. A Rectangle Model for DAGS

Theorem 6.1

Let G be a DAG. Let $TR(G)$ be the transitive reduction of G, and let $TC(G)$ be the transitive closure of G. Then

1. $H(G) = H(TR(G)) = H(TC(G))$
2. $W(TR(G)) \leq W(G) \leq W(TC(G))$

Proof.

1. The set of nodes in G, $TR(G)$, and $TC(G)$ is the same. Let $(i,j) \in G$ be an arc of locality 1, and let G' be the graph G with the arc (i,j) removed. There cannot exist a path from i to j in G' without violating the assumption that the locality of (i,j) in G is 1. It follows immediately that the closure of G and G' is different, since $(i,j) \in TC(G)$, but $(i,j) \notin TC(G')$. Hence, the transitive reduction of G cannot remove from G any arcs whose locality is 1. Similarly, by the definition of transitive closure, the closure of G cannot add to G any

arcs whose locality is 1. As property 3 above states, the level of a node $i \in G$ is established along a (not necessarily unique) path of arcs with locality 1. Therefore, the level of each node $i \in G$ remains the same in both $TR(G)$ and $TC(G)$.

2. This follows directly from the definition of the width of a DAG, the fact that $|TR(G)| \leq |W(G)| \leq |TC(G)|$ and part 1 of this theorem. ■

The relationship between G, $TR(G)$ and $TC(G)$ in the rectangle model is illustrated in Figure 6.9.

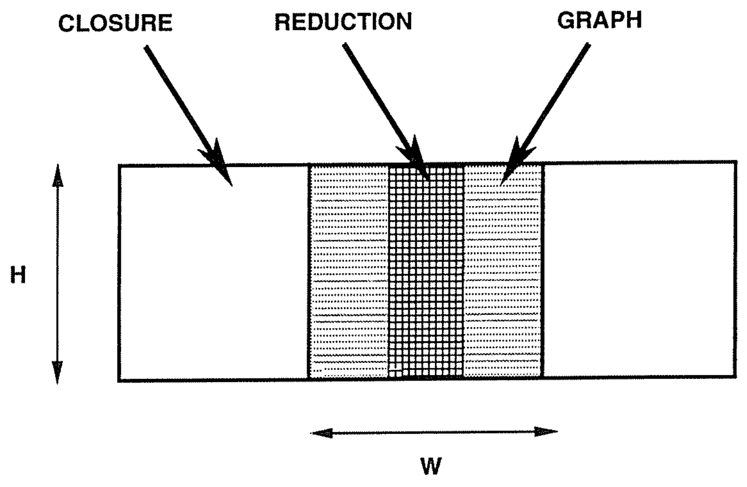


Figure 6.9. Transitive Reduction and Closure in the Rectangle Model

Intuitively, we suggest that the height of a DAG G is an indication of the ''shallowness'' of G, while the width of G is an indication of the ''degree of redundancy'' in G. A high value of $H(G)$ implies that paths in G tend to be long. A high value of $W(G)$ implies that many redundant paths exists in G. In Section 6.5, we suggest that the rectangle model may be useful in understanding the cost of computing the transitive closure of G. The following simple theorem shows that this model can be computed ''cheaply'' using a single scan of G. In particular, we can compute the rectangle model of G during the restructuring phase at no additional cost.

Theorem 6.2

Let G be a DAG. We can compute $H(G)$ and $W(G)$ using a DFS traversal of G.

Proof.

A DFS traversal of G can be trivially modified to count the number of nodes and arcs in G. From the definition of node level, it is also clear than the level of each node can be established immediately after all of its children are visited, as part of the backtracking step of the DFS algorithm. The number of nodes N, the number of arcs e, and the total of the node levels in G are all that is required to compute $H(G)$ and $W(G)$. ■

Table 6.3 characterizes the DAGS used in our study. For each DAG, we give its generation parameters F and l (the number of nodes N was 2000 for all the graphs). We then show the resulting number of arcs in the graph [20], the maximum node level, the height and width of the graph according to the rectangle model, the average locality of all arcs and of the irredundant arcs, and finally, the size of the transitive closure of the graph.

graph name	out deg. F	generation locality l	number of arcs $\|G\|$	max. node level	graph height H	graph width W	average arc locality	average irredundant locality	closure size $\|TC(G)\|$
$G1$	2	20	3892	297	108	36	34	8	1124406
$G2$	2	200	4053	52	20	202	8	3	674123
$G3$	2	2000	4393	25	11	399	5	2	125610
$G4$	5	20	8605	573	253	34	32	5	1750499
$G5$	5	200	9876	115	55	179	11	5	1497537
$G6$	5	2000	9984	48	29	344	10	5	563333
$G7$	20	20	23365	1192	581	40	21	1	1948375
$G8$	20	200	32724	335	174	214	20	4	1883612
$G9$	20	2000	38731	152	106	365	34	6	1463591
$G10$	50	20	33025	1605	798	41	18	1	1974648
$G11$	50	200	82676	610	317	260	34	3	1948217
$G12$	50	2000	92381	273	188	491	65	6	1778046

TABLE 6.3. Graph Parameters

We observe that as the out-degree is increased or the generation locality is decreased (higher values of F, lower values of l), the graphs become deeper and larger. We also note that the average locality of irredundant arcs is much lower than the average locality of all arcs. This emphasizes the importance of the marking optimization: not only does it reduce the number of successor list unions, it also avoids those unions that are likely to be *expensive*, that is, those with high probability of requiring an I/O to be performed.

6.5 Performance Study Results

We now present and analyze the results of our performance study. We first discuss queries that compute the complete transitive closure, and we then discuss queries that compute partial transitive closure with varying selectivity. We present results of experiments involving acyclic graphs only, for the following reasons. First, several of the algorithms that we study are only applicable to acyclic graphs (e.g. [DaJ92, Jak92]). Second, for reachability queries, the closure of a cyclic graph can be computed using the Purdom algorithm [Pur70] by collapsing strongly connected components and computing the closure of the resulting acyclic condensation graph. This technique has been shown to be a strong optimization (see e.g. [GKS91]) and is used by all of the algorithms that we study except the Search algorithm. Third, random graphs generated with an out degree greater than 1 tend to have most nodes concentrated in one large strongly connected component (see [Kar90, SeN91]). Hence, the condensation graph of these random graphs is quite small.

20. The reader may note that the number of arcs is often less than $N \times F$. There are two reasons for this: (1) duplicate tuples produced by the graph generation routine were eliminated, and (2) the locality of a graph provided an upper bound on the actual out-degree of each node (see especially graph $G10$).

6.5.1 I/O and CPU Cost Breakdown — General Trends

Table 6.4 presents a breakdown of the execution cost of algorithm *BTC* for computing the full closure of graph *G*6 with 10, 20 and 50 buffer pages. The experiments were run on a DECstation 5000/25 with 24 MB of memory and an RZ24 SCSI local disk. The real time, user time and system time were obtained using the Unix™ *time* command, while the number of page I/O's was recorded by the simulated buffer manager. The last column in Table 6.4 shows the estimated total I/O time for each experiment if I/O's were not simulated. This number was calculated by multiplying the number of simulated I/O by 20ms, the approximate time for a single I/O on the machine. (The 20ms value was established by a separate set of experiments and is consistent with the disk specifications.) All times given in Table 6.4 are in seconds.

Buffer Size	Real Time	User Time	System Time	Simulated I/O's	Estimated I/O Time
10	52.50	47.28	4.03	10764	215.28
20	51.58	47.20	2.83	9684	193.68
50	49.35	47.35	0.63	8047	160.94

TABLE 6.4. I/O and CPU Cost of BTC (Graph *G*6, CTC, 10-50 buffers)

Comparing the measured user time with the estimated I/O time we see that the closure computation is clearly I/O bound for all three buffer pool sizes used in these experiments. Figure 6.10 shows the I/O breakdown between the preprocessing (restructuring) and computation (expansion) phases relative to the buffer pool size for the closure computation summarized in Table 6.4. The computation phase dominates the I/O cost for all buffer sizes. When the buffer pool size reaches 50, the input relation fits in memory, and the cost of the restructuring phase becomes negligible.

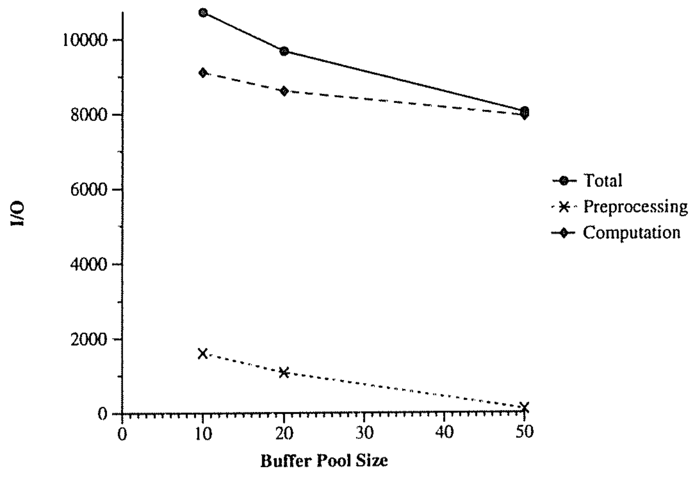


Figure 6.10. I/O Cost Breakdown for BTC (Graph *G*6, CTC, 10-50 buffers)

We also profiled the closure computation summarized in Table 6.4 in order to explore the breakdown of the CPU cost. As expected, the CPU cost was dominated by the cost of operations on successor lists, and in particular, the cost of list union. Duplicate elimination using bit vectors was found to be quite cheap, as we discuss in more detail in the following section.

6.5.2 Computing Full Closure

For ease of exposition, we present the full closure (CTC) results by comparing *BTC* with the other algorithms. For CTC computation, algorithm *BJ* is identical to *BTC*, since it cannot eliminate any non-source nodes. Algorithm *BJ*2 has a slightly higher cost since it performs an extra forward search before expanding nodes in reverse topological order. However, the difference is negligible. We therefore focus on the *HYB*, *SPN*, *JKB* and *JKB*2 algorithms.

Figure 6.11 compares the I/O cost of *BTC* and *HYB* with respect to the available buffer size on graph *G*6. The number in the legend of each of the curves for *HYB* is the value of *ILIMIT*, the ratio of the buffer pool reserved for the diagonal block successor lists. When *ILIMIT* is increased, the cost of *HYB* increases as well. In fact, the algorithm performs best when no blocking is used, in which case it is identical to *BTC*.

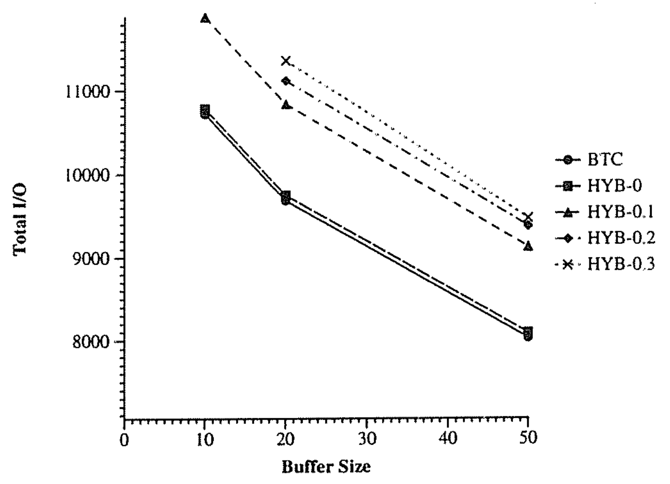


Figure 6.11. Hybrid vs. BTC, Full Closure (*G*9, 10-50 buffer pages)

Blocking was found to be a useful technique for reducing I/O in the Direct algorithms presented in [ADJ90]. It is therefore surprising that blocking has a detrimental effect on the performance of the Hybrid algorithm. However, a closer look at the way that these algorithms utilize blocking explains the discrepancy.

Consider Figure 5.1 again. The central idea behind blocking is that once an off-diagonal list is brought into memory, it may be joined with several diagonal block lists. The number of "interactions" (leading to unions) between the diagonal block lists and the off-diagonal list represents the *blocking benefit* of the algorithm. For example, in Figure 5.1, the off-diagonal list interacts with 3 diagonal block successor lists (blocking benefit = 3). However, there is a major difference between the Direct algorithms and the Hybrid algorithm — the Hybrid algorithm makes use of the immediate successor optimization, which the Direct algorithms do not utilize. In the Direct algorithms, when a successor list is processed, it is joined with it the successor lists of *any* of its successors. In the Hybrid algorithm, when a successor list is processed, it is joined with it the successor lists of its *immediate* successors only. Thus, the number of potential interactions, i.e. the potential blocking benefit, is much smaller in the Hybrid algorithm.

At the same time, blocking may degrade the performance of the Hybrid algorithm for several reasons. First, since the diagonal block lists are fixed in memory, the buffer pool becomes effectively smaller. Second, expansion of the diagonal block lists may cause reblocking, i.e., discarding pages holding other successor lists from the buffer pool. Third, in the Hybrid algorithm the off-diagonal part of each diagonal list is processed before the diagonal part. This may cause the algorithm to expand redundant arcs, arcs that would not be expanded if a strict right-to-left order, corresponding to the topological order, was used. With these negative factors combined, the cost of blocking in the Hybrid algorithm becomes greater than its benefit.

Figure 6.12 (a) compares the I/O cost of *BTC* and the successor tree algorithms *SPN*, *JKB* and *JKB*2, with respect to the average node degree on graphs generated with locality 200 with a buffer pool of 20 pages. The *BTC* algorithm performs better than the other algorithms since its flat successor lists occupy less space then the successor trees of the other algorithms. The successor tree algorithms actually fetch a smaller number of successors because of the selective fashion in which the union of two trees is performed (see Section 5.4.2). However, the reduced ''tuple I/O'' does not translate into reduced page I/O. When processing an arc *(i,j)*, the first page of the successor tree of *j* must always be accessed. For example, consider Figures 5.5 and 5.6 and the union of the successor trees of nodes *i* and *b* (transition from 3rd to 4th stage in Figure 5.6). We must read a part of the successor tree of *b* in order to find out that node *j* is the only child of *b* and consequently to discontinue the tree union (since *j* is already in the tree of *i*). In general, the union of two successor trees, S_i and S_j, may save page I/O compared to a successor list union only if two conditions are met: (i) the successor tree of *j* spans multiple pages, and (ii) there is at least one page of *j's* successor tree such that all the successors of *j* on that page need not be added to the successor tree of *i* (since they are already there, or, in the case of *JKB* and *JKB*2, since they are not special with respect to *i*). In our experiments we found that almost always each of the pages holding the successor tree of *j* had to be accessed, and real (vs. tuple) I/O was not saved.

As Figure 6.12 (a) illustrates, the *SPN* algorithm closes the gap with the *BTC* algorithm as the out-degree goes up, as the relative overhead of storing parent nodes in the successor tree becomes smaller. The *JKB* and *JKB*2 algorithms continue to perform poorly since the cost of the restructuring phase (building *predecessor* trees) is higher for these algorithms. The preprocessing cost for *JKB* becomes very high for a high out-degree, while the preprocessing cost for *JKB*2 is approximately twice that of *BTC* (see Section 6.3.5).

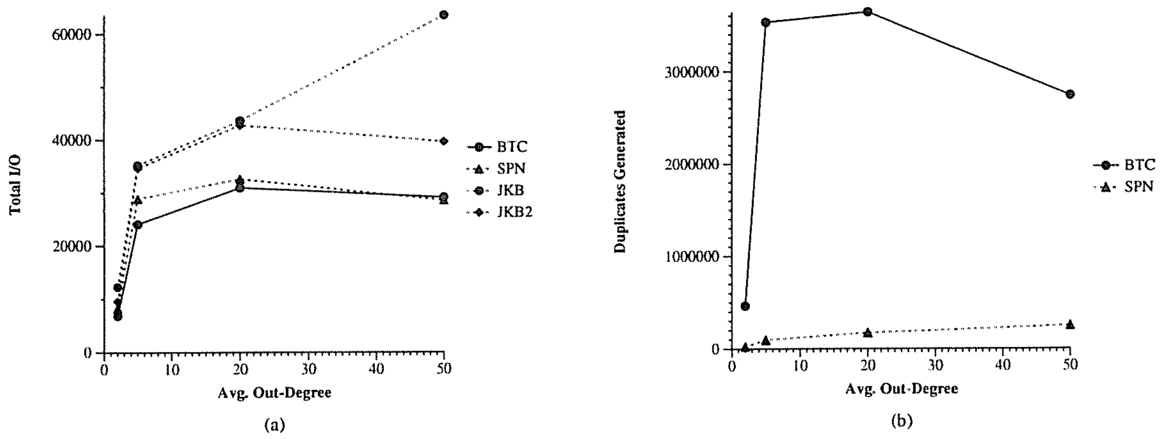


Figure 6.12. The Successor Tree Algorithms vs. BTC, Full Closure (20 buffer pages)

The successor tree algorithms generate far fewer duplicates than the flat successor lists algorithms. Figure 6.12 (b) shows the number of duplicates generated by *BTC* and *SPN* for the same graphs used in Figure 6.12 (a). However, since duplicate elimination is performed in memory, this cost is not reflected in our model. Further, we believe that the CPU cost of in-memory duplicate elimination can be made negligible, e.g., by using bit vectors, as we did in our implementation. As an example, we analyzed the profile of the computation of the full closure of graph $G6$ using algorithm *BTC* with 50 buffer pages. 759548 duplicates were produced during the execution of the algorithm in addition to 563333 new tuples. The cost of duplicate elimination using bit vectors was found to be less than 6% of the total CPU cost of computing the closure. The total CPU cost, in turn, was only about 23% of the estimated total (CPU+I/O) cost of the execution (see Table 6.4). Overall, we estimate that 2-3 seconds out of approximately 50 seconds CPU time and 210 seconds total time were spent on duplicate elimination. Finally, however, we should note that in addition to determining reachability between two nodes in the graph, the successor tree algorithms also establish a path between the two nodes. This additional information, if needed, may justify the higher I/O cost of these algorithms.

The performance of the *HYB* and *SPN* algorithms relative to *BTC* was similar for experiments involving selection. The performance of *JKB* was almost always inferior to that of *JKB*2, and the cost of the preprocessing phase for *JKB* when the out-degree of the graph was high was prohibitly expensive. Therefore, we omit algorithms *HYB*, *SPN* and *JKB* from the discussion of partial transitive closure.

6.5.3 Computing Partial Closure

The computation of the partial transitive closure displays a high variance in the relative performance of the algorithms over different data sets. Overall, the I/O cost of *BJ* and *BJ*2, the two variations of Jiang's algorithm, is slightly lower than that of *BTC*. The *SRCH* algorithm performs the best when the number of source nodes was small, but its cost increases rapidly as the number of source nodes is increased. The cost of *JKB*2 varies greatly with respect to the other algorithms. We consider first PTC computations with high selectivity (2-20 source nodes) and than with low selectivity (200-2000 source nodes).

6.5.3.1 High Selectivity

Figures 6.13 (a) and (b) show two extreme examples of selections with high selectivity. Both graphs display total I/O with respect to s, the number of source nodes, with a buffer pool size of 10 pages. In Figure 6.13 (a), with $G4$ being the input graph, *JKB*2 performs about 1/3 the I/O of algorithms *BTC*, *BJ* and *BJ*2. In Figure 6.13 (b), with $G11$ being the input graph, *JKB*2 performs about 2-3 times more I/O than the other algorithms. How do we explain this behavior?

We identify two major factors that determine the I/O cost of the algorithms for computing selections. We describe these factors in the following subsections.

6.5.3.2 Selection Efficiency

First, we observe that in order to compute the closure of the source nodes in S, the PTC algorithms compute the closure of other non-source nodes in the magic graph. Let tc be the number of tuples generated by an algorithm, and stc be the number of those tuples that are part of the expanded successor list of a node in S. We call the ratio $\frac{stc}{tc}$ the

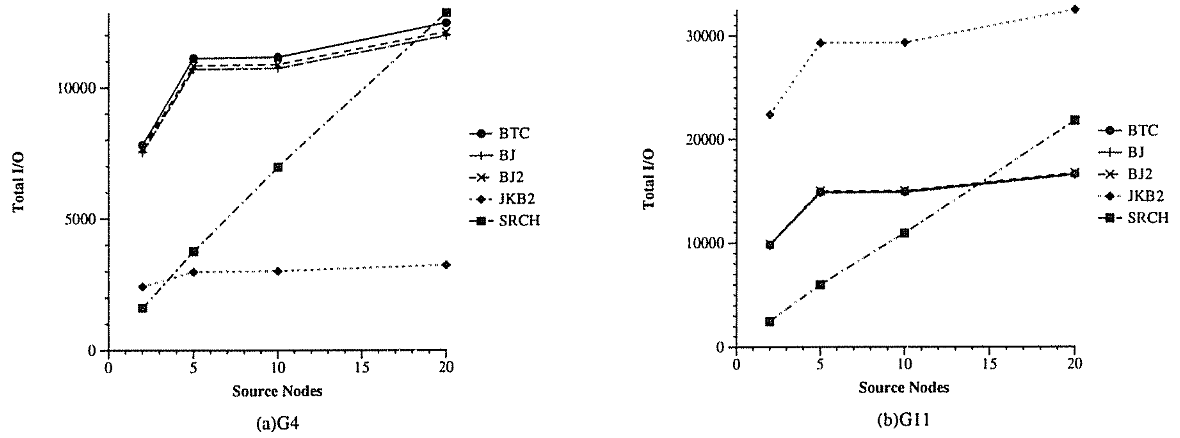


Figure 6.13. Selection with High Selectivity ($G4$ and $G11$, 10 buffer pages)

selection efficiency of the algorithm, and denote it by *se*. The value of *stc* must be the same for all algorithms, since this is the number of tuples in the answer to the query. On the other hand, the value of *tc*, and consequently, the value of *se*, differs between the algorithms. A high value of *se* indicates that the algorithm computes the selection efficiently, in the sense that most of the tuples that it produces are directly part of the answer to the query.

By definition, the *SRCH* algorithm achieves a selection efficiency of 1. In contrast, the *BTC* algorithm exhibits a poor selection efficiency since it completely expands all of the successor lists of nodes in the magic graph. The *BJ* and *BJ*2 algorithms are somewhat more efficient since they avoid expanding the successor lists of single-parent nodes; *BJ*2 is slightly more efficient than *BJ* since it identifies more single-parent nodes. Algorithm *JKB*2 also expands the successor lists (i.e predecessor trees) of each node in the magic graph. However, these lists are *partial* — only *special* nodes are included in each list, and the number of such nodes is at most $2s$.

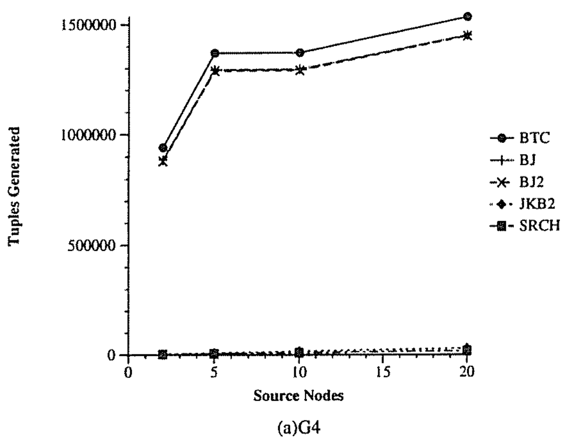
Figures 6.14 (a) and (b) show the values of *tc* for algorithms *BTC*, *BJ*, *BJ*2, *SRCH* and *JKB*2 for the queries of Figures 6.13. Figures 6.14 (c) and (d) show the selection efficiency exhibited by the algorithms for these queries.

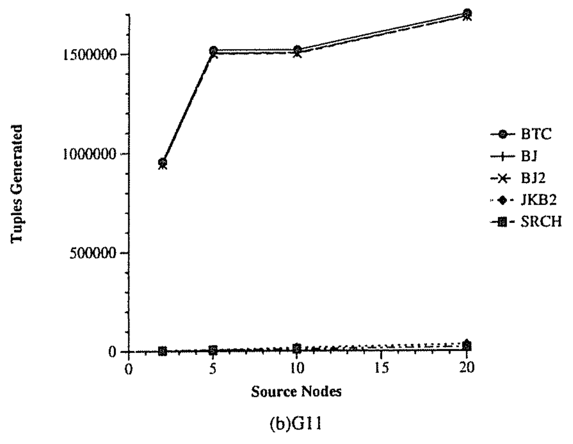
As expected, the selection efficiency of algorithms *BTC*, *BJ* and *BJ*2 is poor. Algorithms *BJ* and *BJ*2 display a higher selection efficiency than *BTC*, with *BJ*2 being slightly better than *BJ*. Algorithm *JKB*2 achieves 60%-70% of the optimal selection efficiency exhibited by the *SRCH* algorithm, generating less than 1% of the number of tuples produced by the *BTC*, *BJ* and *BJ*2.

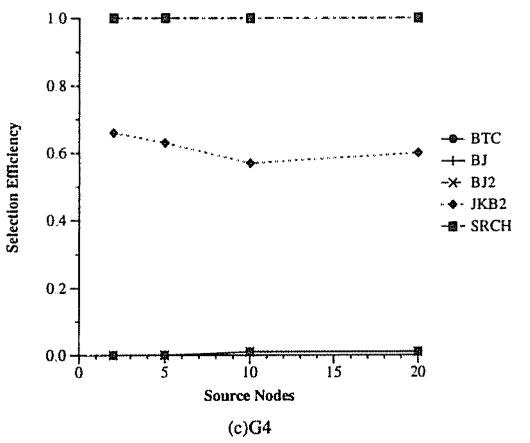
6.5.3.3 Marking Utilization

Figures 6.15 (a) and (b) show the number of successor list unions performed by algorithms *BTC*, *BJ*, *BJ*2, *SRCH* and *JKB*2 for the queries of Figures 6.13.

The number of unions performed by the *SRCH* algorithm increases rapidly as the number of source nodes for the query goes up, causing the performance of the algorithm to deteriorate. This is because the the *SRCH* algorithm does not utilize the immediate successor optimization. The number of unions performed by *BTC*, *BJ*, and *BJ*2 is almost identical (*BJ*2 is slightly lower than *BJ*, which is slightly lower than *BTC*, because they do not expand single-parent nodes). The number of unions performed by *JKB*2 is much higher than that of *BTC*, *BJ* and *BJ*2.







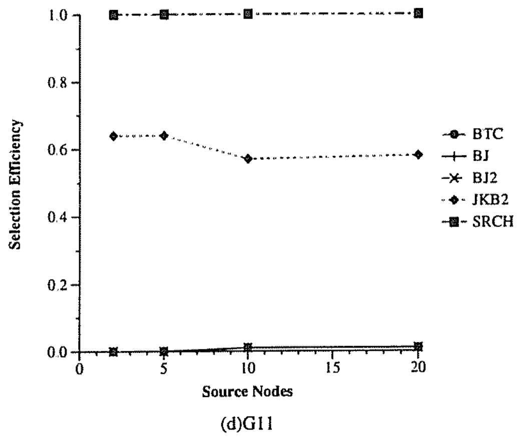
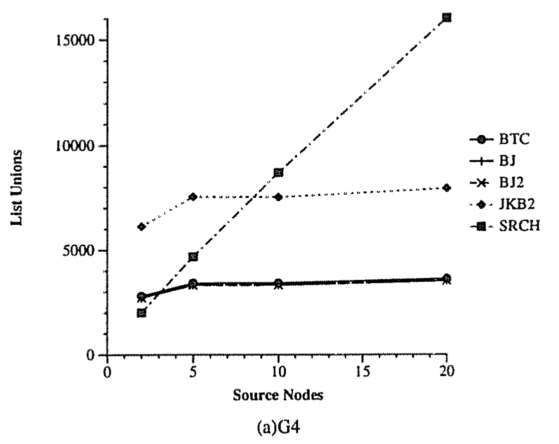


Figure 6.14. Selection Efficiency ($G4$ and $G11$, 10 buffer pages)



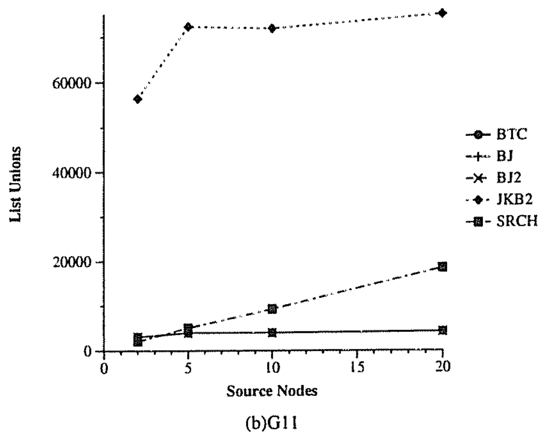


Figure 6.15. Successor List Unions ($G4$ and $G11$, 10 buffer pages)

Algorithm *JKB* performs so many unions because it misses many opportunities to apply the marking optimization. This is because the algorithm uses *partial* successor lists (i.e. predecessor trees) — the list of a node *i* only records nodes that are special with respect to *i*. As an example, consider the magic graph of Figure 5.3 (b), and the predecessor trees for nodes *f* and *j* shown in Figures 5.7 (a) and (c) respectively.

Node *d*, which is a child of nodes *f* and *j* in the inverse graph, does not appear in the tree of *f*. Therefore, the processing of the arc (*j*,*f*) does not result in the marking of the arc (*j*,*d*). When the arc (*j*,*d*) is later considered, a union of the tree of *d* with the tree of *j* is carried out. This union cannot contribute any new arcs to the tree of *j*; all of the predecessors of *d* that are special with respect to *j* (in this case, only node *a*) have already been inserted to the tree of *j* as a result of processing the arc (*j*,*f*). This redundant union requires the predecessor tree of *d* to be in memory, and may cause an I/O if it is not already there.

Figures 6.16 (a) and (b) show the percentage of arcs that were marked by algorithms *BTC*, *BJ*, *BJ*2, and *JKB*2 for the queries of Figures 6.13 (this percentage is 0 for algorithm *SRCH*). The marking percentage is almost 0 for *JKB*2, showing that the algorithm misses almost all of the opportunities to apply the marking optimization.

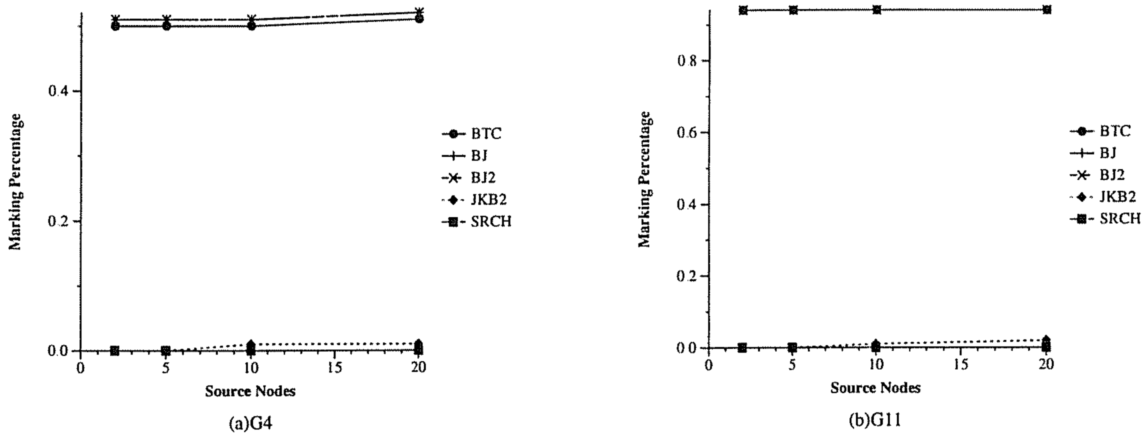


Figure 6.16. Making Percentage ($G4$ and $G11$, 10 buffer pages)

Another consequence of the missed marking opportunities is that the unions performed by *JKB* are more "expensive" than the unions performed by the other algorithms. Figures 6.17 (a) and (b) show the average locality of unmarked arcs for the four algorithms for the queries of Figures 6.13. The locality is much worse for *JKB*. As we discussed in Section 6.4.3, the worse this locality is, the higher the probability that the child's successor list is not found in the buffer pool and that an I/O will thus be required.

We observe that the poor marking utilization problem may arise with any algorithm that uses incomplete successor lists (or trees), independent of the specific criterion by which the nodes in the lists are chosen (i.e., the definition of *special* nodes).

6.5.3.4 Comparing *JKB*2 to *BTC*

The two factors displayed in the last two subsections, *selection efficiency* and *marking utilization*, combine to determine the performance of algorithm *JKB*2 relative to *BTC*. Given the great differences between the algorithms along these dimensions, the following question arises — can we suggest criteria by which to decide which of these algorithms will do better for the computation of a PTC query on a given graph?

We attempt to answer this question by suggesting that the I/O cost of *JKB* relative to *BTC* depends on the *width* of the input graph (see Section 6.4.3). Table 6.5 lists the width of graphs $G1$ through $G12$ and the Total I/O cost of *JKB* relative to *BTC* for computing PTC with 5 and 10 source nodes with a buffer pool pf 10 pages. The graphs are

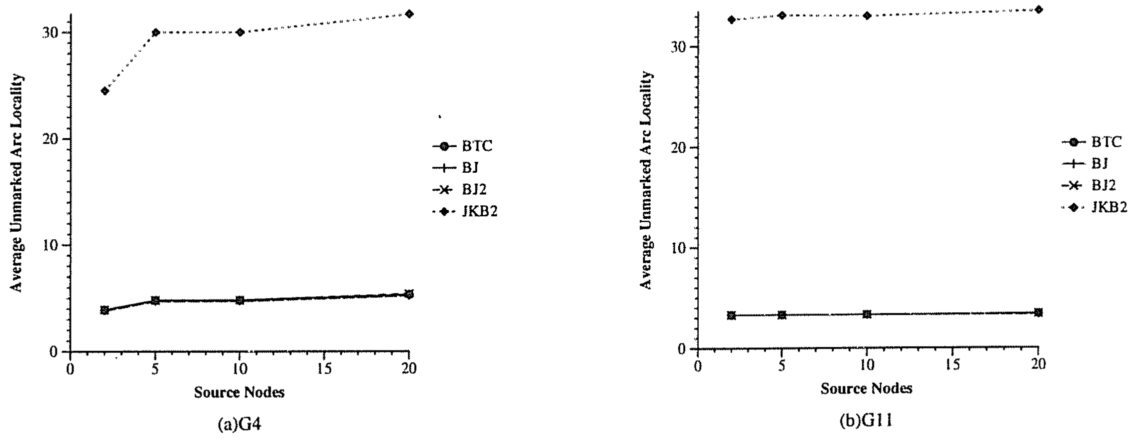


Figure 6.17. Average Irredundant Arc Locality ($G4$ and $G11$, 10 buffer pages)

sorted in increasing width order, and the I/O cost of *JKB* is normalized with respect to the cost of *BTC* for the same queries.

Graph	$G4$	$G1$	$G7$	$G10$	$G5$	$G2$	$G8$	$G11$	$G6$	$G9$	$G3$	$G12$
Width	34	36	40	41	179	202	214	260	344	365	399	491
$s = 5$	0.27	0.39	0.43	0.6	0.35	0.86	0.76	1.97	1.1	1.92	1.54	3.24
$s = 10$	0.28	0.38	0.43	0.6	0.39	0.9	0.80	1.97	1.32	1.86	1.42	3.21
Height	253	108	581	798	55	20	174	317	29	106	11	188

TABLE 6.5. Comparing *JKB* and *BTC* for PTC Queries

Generally speaking, algorithm *JKB* performs well when the width of the graph is low, but badly when the width of the graph is high. This is because of the poor marking utilization of *JKB* which, on wide graphs, causes an excessive number of list unions. The size of the lists being unioned is of secondary importance compared with the number of these unions. Consequently, *JKB* is sensitive mostly to the width of the graph, and is less sensitive to its height. To illustrate this observation we also list the height of graphs $G1$ through $G12$ on the bottom of Table 6.5.

We believe that many realistic database graphs are relatively shallow and wide. For example, the bill-of-materials graph for an automobile database may include thousands of different nodes (parts). The number of arcs (part inclusion relationships) may be several times larger. Yet, the average height of a node (the length of paths between complex and basic parts) is likely to be small. Hence the sensitivity of *JKB* to the width of the input graph may impede its performance for such database applications.

6.5.3.5 Effect of Buffer Pool Size

Figures 6.18 (a) and (b) show the total I/O cost of algorithms *BTC*, *JKB*2 and *SRCH* for graphs $G4$ and $G11$ for a PTC computation with 5 source nodes as the buffer pool size is increased from 10 to 50. Figures 6.18 (c) and (d) show the buffer pool hit ratio of the algorithms for the queries and buffer sizes of Figures 6.18 (a) and (b). For *BTC* and *JKB*2 the hit ratio does not take into account the preprocessing phase, i.e., it gives the percentage of successor list page requests during the computation phase that were satisfied from the buffer pool.

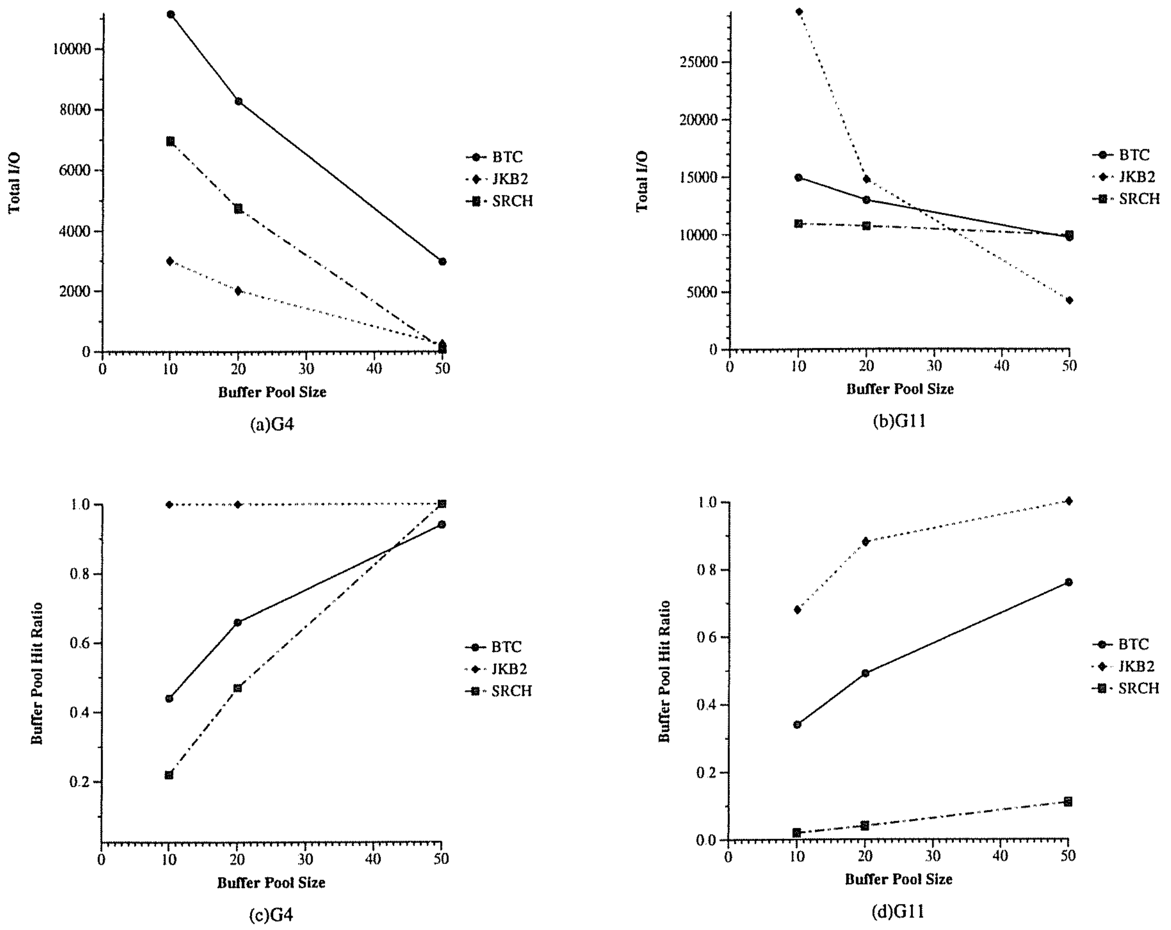


Figure 6.18. High Selectivity — Effect of Buffer Pool Size ($G4$ and $G11$, 10 Source Nodes)

The performance of all three algorithms improves as the buffer pool size goes up since the buffer pool hit ratio improves. The *JKB*2 algorithm is the most sensitive to the buffer pool size. With a buffer page size of 10 for graph $G4$ and of 50 for graph $G11$ the computation of the closure by algorithm *JKB*2 becomes memory resident and the algorithm performs almost no I/O's during the computation phase. This is because the successor lists (i.e. predecessor trees) used by *JKB*2 are much smaller on average than the successor lists used by *BTC*, since they contain only special nodes (see Section 6.5.2.2). The I/O cost of *JKB*2 at this point is dominated by the cost of the preprocessing phase. The reader should keep in mind, however, that *JKB*2 is an implementation of the Compute_Tree algorithm that relies on a dual representation of the input graph by two relations with associated clustered indices (see Section 6.3.5). Without such a dual representation the cost of the preprocessing phase for the Compute_Tree algorithm can be much higher (for example, observe the different curves for *JKB* and *JKB*2 in Figure 6.12 (a)).

6.5.3.6 Low Selectivity

For the low selectivity PTC queries (200-2000 source nodes) the cost of the *SRCH* algorithm is much higher than the other algorithms, usually by 1-2 orders of magnitude. This is to be expected since the algorithm performs an independent search from each of the source nodes. The other algorithms exhibit similar trends to the high selectivity queries, but the magnitudes of the differences between them are much smaller. Figures 6.19 (a) - (d) show respectively the total I/O, number of tuples generated, marking percentage, and number of list unions for *BTC*, *BJ*, *BJ*2 and *JKB*2 for graph *G*9 with 20 buffer pages. Algorithms *BJ* and *BJ*2 perform almost the same as *BTC* in this range, since they can find few non-source single-parent nodes to eliminate. (Recall that the average in-degree of nodes in the magic graph goes up as *s*, the number of source nodes, goes up.) Considering algorithm *JKB*2, we see that both the positive aspects of the algorithm's use of partial successor lists (higher selection efficiency, less tuples generated) and the negative aspects (lower marking percentage, more unions) diminish as *s* approaches 2000. This is because the number of special nodes in the successor trees created by *JKB*2 increases with *s*. Finally, at $s = 2000$ (full closure), the three curves converge. The total I/O curve is higher for *JKB*2 even at this point because of the extra parent information stored in its successor trees (see discussion in Section 6.5.1). The performance of *JKB*2 is actually worse at $s = 1000$ than at $s = 2000$, which we attribute to the fact that at $s = 1000$ the algorithm still misses about 30% of the marking opportunities, consequently performing many redundant unions.

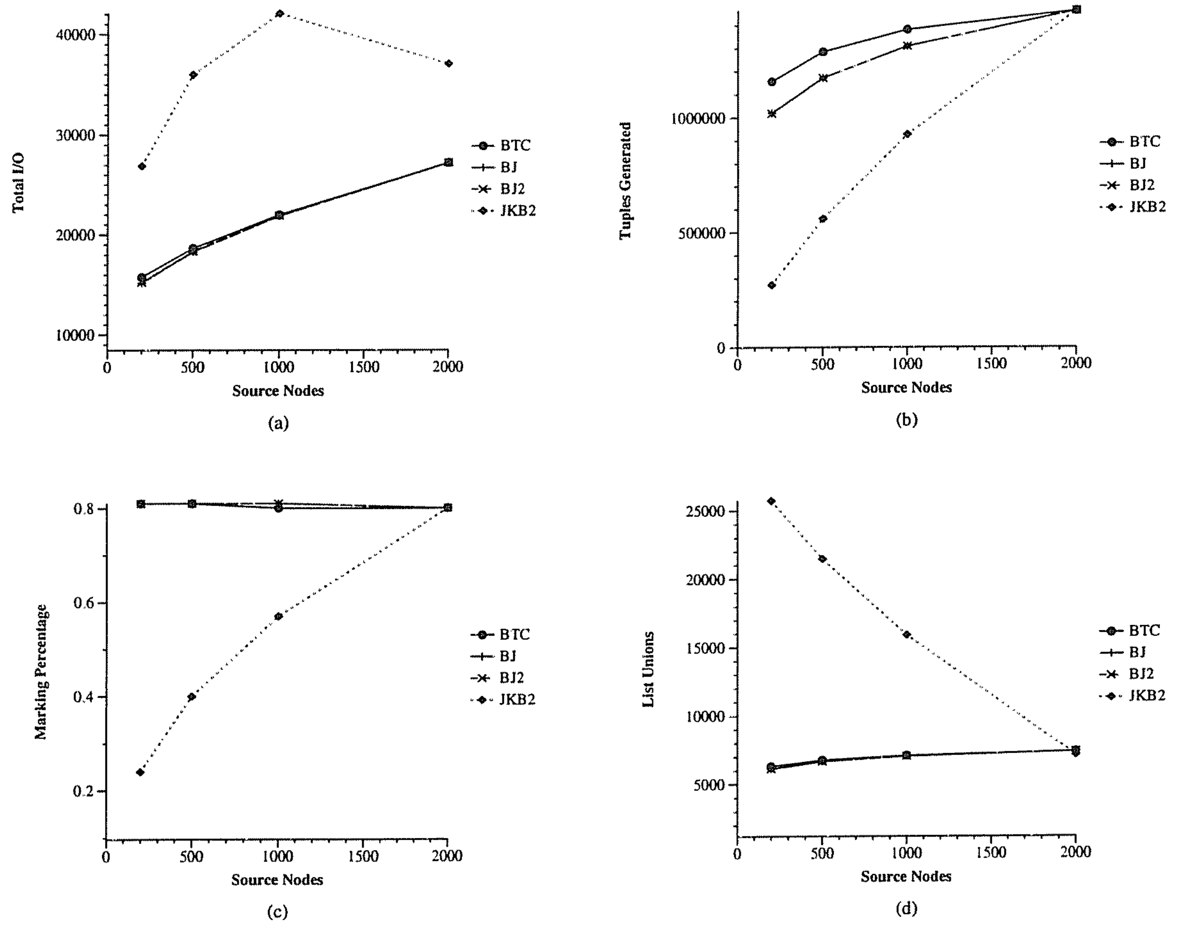


Figure 6.19. Low Selectivity Trends (*G*9, 20 buffer pages)

6.5.4 System Configuration

Each experiment in our study was repeated with all combinations of the two page replacement policies (LUND and LRU) and the three list replacement policies (TC, NC, and DC) discussed in Section 6.4.1. By and large, the choice of these policies had a secondary effect compared to the factors discussed in the previous subsections, and the difference between the best and worst combination was under 20%.

The choice of page replacement policy for the computation of the complete closure was investigated in [IRW]. Ioannidis et al suggested that LRU was the better policy for large and deep graphs, while LUND was better for shallow and small graphs. This may be explained as follows. The LUND policy tries to keep expanded successor lists with many incoming arcs in memory since such lists may be unioned with several of their parents' lists. If the graph is small, those unions are likely to occur shortly after each other. If the graph is large, however, the unions may be far in between, and in the meanwhile, the successor lists that are cached in memory by the LUND policy occupy space that would have been used for other lists under the LRU policy. These trends concerning the size of the graph are with respect to the available buffer size: with a larger buffer, more successor lists can fit in memory at once, and the LUND policy becomes better even for deeper and larger graphs. Regarding the list replacement policy, Ioannidis et al found DC to be the preferred policy in conjunction with LUND, and NC to be preferred with LRU.

Our results generally confirm these suggestions, and they extend them in the following way. When computing PTC, the ratio of the total number of tuples *tc* generated by an algorithm and the available buffer space *M* is the dominant factor effecting the utility of the LUND policy. LUND is therefore the preferred policy for high-selectivity queries, and is less attractive for low selectivity queries with a small buffer. Considering list replacement policies, the TC policy, which was added to this study and was not used in [IRW], was overall the best choice since it keeps the successor lists of parents and children clustered across page splits, allowing later unions of their lists to be performed with higher locality. This policy did not do as well with the *JKB* and *JKB*2 algorithms for high selectivity queries, however, where the NC policy was usually the best. The explanation for this phenomena is twofold. On the one hand, the high number of redundant unions carried out by these algorithms (see Section 6.5.2.1), diminishes the effect of clustering. On the other hand, because the partial successor lists (trees) of these algorithms are smaller than the successor lists of the other algorithms, more of them can fit on a page. The NC policy, which balances the distribution of successor lists across pages, helps to achieve a high utilization of pages in this situation.

6.6 Evaluation Methodology

A significant consequence of our study is a better understanding of the impact that the choice of cost metrics can have on the results of a performance evaluation. In Section 6.2 we reviewed many cost metrics that have been suggested in the literature for evaluating and comparing transitive closure algorithms. Since the computation of the transitive closure of a large graph is likely to be I/O bound (see e.g., [ADJ90, IRW, KIC92]), an important question to be asked is — can cost metrics which count operations at the tuple or successor list level be used to estimate the I/O cost of a transitive closure computation?

We believe that the results of Section 6.4 clearly demonstrate that such extrapolation is highly unreliable. We substantiate this claim with a few examples.

- Based on the number of tuples produced, or the tuple I/O (or I/O complexity) metric, the successor tree based algorithms would appear to perform much better than the *BTC* algorithm for the computation of the full closure (Figure 6.12), while in fact they perform worse.

- Based on the number of distinct tuples derived, algorithm *JKB*2 would appear to always outperform algorithm *BTC* for the computation of partial closure with high selectivity (Figures 6.14 (a) and (b)).
- On the other hand, based on the number of successor list unions, or the successor list I/O metric, the opposite conclusion would be drawn (Figures 6.15 (a) and (b)).
- As we demonstrate, neither of these conclusions holds in all cases (Figures 6.13 (a) and (b)).

Our conclusion is that a reliable evaluation of the page I/O cost of a transitive closure computation can only be obtained via a performance study that directly considers that I/O cost, rather than trying to estimate it using higher level metrics and postulating some correlation between these higher level metrics and the page I/O metric.

A related question is the choice of evaluation method. The three common methods of evaluating algorithms are *analysis*, *simulation*, and *implementation*. In our opinion. it is highly doubtful that an analysis of page I/O is feasible. Likewise, while simulation often offers more flexibility than an implementation, such a simulation must cover the components of the database system that determine the cost of I/O intensive operations: clustering of data on disk and in memory, indexing, and buffer management. The simulation used for this study addresses all of these issues. In the restructuring phase, tuples from the graph relation are converted into successor lists format using a clustered index (Section 6.2.1). The organization of these successor lists utilizes both inter- and intra-successor list clustering. In addition, we experimented with several page and list replacement policies (Section 6.4.1), and suggested heuristics for choosing suitable policies for a given query (Section 6.5.3). We believe that only such a careful simulation or implementation is likely to provide an accurate understanding of the I/O behavior of transitive closure algorithms.

6.7 Related Work

Many algorithms have been suggested in recent years for the computation of the complete or partial transitive closure of a large graph. However, most previous performance investigations have involved a small set of algorithms and data sets, and only in a few studies were the algorithms actually implemented and the I/O cost directly measured. We review earlier performance studies below and briefly summarize their results.

Agrawal et al [ADJ90, AgJ87] studied the Blocked Warshall (aka. Blocked Column) and Blocked Warren (aka. Blocked Row) Direct algorithms as well as an implementation of seminaive based on single-sided composition. They measured page I/O and elapsed time and found I/O to be the dominant cost factor. The Direct algorithms were found to perform much better for the computation of CTC over all input graphs.

Kabler et al [KIC92] explored several implementations of the Seminaive, Smart and Blocked Warren algorithms using different join methods, duplicate elimination techniques, and buffer management policies. They implemented the algorithms and measured page I/O and elapsed time. They found Seminaive to always outperform Smart. Blocked Warren was overall the best algorithm for CTC, but for PTC computation with less than 1/3 of the source nodes, Seminaive performed better.

The results of [ADJ90, AgJ87, KIC92] demonstrated that the matrix-based algorithms out-perform the iterative algorithms for the computation of the complete transitive closure by a wide margin.

Agrawal and Jagadish [AgJ90] compared the Hybrid algorithm to the Blocked Warren algorithm [ADJ90], the Grid algorithm [UlY90] and a DFS algorithm based on [IoR88] and extended with caching of successor lists. Tuple I/O was the performance metric. Hybrid was found to be the best performer. It won over the Blocked Warren algorithm by employing the immediate successor and marking optimizations of graph-based algorithms. It performed better than DFS due to the use of blocking, and it did better than the Grid algorithm because in the latter the block sizes cannot be determined dynamically, and memory is thus underutilized.

Dar and Jagadish [DaJ92] compared the Spanning Tree algorithm to the Hybrid algorithm. Tuple I/O was the performance metric. The Spanning Tree algorithm, although it was penalized via a 50% overhead for storing the tree structure, still outperformed Hybrid, since its selective union of successor trees avoids redundant tuple reads.

Ioannidis et al [IRW] studied the performance of several graph-based algorithms including Schmitz [Sch83], BTC, and the more complex DFS algorithms presented in [IRW]. They measured page I/O and CPU time. Overall, BTC was found to be the best algorithm with regard to both I/O and CPU cost. Ioannidis et al found that the algorithms were usually I/O bound. They explored several techniques for reducing the I/O cost of the algorithms, including inter and intra successor list clustering and the use of alternative list and page replacement policies (see Section 6.2.1). The implementation of the algorithms for this study was based on the implementation of Ioannidis et al, as described in [Win92]. Ioannidis et al also compared the performance of the graph based algorithms to the performance of the Seminaive, Smart and Blocked Warren algorithms, using the results of [KIC92], for similar graphs and buffer sizes. The graph-based algorithms were superior over all input graphs, with an order of magnitude difference for cyclic graphs.

Jiang [Jia90] investigated the computation of PTC with varying selectivity, including single-source and strong multi-source selections. He compared many graph based algorithms, including those by Eve and Kurki-Sonio [EvK77], Ebert [Ebe81], two algorithms from Schmitz [Sch83], the DFTC algorithm from [IoR88], and several algorithms of his own [Jia90, Jia90]. The two performance metrics were the number of successor list reads and the number of successor list unions. The main results were that BFS was better than DFS for computing multi-source queries, and that the use of the single-parent optimization reduces the cost of the algorithms by avoiding both reads and unions of single-parent successor lists.

Taken together, the results reported in [AgJ90, DaJ92, IRW, Jia90] clearly demonstrate that the graph-based algorithms and their hybrid variants are superior to the iterative and matrix-based algorithms, both for the computation of CTC and for the computation of PTC with low selectivity (large number of source nodes). We therefore focused our performance study on graph-based and hybrid algorithms, including the algorithms that were the best performers in those earlier studies. As suggested in [Jak92, Jia90], for PTC with high selectivity, a simple search from the source nodes may be the most efficient strategy. Hence, we also added the Search algorithm to our repertoire.

Materialization of transitive closure has been studied in many recent papers, and several of these papers investigate the performance of various schemes for encoding the transitive closure graph (see [AgK93, AgJ89, ABJ89, ChH91, GuY, HuS93, Jag90, LaD89, VaB86, Yan90, Yel88]). The emphasis of this work is on developing data structures that represent the transitive closure graph and that support queries against the precomputed closure, and on algorithms that maintain the consistency of the data structures when the underlying graph is modified. In contrast, we study the cost of computing the full or partial closure ''from scratch''. These two approaches are complimentary, and the choice between them depends to a large degree on the frequency of updates to the graph relation, an issue that we do not consider in this thesis.

6.8 Conclusions

In this chapter, we have studied the performance of reachability algorithms for computing the full and partial transitive closure. Our investigation was based on a detailed simulation of the algorithms using a uniform implementation framework. The following are the main conclusions:

1. The Hybrid algorithm and the Spanning Tree algorithms generally performed worse than the other algorithms. In the case of Hybrid, we attributed that to the ineffectiveness of the blocking technique used by the algorithm. In the case of the Spanning Tree algorithms, we have shown that the savings in tuple reads and in the number of duplicates produced does not necessarily translate into an improved I/O behavior.

2. Jiang's single-parent optimization offered a (small) performance improvement for PTC queries, especially for high selectivity queries over graphs with a small out-degree.

3. The performance of Jakobsson's algorithm varied widely for high selectivity queries. We identified two factors, selection efficiency and marking utilization, that determine the performance of the algorithm for such queries. We also developed a simple model for characterizing directed acyclic graphs, and we presented some evidence that this model may be useful in understanding the performance of Jakobsson's algorithm relative to the BTC algorithm for PTC queries.

4. The Search algorithm performed best for high selectivity queries. As anticipated, its performance deteriorated rapidly as the number of source nodes was increased.

5. In addition to directly comparing specific transitive closure algorithms, we investigated the evaluation methodology for such algorithms. We demonstrated that performance metrics based on tuples or successor lists, used in many previous studies, simply cannot be used reliably to evaluate the I/O cost of transitive closure computation.

Chapter 7

Conclusions

This thesis has presented a framework for augmenting databases with Generalized Transitive Closure (GTC) functionality. It makes contributions in the following areas: algebraic modeling of GTC queries, language extensions to express such queries, techniques for optimizing them, algorithms for executing them, and performance criteria for evaluating these algorithms.

Our GTC query model incorporates both path enumeration and path aggregation queries, and allows selections on arcs, paths, and sets of paths. The SQL/TC language extends SQL with the ability to formulate such queries in a declarative and concise fashion, and in particular, to record and manipulate path information in a special PATH relation. We illustrated the expressive power of SQL/TC through many examples. We investigated the optimization of GTC queries involving selections and label computations, and suggested techniques to reduce both the number of paths generated and the space required to store these paths. We then examined many transitive closure algorithms proposed in the database literature. We discussed the extension of these algorithms to compute both general and partial transitive closures. We studied closely the performance of a subset of these algorithms for the restricted case of computing full or partial reachability; we believe that this study sheds new light on the methodology necessary for reliably evaluating the performance of transitive closure algorithms.

We plan to extend our performance study by considering algorithms for generalized transitive closure and in particular, shortest path algorithms. We also intend to explore other methods for predicting the performance of transitive closure algorithms in order to assist a query optimizer in selecting a suitable algorithm for a given query. Finally, we hope to implement the ideas presented in this thesis and to integrate them into a working database system.

REFERENCES

[ADJ88a] R. Agrawal, S. Dar and H. V. Jagadish, ''Transitive Closure Algorithms Revisited: The Case of Path Computations'', Technical Memorandum, AT&T Bell Laboratories, Murray Hill, New Jersey, 1988.

[ADJ88b] R. Agrawal, S. Dar and H. V. Jagadish, ''On Transitive Closure Problems Involving Path Computations'', Technical Memorandum, AT&T Bell Laboratories, Murray Hill, New Jersey, 1988.

[AgJ90] R. Agrawal and H. V. Jagadish, ''Hybrid Transitive Closure Algorithms'', *Proc. 16th Int'l Conf. Very Large Data Bases*, Brisbane, Australia, 1990.

[AgK93] R. Agrawal and G. Kiernan, ''An Access Structure for Generalized Transitive Closure Queries'', *Proc. IEEE 9th Int'l Conf. on Data Engineering*, Vienna, Austria, April 1993.

[AgD89] R. Agrawal and P. Devanbu, ''Moving Selections into Linear Least Fixpoint Queries'', *IEEE Trans. Knowledge and Data Engineering 1*, 4 (Dec. 1989), 452-461. Also Proc. IEEE 4th Int'l Conf. Data Engineering (Feb. 1988).

[Agr87] R. Agrawal, ''Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries'', *Proc. IEEE 3rd Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1987, 580-590. Also in *IEEE Trans. Software Eng.* 14, 7 (July 1988), 879-885.

[ADJ89] R. Agrawal, S. Dar and H. V. Jagadish, ''Composition of Database Relations'', *Proc. IEEE 5th Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1989, 102-109.

[AgJ89] R. Agrawal and H. V. Jagadish, ''Materialization and Incremental Update of Path Information'', *Proc. IEEE 5th Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1989.

[ABJ89] R. Agrawal, A. Borgida and H. V. Jagadish, ''Efficient Management of Transitive Relationships in Large Data and Knowledge Bases'', *Proc. ACM-SIGMOD 1989 Int'l Conf. on Management of Data*, Portland, Oregon, May-June 1989.

[ADJ90] R. Agrawal, S. Dar and H. V. Jagadish, ''Direct Transitive Closure Algorithms: Design and Performance Evaluation'', *ACM Trans. Database Syst. 15*, 3 (Sep. 1990), . (Preliminary version appeared as: R. Agrawal and H.V. Jagadish, ''Direct Algorithms for Computing the Transitive Closure of Database Relations'', *Proc. 13th Int'l Conf. Very Large Data Bases*, Brighton, England, Sept. 1987).

[AgJ87] R. Agrawal and H. V. Jagadish, ''Direct Algorithms for Computing the Transitive Closure of Database Relations'', *Proc. 13th Int'l Conf. Very Large Data Bases*, Brighton, England, Sept. 1987, 255-266.

[AHU75] A. V. Aho, E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1975.

[AhU79] A. V. Aho and J. D. Ullman, ''Universality of Data Retrieval Languages'', *Proc. 6th ACM Symp. Principles of Programming Languages*, San-Antonio, Texas, Jan. 1979, 110-120.

[AHB86] P. M. G. Apers, M. A. W. Houtsma and F. Brandse, ''Extending a Relational Interface with Recursion'', *Proc. 6th Advanced Database Symposium*, Tokyo, Japan, Aug. 1986, 159-166.

[BKM89] I. Balbin, D. B. Kemp, K. Meenakshi and K. Ramamohanarao, ''Propagating Constraints in Recursive Deductive Databases'', *Proc. North American Conference on Logic Programming*, Oct. 1989, 16-20.

[BaR87] I. Balbin and K. Ramamohanarao, ''A Generalization of the Differential Approach to Recursive Query Evaluation'', *Journal of Logic Programming 4*, 3 (Sep. 1987), .

[BaR] F. Bancilhon and R. Ramakrishnan, ''Performance Evaluation of Data Intensive Logic Programs'', in *Foundations of Deductive Databases and Logic Programming*, J. Minker (ed.), Morgan Kaufman, . to appear.

[Ban85] F. Bancilhon, ''Naive Evaluation of Recursively Defined Relations'', in *On Knowledge Base Management Systems — Integrating Database and AI Systems*, M. Brodie and J. Mysopoulos (ed.), Springer-Verlag, 1985.

[BeR87] C. Beeri and R. Ramakrishnan, ''On the Power of Magic'', *Proc. 6th Symp. Principles of Database Systems*, San Diego, California, March 1987, 269-283.

[BKB90] C. Beeri, P. Kanellakis, F. Bancilhon and R. Ramakrishnan, ''Bounds on the Propagation of Selection into Logic Programs'', *J. Computer and System Sciences 41*, 2 (October 1990), 157-180. (Preliminary version appeared as: C. Beeri, P. Kanellakis, F. Bancilhon and R. Ramakrishnan ''Bounds on the Propagation of Selection into Logic Programs'', *Proc. 6th Symp. Principles of Database Systems*, San Diego, California, 1987).

[Bel58] R. E. Bellman, ''On a Routing Problem'', *Quart. Appl. Math. 16*, (1958), 87-90.

[Car78] B. Carre, *Graphs and Networks*, Clarendon Press, Oxford, 1978.

[ChH91] J. Cheiney and Y. Huang, ''Efficient Maintenance of Explicit Transitive Closures with Set-Oriented Update Propagation and Parallel Processing'', Tech. Report, Ecole Nationale Superieure des Telecommunications, Paris, 1991.

[CoM90] M. P. Consens and A. O. Mendelzon, ''GraphLog: a Visual Formalism for Real Life Recursion'', *Proc. 9th Symp. Principles of Database Systems*, 1990, 404-416.

[CoW87] D. Coopersmith and S. Winograd, ''Matrix Multiplication via Arithmetic Progressions'', *Proceedings of the 19th ACM Symposium on the Theory of Computing*, 1987, 1-6.

[CMW86] I. F. Cruz, A. O. Mendelzon and P. T. Wood, ''A Graphical Query Language Supporting Recursion'', *Proc. ACM SIGMOD Conf. on Management of Data*, 1986, 16-52.

[CMW88] I. F. Cruz, A. O. Mendelzon and P. T. Wood, ''G+: Recursive Queries Without Recursion'', *Proc. 2nd Int'l Conf. Expert Database Systems*, 1988, 355-368.

[CrN89] I. F. Cruz and T. S. Norvell, ''Aggregative Closure: An Extension of Transitive Closure'', *Proc. IEEE 5th Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1989, 384-393.

[DaA] S. Dar and R. Agrawal, ''Extending SQL with Generalized Transitive Closure'', *IEEE Trans. Knowledge and Data Engineering*, . to appear.

[DAJ91] S. Dar, R. Agrawal and H. V. Jagadish, ''Optimization of Generalized Transitive Closure Queries'', *Proc. IEEE 7th Int'l Conf. Data Engineering*, Tokyo, Japan, Feb. 1991.

[DaJ92] S. Dar and H. V. Jagadish, ''A Spanning Tree Transitive Closure Algorithm'', *IEEE 8th Int'l Conf. on Data Engineering*, Phoenix, Arizona, Feb. 1992, 2-11.

[DaS85] U. Dayal and J. M. Smith, ''PROBE: A Knowledge-Oriented Database Management System'', *Proc. Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, Islamorada, Florida, Feb. 1985, 103-137.

[DKO84] D. J. DeWitt, R. Katz, F. Olken, D. Shapiro, M. Stonebraker and D. Wood, ''Implementation Techniques for Main Memory Database Systems'', *Proc. ACM-SIGMOD 1984 Int'l Conf. on Management of Data*, Boston, Mass., June 1984, 1-8.

[Dij59] E. W. Dijkstra, ''A Note on Two Problems in Connection with Graphs'', *Numer. Math. 1*, (1959), 269-271.

[Ebe81] J. Ebert, ''A Sensitive Transitive Closure Algorithm'', *Information Processing Letters 12*, (1981), 255-258.

[Eng87] R. W. Engles, ''Structured Tables'', ANSI X3H2-87-331, Dec. 1987.

[EvK77] J. Eve and R. Kurki-Suonio, ''On Computing the Transitive Closure of a Relation'', *Acta Informatica 8*, (1977), 303-314.

[Flo62] R. W. Floyd, ''Algorithm 97: Shortest Path'', *Commun. ACM 5*, 6 (1962), 345.

[GKS91] S. Ganguly, R. Krishnamurthy and A. Silberschatz, ''An Analysis Technique for Transitive Closure Algorithms: A Statistical Approach'', *Proc. IEEE 7th Int'l Conf. Data Engineering*, Tokyo, Japan, Feb. 1991, 728-735.

[GGZ91] S. Ganguly, S. Greco and C. Zaniolo, ''Minimum and Maximum Predicates in Logic Programming'', *Proc. 10th Symp. Principles of Database Systems*, Denver, Colorado, May 1991, 154-163.

[Gel92] A. V. Gelder, ''The Well-Founded Semantics of Aggregation'', *Proc. 11th Symp. Principles of Database Systems*, San Diego, California, June, 1992, 127-138.

[GoK79] A. Goralcikova and V. Koubek, ''A Reduct and Closure Algorithm for Graphs.'', *Proceedings of the Int'l Conf. on Mathematical Foundations of Computer Science*, 1979, 301-307.

[Gua90] G. V. Guaardalben, ''The Transitive Closure and the Probabilistic Relational Data Model'', Unpublished Manuscript, 1990.

[GuY] K. Guh and C. T. Yu, ''Efficient Management of Materialized Transitive Closure in Centralized and Parallel Environments'', *IEEE Trans. Knowledge and Data Engineering*, . to appear.

[GKB87] U. Guntzer, W. Kiessling and R. Bayer, ''On the Evaluation of Recursion in Deductive Database Systems by Efficient Differential Fixpoint Iteration'', *Proc. IEEE 3rd Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1987, 120-129.

[HMM87] T. Harder, K. Meyer-Wegener, P. B. Mitschang and A. Sikeler, ''PRIMA - a DBMS Prototype Supporting Engineering Applications'', *Proc. 13th Int'l Conf. Very Large Data Bases*, Brighton, England, Sept. 1987, 433-442.

[HSS87] M. Hardwick, G. Samaras and D. L. Spooner, ''Evaluating Recursive Queries in CAD Using an Extended Projection Function.'', *Proc. IEEE 3rd Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1987, 138-149..

[HoK80] J. Hong and H. Kung, ''The Red-Blue Pebble Game'', *Proceedings of the 13th ACM Symposium on the Theory of Computing*, 1980.

[HuS93] K. A. Hua and J. X. W. Su, ''Efficient Evaluation of Traversal Recursion Using Connectivity Index'', *Proceedings 9th Int'l Conf. on Data Engineering*, Vienna, Austria, April 1993.

[IRW] Y. E. Ioannidis, R. Ramakrishnan and L. Winger, ''Transitive Closure Algorithms Based on Depth-First Search'', *ACM Trans. Database Syst.*, . to appear.

[Ioa86] Y. E. Ioannidis, ''On the Computation of the Transitive Closure of Relational Operators'', *Proc. 12th Int'l Conf. Very Large Data Bases*, Kyoto, Japan, Aug. 1986, 403-411.

[IoR88] Y. E. Ioannidis and R. Ramakrishnan, ''An Efficient Transitive Closure Algorithm'', *Proc. 14th Int'l Conf. Very Large Data Bases*, Los Angeles, California, Aug.-Sept. 1988.

[Jag90] H. V. Jagadish, ''A Compression Technique to Materialize Transitive Closure'', *ACM Transactions on Database Systems 15*, 4 (Dec. 1990), . (Previously appeared as: ''A Compressed Transitive Closure Technique for Efficient Fixed-Point Query Processing'' in Proc. 2nd Int'l Conf. on Expert Database Systems, Tyson's Corner, VA, 1988).

[Jak91] H. Jakobsson, ''Mixed-Approach Algorithms for Transitive Closure'', *Proc. 10th Symp. Principles of Database Systems*, Denver, Colorado, 1991, 199-205.

[Jak92] H. Jakobsson, ''On Tree-Based Techniques for Query Evaluation'', *Proc. 11th Symp. Principles of Database Systems*, 1992, 380-392.

[Jak93] H. Jakobsson, ''Mixed-Approach Algorithms for Transitive Closure'', Unpublished Manuscript, Computer Science Dept., Stanford Univ., Stanford, California, 1993.

[Jia90] B. Jiang, ''A Suitable Algorithm for Computing Partial Transitive Closures in Databases'', *Proc. IEEE 6th Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1990.

[Jia92] B. Jiang, ''I/O Efficiency of Shortest Path Algorithms: An Analysis'', IEEE 8th Int'l Conf. on Data Engineering., Phoenix, Arizona, Feb. 1992.

[Jia90] B. Jiang, ''Design, Analysis, and Evaluation of Algorithms for Computing Partial Transitive in Databases'', Computer Science Tech. Rep., ETH Zurich, June 1990.

[KIC92] R. Kabler, Y. E. Ioannidis and M. Carey, ''Performance Evaluation of Algorithms for Transitive Closure'', *Information Systems 17*, 5 (Sep. 1992), .

[Kar90] R. M. Karp, ''The Transitive Closure of a Random Graph'', *Random Structures and Algorithms 1*, 1 (1990), 73-93.

[KiL86] M. Kifer and E. L. Lozinskii, ''A Framework for an Efficient Implementation of Deductive Database Systems'', *Proc. 6th Advanced Database Symposium*, Tokyo, Japan, Aug. 1986.

[Kle56] S. C. Kleene, *Representation of Events in Neural Nets and Finite Automata*, Princeton University Press, 1956.

[Knu73] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.

[KuP87] S. M. Kuck and S. Pax, ''A Relational Calculus with Transitive Closure'', Technical Report, Univ. Illinois, Urbana, Illinois, 1987.

[LaD89] P. A. Larson and V. Deshpande, ''A File Structure Supporting Traversal Recursion,'', *Proc. ACM-SIGMOD 1989 Int'l Conf. Management of Data*, Portland, Oregon, May-June 1989, 243-252.

[Lin87] V. Linnemann, ''Non First Normal Form Relations and Recursive Queries: An SQL Based Approach'', *Proc. IEEE 3rd Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1987, 591-598.

[LiN89] R. J. Lipton and J. F. Naughton, ''Estimating the Size of Generalized Transitive Closures'', *Proc. of the 15th Int'l Conf. on Very Large Databases*, Amsterdam, the Netherlands, Aug. 1989, 165-172.

[LM87] H. Lu, K. Mikkilineni and J. P. Richardson, ''Design and Evaluation of Algorithms to Compute the Transitive Closure of a Database Relation'', *Proc. IEEE 3rd Int'l Conf. Data Engineering*, Los Angeles, California, Feb. 1987, 112-119.

[Lu87] H. Lu, ''New Strategies for Computing the Transitive Closure of a Database Relation'', *Proc. 13th Int'l Conf. Very Large Data Bases*, Brighton, England, Sept. 1987.

[LZH90] W. S. Luk, W. Zhang and J. Han, ''Path: An Approach to Incorporate List Processing in a Relational Database'', *2nd Int'l Conf. on Software Eng. and Knowledge Eng.*, Skokie, Illinois, 1990.

[Meh84] K. Mehlhorn, *Graphs Algorithms and NP Completeness*, Springer-Verlag, 1984.

[Mel92] J. Melton, (ed.), ''(ISO-ANSI Working Draft) Database Language SQL (SQL3)'', ANSI X3.135-1992, Nov. 1992.

[MeW89] A. O. Mendelzon and P. T. Wood, ''Finding Regular Simple Paths in Graph Databases'', *Proc. 15th Int'l Conf. Very Large Data Bases*, Amsterdam, The Netherlands, Aug. 1989, 185-194.

[Mit89] P. B. Mitschang, ''Extending the Relational Algebra to Capture Complex Objects'', *Proc. 15th Int'l Conf. Very Large Data Bases*, Amsterdam, The Netherlands, Aug. 1989, 297-305.

[Moh88] C. Mohan, Personal Communication, , 1988.

[MFP90] I. Mumick, S. Finkelstein, H. Pirahesh and R. Ramakrishnan, ''Magic Conditions'', *Proc. 9th Symp. Principles of Database Systems*, 1990, 314-330.

[MPR90] I. Mumick, H. Pirahesh and R. Ramakrishnan, ''The Magic of Duplicates and Aggregates'', *Proc. 16th Int'l Conf. Very Large Data Bases*, Brisbane, Australia, 1990.

[MFP90] I. Mumick, S. Finkelstein, H. Pirahesh and R. Ramakrishnan, ''Magic is Relevant'', *Proc. ACM-SIGMOD 1990 Int'l Conf. on Management of Data*, 1990, 247-258.

[MuP91] I. Mumick and H. Pirahesh, ''Overbound and Right-Linear Queries'', *Proc. 10th Symp. Principles of Database Systems*, Denver, Colorado, May 1991, 127-141.

[NaT89] S. Naqui and S. Tsur, ''A Logical Language for Data and Knowledge Bases'', in *Principles of Computer Science*, A. V. Aho and J. D. Ullman (ed.), Computer Science Press, New York, 1989.

[NRS89] J. F. Naughton, R. Ramakrishnan, Y. Sagiv and J. D. Ullman, ''Efficient Evaluation of Right-, Left-, and Multi-Linear Rules'', *Proc. ACM-SIGMOD 1989 Int'l Conf. Management of Data*, Portland, Oregon, May-June 1989, 235-242.

[PiA86] P. Pistor and F. Andersen, ''Designing A Generalized NF2 Model With an SQL-Type Interface'', *Proc. 12th Int'l Conf. on Very Large Databases*, Kyoto, Japan, Aug. 1986, 278-285.

[Pur70] P. Purdom, ''A Transitive Closure Algorithm'', *BIT 10*, (1970), 76-94.

[QHK89] G. Qadah, L. Henschen and J. Kim, ''The Efficient Processing of Instantiated Transitive Closure Queries'', Northwestern University Technical Memorandum, Evanston, Illinois, Feb. 1989.

[RSS92] R. Ramakrishnan, D. Srivastava and S. Sudarshan, ''Efficient Bottom-up Evaluation of Logic Programs'', in *The State of the Art in Computer Systems and Software Engineering*, J. Vandewalle (ed.), Kluwer Academic Publishers, New York, 1992.

[Ram88] R. Ramakrishnan, ''Magic Templates: A Spellbinding Approach to Logic Programs'', *Proc. 5th Int'l Conference on Logic Programming*, Seatlle, Washington, Aug. 1988, 140-159.

[RHD86] A. Rosenthal, S. Heiler, U. Dayal and F. Manola, ''Traversal Recursion: A Practical Approach to Supporting Recursive Applications'', *Proc. ACM-SIGMOD 1986 Int'l Conf. on Management of Data*, Washington D.C., May 1986, 166-176.

[Ros90] K. A. Ross, ''Modular Stratification and Magic Sets for DATALOG programs with negation'', *Proc. 9th Symp. Principles of Database Systems*, Nashville, Tennessee, April 1990, 161-171.

[RoS92] K. A. Ross and Y. Sagiv, ''Monotonic Aggregation in Deductive Databases'', *Proc. 11th Symp. Principles of Database Systems*, San Diego, California, June 1992, 114-126.

[RKB87] M. A. Roth, H. F. Korth and D. S. Batory, ''SQL/NF: A Query Langage For ¬1NF Relational Databases'', *Information Systems 12*, 1 (1987), 99-114.

[Sar89] Y. P. Saraiya, ''Linearizing Non-Linear Recursions in Polynomial Time'', *Proc. 8th Symp. Principles of Database Systems*, 1989.

[ScS86] H. J. Schek and M. Scholl, ''The Relation Model with Relation-Valued Attributes'', *Information Systems 11*, 2 (1986), .

[Sch83] L. Schmitz, ''An Improved Transitive Closure Algorithm'', *Computing 30*, (1983), 359-371.

[Sch78] C. P. Schnorr, ''An Algorithm for Transitive Closure with Linear Expected Time'', *SIAM J. Computing 7*, 2 (May 1978), 127-133.

[SeN91] S. Seshadri and J. F. Naughton, ''On the Expected Size of Recursive Datalog Queries'', *Proc. 10th Symp. Principles of Database Systems*, Denver, Colorado, 1991, 268-279.

[Sha88] P. Shaw, ''A Generalization of Recursive Expressions for Non-Linear Recursion of fixed Degree'', ANSI X3H2-88-93REV, April 1988.

[Sha87] P. Shaw, ''CLOSURE Expressions'', ANSI X3H2-87-330, Dec. 1987.

[SiS88] S. Sippu and E. Soisalon-Soininen, ''A Generalized Transitive Closure for Relational Queries'', *Proc. 7th Symp. Principles of Database Systems*, March 1988.

[SHS87] D. L. Spooner, M. Hardwick and G. Samaras, ''Some Conceptual Ideas For Extending SQL For Object-Oriented Engineering Database Systems'', *Proc. IEEE 1st Int'l Conf. on Data and Knowledge Systems for Manufacturing and Engineering*, Oct. 1987, 163-169.

[Sri93] D. Srivastava, ''Representing and Querying Complex Information in the CORAL Deductive Database System'', Univ. Wisconsin, Madison, Madison, Wisconsin, 1993. Ph.D. Dissertation.

[SrRar] D. Srivastava and R. Ramakrishnan, ''Pushing Constraint Selections'', *Journal of Logic Programming*, to appear. (Preliminary version appeared as: D. Srivastava and R. Ramakrishnan, ''Pushing Constraint Selections'', *Proc. 11th Symp. Principles of Database Systems*, San Diego, California, June 1992).

[Sto88] M. Stonebraker, Personal Communication, , 1988.

[SSR93] S. Sudarshan, D. Srivastava, R. Ramakrishnan and C. Beeri, ''Extending the Well-Founded and Valid Semantics for Aggregation'', *Proc. Int'l Logic Programming Symposium*, Vancouver, B.C., Canada, Nov. 1993.

[SuR91] S. Sudarshan and R. Ramakrishnan, ''Aggregation and Relevance in Deductive Databases'', *Proc. of the 17th Int'l Conf. on Very Large Databases*, Barcelona, Spain, Sept. 1991.

[Sul87] J. Sullivan, ''Tree Structured Traversal'', ANSI X3H2-87-306, Nov. 1987.

[Tar72] R. Tarjan, ''Depth-First Search and Linear Graph Algorithms'', *SIAM J. Computing 1*, (1972), 146-160.

[Tri82] K. S. Trivedi, *Probability and Statistics With Reliability, Queuing, and Computer Science Applications*, Prentice-Hall, 1982.

[Ull89] J. D. Ullman, *Database and Knowledge-Base Systems (Volume 2)*, Computer Science Press, 1989.

[UlY90] J. D. Ullman and M. Yannakakis, ''The Input/Output Complexity of Transitive Closure'', *Proc. ACM-SIGMOD 1990 Int'l Conf. on Management of Data*, Atlantic City.

[VaB86] P. Valduriez and H. Boral, ''Evaluation of Recursive Queries Using Join Indices'', *Proc. 1st Int'l Conf. Expert Database Systems*, Charleston, South Carolina, April 1986, 197-208.

[War75] H. S. Warren, ''A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations'', *Commun. ACM 18*, 4 (April 1975), 218-220.

[War62] S. Warshall, ''A Theorem on Boolean Matrices'', *J. ACM 9*, 1 (Jan. 1962), 11-12.

[Win92] L. Winger, ''Analysis of Depth-First Transitive Closure Algorithms'', Univ. Wisconsin, Madison, Madison, Wisconsin, 1992. M.S. Dissertation.

[Yan90] M. Yannakakis, ''Graph-Theoretic Methods in Database Theory'', *Proc. 9th Symp. Principles of Database Systems*, Nashville, Tennessee, April 1990, 230-242.

[Yel88] D. Yellin,, ''A Dynamic Transitive Closure Algorithm'', Technical Report, IBM, Yorktown Heights, Revised 6/20/88.

[ZYT90] W. Zhang, C. T. Yu and D. Troy, ''Necessary and Sufficient Conditions to Linearize Doubly Recursive Programs in Logic Databases'', *ACM Trans. Database Syst. 15*, 3 (Sep. 1990), 459-482.

Appendix 1

The BNF Syntax of SQL/TC

We present below the relevant part of the extended ANSI/SQL BNF syntax. The new clauses are printed in constant-width font. Existing clauses whose definition was changed are printed in italics. The other clauses are existing ANSI/SQL constructs that are given to clarify the ''context'' of our extensions.

<query specification> ::=
  SELECT [ALL | DISTINCT] <select list> <table expression>

<table expression> ::=
  <from clause>
  [<where clause>]
  [<group by clause>]
  [<having clause>]

<from clause> ::=
  FROM <table reference> [{ , <table reference> } ...]

<table reference> ::=
  <table name> [[AS] <correlation name> [(derived column list>)]]
  | <derived table> [AS] <correlation name> [(derived column list>)]
  | <joined table>

<derived table> ::=
  (<query expression>)

```
    | ( <closure expression> )

<closure expression> ::=
    <closure clause>
    [ <path label clause> ]
    [ <path selection> ]

<closure clause> ::=
    CLOSURE <closure predicate> OF <closure table>

<closure predicate> ::=
    <closure condition> [ { AND <closure condition> } ... ]

<closure condition> ::=
    <column expression> <comp op> NEXT <column expression>
    | <column expression> <comp op> <literal>
```

<column expression> ::= <value expression>

```
<closure table> ::= <table reference>
```

```
<path label clause> ::=
    WITH <path label specification list>

<path label specification list> ::=
    <path label specification>
    [ { , <path label specification> } ... ]

<path label specification> ::=
    <path label definition> [ <arc selection> ]

<path label definition> ::=
    <column name> = <value expression>

<arc selection> ::= <where clause>

<path selection> ::= <where clause>
```

<key word> ::= ... PATH ...

Appendix 2

Algebraic Formulation of Queries 1 - 13

To make the queries easier to read, we denoted the value of path labels and path-set labels using the notation

$Label := Expression$

Example 1:

$$\pi_{Src,Dest}\ \underset{Dest=Src}{\text{PATHS}}\ (Map)$$

Example 2:

$$\pi_{Src,Dest}\ \underset{Dest=Src\ \&\ Dest<>'Chicago'}{\text{PATHS}}\ (Map)$$

Example 3:

$$\pi_{Src,Dest}\ \underset{Dest=Src\ \&\ A\_Time \le D\_Time-1}{\text{PATHS}}\ (Map)$$

Example 4:

$$\pi_{Src,Dest}\ \sigma_{Src='NewYork'}\ \underset{Dest=Src}{\text{PATHS}}\ (Map)$$

Example 5:

$$\pi_{Src,Dest,Cost}\ \sigma_{Src='Paris'}\ Cost := \underset{Price}{\text{SUM}}\ \underset{Dest=Src}{\text{PATHS}}\ (Trains)$$

Example 6:

$$\pi_{Src,Dest,Cost}\ \sigma_{Src='Paris'}\ ED := \underset{Dist,Kind='Express'}{\text{SUM}},\ RD := \underset{Dist,Kind='Regular'}{\text{SUM}}\ \underset{Dest=Src}{\text{PATHS}}\ (Trains)$$

Example 7:

$$\pi_{Src,Dest,MN\_TD,MX\_MC}\ MN\_TD := \underset{TD}{\text{MIN}}, MX\_MC = \underset{MC}{\text{MAX}}\ \sigma_{Src='a'\ \&\ Dest='b'}$$
$$TD := \underset{Dist}{\text{SUM}},\ MC := \underset{Cap}{\text{MIN}}\ \underset{Dest=Src}{\text{PATHS}}\ (Roads)$$

Example 8:

$$\pi_{Subpart,Tot\_Qty}\ Tot\_Qty := \underset{Sub\_Qty,\ Subpart}{SUM}\ \sigma_{Part='a'}\ Sub\_Qty := \underset{Qty}{PRODUCT}\ \underset{Subpart=Part}{\text{PATHS}}\ (Assembly)$$

Example 9:

$$\pi_{Dest,MX\_AR}\ \delta_{MX\_AR>0.9}\ MX\_AR := \text{MAX}_{AR,Dest}\ \sigma_{Src='a'}\ AR := PRODUCT_{Reliability}\ \text{PATHS}_{Dest=Src}\ (Circuit)$$

Example 10:

$$\pi_{PATH}\ \sigma_{Src='Paris'\ \&\ Dest='Vancouver'\ \&\ KC>0}\ KC := \text{SUM}_{ALL,Airline='KLM'}\ \text{PATHS}_{Dest=Src}\ (Flights)$$

Example 11:

$$\pi_{PATH}\ \sigma_{Src='Paris'\ \&\ Dest='Vancouver'\ \&\ \exists(\sigma_{Dest='Toronto'} PATH)}\ \text{PATHS}_{Dest=Src}\ (Flights)$$

Example 12:

$$\pi_{Dest}\ \sigma_{Src='Paris'}\ \text{PATHS}_{Dest=Src}\ (\pi_{Src,Dest}\ Flights \cup \pi_{Src,Dest}\ \sigma_{Kind='Express'}\ Trains))$$

Example 13:

$$\pi_{PATH}\ \sigma_{Src='Paris'\ \&\ Dest='Vancouver'\ \&\ \text{COUNT}(\sigma_{Country='USA'} PATH \bowtie_{Dest=Name} City)>2}\ \text{PATHS}_{Dest=Src}\ (Flights)$$

Appendix 3

ANSI/SQL Formulation of Queries 1 - 13

Example 1:

```
SELECT Src, Dest FROM
   RECURSIVE TC
      INITIAL SELECT Map.Src, Map.Dest FROM Map
      ITERATION SELECT TC.Src, Map.Dest FROM TC, Map
         WHERE TC.Dest = Map.Src
```

Example 2:

```
SELECT * FROM
   RECURSIVE TC
      INITIAL SELECT Map.Src, Map.Dest FROM Map
         WHERE Dest <> 'Chicago'
      ITERATION SELECT TC.Src, Map.Dest FROM TC, Map
         WHERE TC.Dest = Map.Src
         AND TC.Dest <> 'Chicago'
```

or:

```
CREATE Map1(City_Name Src, City_Name Dest) ...

INSERT INTO Map1
SELECT Src, Dest FROM Map
   WHERE Dest <> 'Chicago'

SELECT * FROM
   RECURSIVE TC
      INITIAL SELECT Map1.Src, Map1.Dest FROM Map1
      ITERATION SELECT TC.Src, Map1.Dest FROM TC, Map1
         WHERE TC.Dest = Map1.Src
```

Example 3:

```
SELECT * FROM
  RECURSIVE TC
    INITIAL SELECT Flights.Src, Flights.Dest FROM Flights
    ITERATION SELECT TC.Src, Flights.Dest FROM TC, Flights
      WHERE TC.Dest = Flights.Src
      AND TC.A_Time ≤ Flights.D_Time - 1
```

Example 4:

```
SELECT * FROM
  RECURSIVE TC
    INITIAL SELECT Map.Src, Map.Dest FROM Map
      WHERE Map.Src = 'New York'
    ITERATION SELECT TC.Src, Map.Dest FROM TC, Map
      WHERE TC.Dest = Map.Src
```

Example 5:

```
SELECT Src, Dest, Cost FROM
  RECURSIVE TC(Src, Dest, Cost)
    INITIAL SELECT Trains.Src, Trains.Dest, Trains.Price FROM Trains
      WHERE Trains.Src = 'Paris'
    ITERATION SELECT TC.Src, Trains.Dest, TC.Cost + Trains.Price
    FROM TC, Trains
      WHERE TC.Dest = Trains.Src
```

Example 6:

```
SELECT Src, Dest, E_Dist, R_Dist FROM
  RECURSIVE TC(Src, Dest, E_Dist, R_Dist)
    INITIAL SELECT Trains.Src, Trains.Dest,
      IF Trains.Kind = 'Express' THEN Trains.Dist ELSE 0,
      IF Trains.Kind = 'Regular' THEN Trains.Dist ELSE 0
      FROM Trains
      WHERE Trains.Src = 'Paris'
    ITERATION SELECT TC.Src, Trains.Dest,
      IF Trains.Kind = 'Express' THEN TC.E_Dist + Trains.Dist ELSE TC.E_Dist,
      IF Trains.Kind = 'Regular' THEN TC.R_Dist + Trains.Dist ELSE TC.R_Dist
      FROM TC, Trains
      WHERE TC.Dest = Trains.Src
```

Example 7:

```
SELECT MIN ( TC.Tot_Dist ), MAX ( TC.Min_Cap ) FROM
  RECURSIVE TC(Src, Dest, Tot_Dist, Min_Cap)
    INITIAL SELECT Src, Dest, Dist, Cap FROM Roads
    ITERATION SELECT TC.Src, Roads.Dest,
      TC.Tot_Dist + Trains.Dist,
      IF TC.Min_Cap < Trains.Cap THEN TC.Min_Cap ELSE Trains.Cap
      FROM TC, Roads
      WHERE TC.Dest = Roads.Src
WHERE TC.Src = 'a' AND TC.Dest = 'b'
```

Example 8:

```
SELECT Subpart, SUM(Qty) FROM
  RECURSIVE TC(Part, Subpart, Qty)
    INITIAL SELECT Part, Subpart, Qty FROM Assembly
      WHERE Part = 'a'
    ITERATION SELECT TC.Part, Assembly.Subpart,
      TC.Qty * Assembly.Qty FROM TC, Assembly
      WHERE TC.Subpart = Assembly.Part
GROUP BY TC.Subpart
```

Example 9:

```
SELECT Dest, MAX(Acc_Rel) FROM
  RECURSIVE TC(Src, Dest, Acc_Rel)
    INITIAL SELECT Src, Dest, Reliability FROM Circuit
      WHERE Src = 'a'
    ITERATION SELECT TC.Src, Circuit.Dest,
      TC.Acc_Rel * Circuit.Reliability FROM TC, Circuit
      WHERE TC.Dest = Circuit.Src
GROUP BY TC.Dest
HAVING MAX(Acc_Rel) > 0.9
```

Example 10 (without retrieving path information):

```
SELECT Src, Dest FROM
  RECURSIVE TC(Src, Dest, KLM_connections)
    INITIAL SELECT Flights.Src, Flights.Dest,
      IF Flights.Airline = 'KLM' THEN 1 ELSE 0
      FROM Flights
    ITERATION SELECT TC.Src, Flights.Dest,
      IF Flights.Airline = 'KLM' THEN KLM_connections + 1
  ELSE KLM_connections
      FROM TC, Flights
      WHERE TC.Dest = Flights.Src
WHERE TC.Src = 'Paris' AND TC.Dest = 'Vancouver' AND KLM_connections > 0
```

Example 11 (without retrieving path information):

```
SELECT Src, Dest FROM
  RECURSIVE TC(Src, Dest, Toronto_Stops)
    INITIAL SELECT Flights.Src, Flights.Dest,
      IF Flights.Dest = 'Toronto' THEN 1 ELSE 0
      FROM Flights
    ITERATION SELECT TC.Src, Flights.Dest,
      IF Flights.Dest = 'Toronto' THEN Toronto_Stops + 1 ELSE Toronto_Stops
      FROM TC, Flights
      WHERE TC.Dest = Flights.Src
WHERE TC.Src = 'Paris' AND TC.Dest = 'Vancouver' AND Toronto_Stops > 0
```

Example 12:

```
SELECT Dest FROM
  RECURSIVE TC
    INITIAL SELECT Travel.Src, Travel.Dest FROM
      (( SELECT Src, Dest FROM Flights ) UNION
      ( SELECT Src, Dest FROM Trains WHERE Kind = 'Express')) AS Travel
      WHERE Travel.Src = 'Paris'
    ITERATION SELECT TC.Src, Travel.Dest FROM TC,
      (( SELECT Src, Dest FROM Flights ) UNION
      ( SELECT Src, Dest FROM Trains WHERE Kind = 'Express')) AS Travel
      WHERE TC.Dest = Travel.Src
```

or:

```
CREATE Travel(City_Name Src, City_Name Dest) ...

INSERT INTO Travel
( SELECT Src, Dest FROM Flights ) UNION
( SELECT Src, Dest FROM Trains WHERE Kind = 'Express' )

SELECT Dest FROM
  RECURSIVE TC
    INITIAL SELECT Travel.Src, Travel.Dest FROM Travel
      WHERE Travel.Src = 'Paris'
    ITERATION SELECT TC.Src, Travel.Dest FROM TC, Travel
      WHERE TC.Dest = Travel.Src
```

Example 13 (without retrieving path information, and with dubious syntax):

```
SELECT Src, Dest FROM
  RECURSIVE TC(Src, Dest, USA_Stops)
    INITIAL SELECT Flights.Src, Flights.Dest,
      IF Flights.Dest IN (SELECT Name FROM City WHERE Country = 'USA')
  THEN 1 ELSE 0
      FROM Flights
    ITERATION SELECT TC.Src, Flights.Dest,
      IF Flights.Dest IN (SELECT Name FROM City WHERE Country = 'USA')
      THEN USA_Stops + 1 ELSE USA_Stops
      FROM TC, Flights
      WHERE TC.Dest = Flights.Src
WHERE TC.Src = 'Paris' AND TC.Dest = 'Vancouver' AND USA_Stops > 2
```

Appendix 4

Classification of SQL/TC Selection Criteria

The following is a classification of the incrementally computable selections admitted by SQL/TC. This list is given here to illustrate the table-driven approach to classification of selections suggested in Section 4.3.6.

1. Selection on start node/arc — extension-monotonic.
2. Selection on consecutive arcs — monotonically negative.
3. Selection on all nodes/arcs — decomposable.
4. Selection on end node/arc — non-monotonic.
5. Selection on path label — written in the format:

   <aggregate function yielding a path label> {comparison operators}

   MAX, $\{<, \leq\}$: decomposable.
   MAX, $\{=\}$: non-monotonic.
   MIN, $\{>, \geq\}$: decomposable.
   MIN, $\{=\}$: non-monotonic.
   COUNT, $\{<, \leq\}$: monotonically negative.
   COUNT, $\{=\}$: non-monotonic.

   SUM: depends on the domain values

   For non-negative values we have:

   SUM, $\{<, \leq\}$: monotonically negative.
   SUM, $\{=\}$: non-monotonic.

   PRODUCT: depends on the domain values

   For values ≥ 1 we have:

   PRODUCT, $\{<, \leq\}$: monotonically negative.
   PRODUCT, $\{=\}$: non-monotonic.

   For (real) values in [0, 1] we have:

   PRODUCT, $\{>, \geq\}$: monotonically negative.
   PRODUCT, $\{=\}$: non-monotonic.

   AVG, $\{<, \leq, =, >, \geq\}$: non-monotonic.
   AND, $\{=\}$: decomposable.