On the Cost of Transitive Closures in Relational Databases

Zhe Li
Computer Science Department
Columbia University
New York, NY 10027
li@cs.columbia.edu

Kenneth A. Ross
Computer Science Department
Columbia University
New York, NY 10027
kar@cs.columbia.edu

Technical Report No. CUCS-004-93
February 1993

Abstract

We consider the question of taking transitive closures on top of pure relational systems (Sybase and Ingres in this case). We developed three kinds of transitive closure programs, one using a stored procedure to simulate a built-in transitive closure operator, one using the C language embedded with SQL statements to simulate the iterated execution of the transitive closure operation, and one using Floyd's matrix algorithm to compute the transitive closure of an input graph. By comparing and analyzing the respective performances of their different versions in terms of elapsed time spent on taking the transitive closure, we identify some of the bottlenecks that arise when defining the transitive closure operator on top of existing relational systems. The main purpose of the work is to estimate the costs of taking transitive closures on top of relational systems, isolate the different cost factors (such as logging, network transmission cost, etc.), and identify some necessary enhancements to existing relational systems in order to support transitive closure operation efficiently. We argue that relational databases should be augmented with efficient transitive closure operators if such queries are made frequently.

On the Cost of Transitive Closures in Relational Databases

Zhe Li
Computer Science Department
Columbia University
New York, NY 10027
li@cs.columbia.edu

Kenneth A. Ross
Computer Science Department
Columbia University
New York, NY 10027
kar@cs.columbia.edu

April 1993

Abstract

We consider the question of taking transitive closures on top of pure relational systems (Sybase and Ingres in this case). We developed three kinds of transitive closure programs, one using a stored procedure to simulate a built-in transitive closure operator, one using the C language embedded with SQL statements to simulate the iterated execution of the transitive closure operation, and one using Floyd's matrix algorithm to compute the transitive closure of an input graph. By comparing and analyzing the respective performances of their different versions in terms of elapsed time spent on taking the transitive closure, we identify some of the bottlenecks that arise when defining the transitive closure operator on top of existing relational systems. The main purpose of the work is to estimate the costs of taking transitive closures on top of relational systems, isolate the different cost factors (such as logging, network transmission cost, etc.), and identify some necessary enhancements to existing relational systems in order to support transitive closure operation efficiently. We argue that relational databases should be augmented with efficient transitive closure operators if such queries are made frequently.

1 Introduction

It has been argued in [1, 9] that *declarativeness* and stronger expressive power are important features of deductive database systems. Declarative formulations of queries allow for a clear expression of exactly *what* should be the answer to a query instead of *how* to get the answer tuples. In [1] it was shown that the least fixpoint operation is not expressible by relational algebra and relational calculus. Typical relational query languages (such as SQL/QUEL) have the favorable property of declarativeness but lack the expressive power for the least fixpoint operation. In the real world a variety of database applications require some kind of fixpoint operations. For example: In an airline reservation system, a customer would like to know the least expensive flight schedule from one city to another. In a network analysis system one may wish to determine whether two network nodes are connected, and the least loaded connection is used for routing decisions. In an object-oriented database, one may wish to check properties of all subclasses of a parent class. None of these queries can be couched directly in relational algebra. In order to answer these queries the user has to write complex application programs (usually by embedding SQL/QUEL statements into a host programming language) and provide a procedural evaluation algorithm. Aside from the impedance mismatch problem (incompatible data types and separate compiler optimizations) and loss of declarativeness (an ad hoc evaluation algorithm has to be chosen and specified procedurally), this paradigm also suffers a severe performance penalty. We quantify this performance penalty in this paper.

It was noted in [8] that attempting to build a deductive database system on top of an existing relational system is a mistake. In a relational database, the relations are typically large in size and persistent in nature. In deductive database things are just the opposite. Most least fixpoint query evaluation algorithms try to simulate sideways information passing [9] to restrict redundant computation. These algorithms usually create some supplementary relations to pass bindings and these relations are typically small and temporary in nature. If we use a relational system backend for writing a least fixpoint program, all these supplementary relations would have to be materialized (stored to disk) during each iteration of the program. Besides suffering this thrashing disk I/O and temporary storage overhead, the relational system will also perform concurrency control (such as locking) and logging on these relations. Again this incurs both CPU, I/O and storage overhead. In summary, a variant of the mismatch problem also occurs between the front end (application programs consist of SQL/QUEL statements and host language flow-of-control facilities) and the backend (SQL server).

This paper tries to further examine the problem of augmenting a relational database system to a deductive database system, and to quantify the performance overhead incurred by this approach. The experiment is done using Sybase and Ingres. We consider the question of taking transitive closures on top of pure relational systems (Sybase and Ingres in this case). We developed three kinds of transitive closure programs, one using a stored procedure[1] to simulate a built-in transitive closure operator, one using the C language embedded with SQL statements to simulate the iterated execution of a transitive closure operation, and one using the version of Floyd's matrix algorithm presented in [6] to compute the transitive closure of an input graph.

Floyd's algorithm will serve as an "ideal" algorithm for performing the transitive closure. By seeing how far from the ideal the results for transitive closures involving Ingres and Sybase come, we can demonstrate the presence of a significant performance gap.

By comparing and analyzing the different systems' respective performances in terms of elapsed time spent on taking the transitive closure, we identify some of the bottlenecks that arise when defining the transitive closure operator on top of existing relational systems. The main purpose of the work is to estimate the costs of taking transitive closures on top of relational systems, isolate the different cost factors (such as logging, network transmission

[1] A stored procedure is a precompiled collection of SQL statements, often including control-of-flow language.

cost, etc.) and identify some necessary enhancements to existing relational systems in order to support transitive closure operation efficiently. Furthermore, we try to come up with an estimated cost model and more efficient implementation strategies for our future research.

The remainder of the paper is organized as follows. Section 2 discusses the basic algorithm, the sample database we use, and the environment in which we implement the transitive closure operation. Section 3 presents the rationale of our experiment and the problems that we anticipate. Section 4 describes three kinds of a program (Sybase, Ingres and Floyd's matrix algorithm) for taking the whole transitive closure. Different processes of executing these three programs are briefly described. The respective performances of their different versions in terms of elapsed time are then compared and analyzed, logging and network costs are also isolated and contrasted. Section 5 describes the same three versions of programs for taking the single source version of transitive closure. Their relative performances of different versions are analyzed as before. Section 6 lists some other cost factors involved when performing transitive closure operations. In section 7 we briefly survey some relevant research done in this field. Finally in section 8 we discuss the conclusions we draw from this experiment and some practical issues related to taking transitive closure operation.

2 Algorithm, Sample Database and Environment

In the context of a deductive database, many techniques have recently been proposed for optimizing recursive queries [3, 2, 4]. The query evaluation method adopted in our experiment is "semi-naive" evaluation [3].

The "semi-naive" evaluation algorithm can be depicted as follows:

$$\begin{array}{l}
\Delta tc(X,Y) = e(X,Y) \\
tc(X,Y) = \Delta tc(X,Y) \\
\texttt{repeat} \\
\Delta Q(X,Y) = \Delta tc(X,Y) \\
\Delta tc(X,Y) = \Delta Q(X,Y) \bowtie e(X,Y) \\
\Delta tc(X,Y) = \Delta tc(X,Y) - tc(X,Y) \\
tc(X,Y) = tc(X,Y) \cup \Delta tc(X,Y) \\
\texttt{until } \Delta tc(X,Y) = \phi
\end{array}$$

From the algorithm we note that two temporary relations ΔQ and Δtc are introduced to reduce redundant computation and perform duplicate elimination.

For simplicity [2] we choose our sample database with only one binary relation "family" in it. The relation "family" has the "chain" property, i.e. , it is of the form:

[2] Different graph structures for the base relation will surely affect the cost of the evaluation of transitive closure. However, the overheads that we are trying to measure are incurred independent of the graph structure.

parent	child
1	2
2	3
3	4
4	5
5	6
$\vdots$	$\vdots$
n	n+1

family

Suppose the number of tuples in this chain relation is n, then we can easily calculate the size of its whole transitive closure as follows:

$$|TC(family(parent, child))| = n + n - 1 + \cdots + 1 = n * (n + 1)/2 = \Theta(n^2)$$

.

The size of the single source version of this transitive closure (i.e., tell me all nodes reachable from node i) is:

$$|\sigma_{parent=i} TC(family(parent, child))| = n + 1 - i = \Theta(n)$$

.

The relational database management systems we used are Sybase and Ingres. Sybase provides a comprehensive set of **DB-Library routines**[3] that application programs can call to perform various tasks. Sybase also provides a set of built-in flow-of-control statements as part of its "Transact-SQL" language (an extension of SQL). So the transitive closure operation can be easily implemented as a stored procedure as well. The same functions are also provided by Ingres.

The typical way of using DB-Library routines is to embed SQL statements into a host programming language such as C. The application sets up a connection with the SQL server, then assembles the necessary SQL commands in the system command buffer and sends the command batch to the SQL server for execution. The results are returned in a tuple-at-a-time fashion. The batch SQL commands are interpreted, e.g., parsing and optimization are done for each command batch. So the body of a "while" loop using a DB-library routine would be parsed and optimized many times.

For a **stored procedure**, things are quite different. During the creation of the procedure, the SQL statements are precompiled (parsed and optimized). When the user invokes the procedure, the "isql" frontend simply sends the query "execute procedure" to the SQL backend and waits. The SQL backend then executes the precompiled program code. After it finishes, the backend returns a status code to the front end and some result tuples (if any) for further processing. Stored procedures avoid repetitive parsing and query planning

[3] A set of C routines and macros that allow applications to interact with the SQL Server. It includes routines that send Transact-SQL commands to the SQL Server and others that process the results of those commands.

overhead, save the iterated transmission overhead of sending batch SQL commands, and are usually much faster than doing the same thing using embedded SQL facilities.

Ingres system 6.4 has the additional feature of "set nologging", so we can use this option to turn off system logging and isolate the logging cost factor. By comparing the results produced by programs running on a local machine (the same site as the server machine) and on a remote machine connected through an Ethernet, we can also isolate the network cost factor[4].

A deductive database system would perform efficient caching on the intermediate results. It would also reduce the repetitive parsing and optimization overhead, reduce the useless logging and locking on these temporary relations and eliminate the iterated cost of sending SQL command packets across the network (expensive IPC calls) due to its built-in transitive closure operator. The dominant cost factor would be the join cost. Floyd's algorithm's behavior can be made similar to that of a deductive database system in the sense that join cost can be made dominant in program's total elapsed time. Any data reading or writing done inside Floyd's algorithm's main loop can either be eliminated (assuming the allocated main memory is large enough to accommodate all the data) or cached (the UNIX operating system provides block I/O buffering).

We intend to measure the total elapsed time (including CPU time, I/O time and network time) spent by these programs. We then use these results to develop the cost formulas and expose some critical cost factors in performing the transitive closure operation.

The experiment is done using Ingres 6.4 and Sybase 4.2. We use a client-server model where the client and server machines are both Sun Sparc 600MP machines running SunOS 4.1.1 (except for the caching and indexing experiment, with the Sybase server running on a Sparc station 2 and SunOS 4.1.3). We have endeavored to make sure that the database server was always lightly loaded during our experiment. The raw Ethernet transmission speed is 10Mb/s. The machines connected by the Ethernet share NFS filesystems. The time in our experiments is measured using the "getrusage()" system call and the granularity is taken in seconds. [5]

3 Rationale and Problem Envisioned

The Database Systems

There are various cost factors that affect the execution of transitive closure programs. For instance:

- CPU time spent on relational operations such as join.
- CPU time for repetitive locking.
- Query parsing and planning time.

[4]Note that running remotely won't necessarily be too different from running locally since the interprocess communication (IPC) overhead has a fixed cost (overhead involved in executing the system calls read/write and the setup time involved in moving any amount of data between the user process and the kernel), along with a variable cost that is proportional to the size of data being transmitted.

[5]Our code for the Sybase and Ingres queries are not written using the same set of SQL statements. In particular, some system-specific enhancements are used (such as the "SELECT INTO" statement provided by Sybase that requires less logging). So a direct comparison between their relative performance is not fair.

- I/O time for scanning the participating relations to generate new tuples, storing the temporary relations and system logging. Because frequent updates are performed on the participating relations, the log size grows quadratically for the whole transitive closure case.

- Network setup and packet transmission (round-trip) overhead due to the client server DBMS architecture [6], i.e., to send SQL batch commands during each iteration to the server and get results back from the server for termination detection. Note that the network cost is usually not too big unless the load on the network is very high. The reason why it is sometimes comparable to disk access latency is due to the queuing delay and data encoding/decoding overhead.

- Data conversion overhead (due to the impedance mismatch between the data types of SQL and host programming language). We also have to do data encoding and decoding when running on heterogeneous machines.

The overhead of using SQL library routines inside a C program involves network setup and transmission time. i.e. the client sends the batch command packets to the server, then gets the answer tuples and status code of the query back to the client side. If the program iterates n times, we end up with $n * (network\_cost\_per\_batch + one\_status\_code\_cost)$ which is $\Theta(n)$ cost. Since during each iteration the queries are optimized separately, it is hard to achieve global optimization for the whole program. Compared with a built-in transitive closure operator, we lose global optimization for executing the program. Other overheads are repeated parsing and query re-planning done by the server for the same set of SQL batch commands during each iteration of "semi-naive" evaluation.

Using the stored procedure, little network overhead is incurred. Also the query program is parsed and optimized prior to its execution. However, both Sybase and Ingres have some inner constraints on creating a stored procedure. For instance, within a stored procedure, one cannot create an object (including a temporary table), drop it, and then create a new object with the same name. Basically all the objects referenced inside a stored procedure have to exist beforehand due to compilation and optimization requirements. The relevant relations' names are hard-coded into the procedure. Thus we can't make our transitive closure program generic. This constraint also has the implication that we can't use "SELECT INTO" statement (Sybase) to reduce some logging overhead. Instead, we have to "TRUNCATE" the ΔQ table each time, "SELECT" the newly derived tuples and "INSERT" into ΔQ. Note that the relations inside the main loop are tightly coupled, that is: just after one temporary relation is created, it is immediately used in a join operation, then the tuples in it are deleted.

The drawback of a stored procedure is its non-declarativeness and difficulty to optimize when loops are present. Because of compiling considerations, the base relation names are hard-coded inside the program, thus the program is not reusable. The user is responsible for providing a procedural, efficient program to evaluate recursion. However, the efficiency and appropriateness of an algorithm depends on system maintained statistics (such as size of base relation, available indexes, buffer replacement policy, size of buffer cache etc.). Also the user has to specify the procedural steps of a particular evaluation algorithm. In contrast, a

[6] Note that if the server runs on the same machine as the application program, then the network is no longer an issue. Our experiment is designed such that the client program and the SQL server run on different machines connected through an Ethernet LAN. In this scenario, network cost is not negligible.

deductive database system can choose dynamically which optimization or rewriting technique be used on different recursive programs. This flexibility is especially useful for single source or aggregation recursive queries.

From the "semi-naive" evaluation algorithm, we can see that the size of ΔQ and Δtc are typically smaller than those of base relations, especially when we are dealing with the single source case (if the average outgoing degree of the graph is large, then sometimes Δs can be larger than base relations) and tuples in ΔQ and Δtc are very short-lived (refreshed after each iteration). But in a relational system ΔQ and Δtc would be treated just the same as other base relations. System tasks such as concurrency control (expensive locking) and logging are indiscriminately performed on them as well during each iteration. Basically the execution patterns of these Δ relations are: "First they are created and stored. After the system performs some routine functions on them, they are immediately loaded for the next-round join operation. After the join these Δ relations are destroyed.". Each iteration is characterized by frequent updating on the participating relations. Note that during each iteration these relations are logged twice, once during insertion, once during deletion. If these Δs are larger than base relation, then the logging overhead on them might be comparable to the join cost, becoming dominant in the overall cost. Generally these Δ relations don't require concurrency control (such as locking) since they are private and transient to the transitive closure program that creates them. No other programs will try to access them. Logging is also unnecessary since these Δs are very short-lived and temporary and can be fully created dynamically. If the system crashes while updating these Δ relations, they can be simply discarded since there is no need to recover them. Logging on them not only requires a large amount of disk storage, but also consumes a lot of I/O bandwidth. It would be much more cost effective if these Δ relations could be cached in main memory, and logging on them turned off. One important database tuning technique is to put logs on a separate disk device to avoid I/O bandwidth contention and minimize some possible incurred seek overhead. The same technique can also be applied to those temporary relations. If those temporary relations can't entirely fit into main memory, it would be better to store them on a third disk device distinct from those of log and main database. But most existing relational systems don't support this specific requirement[7]. Both the C version and the stored procedure version do not achieve *global* optimization.

Another observation is that by using index on the join relations, implementing more efficient join method such as hybrid-hash join and allocating a larger buffer cache, we could reduce the join cost significantly, but the logging overhead and the network overhead would stay constant, thus forming an overall bottleneck. If the buffer size is made large enough to hold all the participating relations, the only I/O operations incurred are by system logging.

Among the different cost factors, the query re-parsing and re-planning and network overhead are linear terms in the number of iterations, which is n, because during each iteration the same set of SQL command packets are parsed, optimized and sent across the network. The logging cost would be proportional to the number of tuples generated, with a factor greater than 3 (due to copying, inserting and deleting the generated tuples). For the whole transitive closure case, the logging cost would be roughly $\Theta(n^2)$. For the single source version, it is roughly $\Theta(n)$.

[7] Sybase allows you to put relations on a segment on a different device, but no special treatment on the relation otherwise. For taking single source which is considered a more common operation, Δs would usually fit in memory. But existing system don't provide "pinning" operations (except DB2) for relations.

Using Floyd's Algorithm

For programs computing the whole transitive closure of an input graph using Floyd's algorithm, we can assume two different models: the "main-memory model" and the "secondary-storage model". A program assuming the "main-memory model" would read the input data once into memory and perform all the join operations (matrix operations in this case) in main memory. The results are written to disk only once when the program terminates. The program's elapsed time would give a referenced estimation of worst-case CPU cost spent on the join operations. Our estimation is $\Theta(n^3)$ (n is the dimension of the adjacency matrix of input graph) since that is the asymptotic complexity of Floyd's method.

Programs assuming the "secondary-storage model" could roughly simulate the DBMS execution model (such as logging and buffering) by reading the data into memory, performing one iteration on them, writing the intermediate results back to the disk after each iteration, and reading the data back immediately at the beginning of the next iteration. We can simulate this behavior by adding some (extraneous) statements to write out and read back in the intermediate matrix at each step.

Note that I/O cost taken by programs assuming "secondary-storage model" would be $\Theta(n^3)$ because the matrix size is $\Theta(n^2)$, and we want to write the matrix back to the disk after each program iteration. However, because many entries in the matrix will fit in a single disk block, the contribution of this cubic term is inversely proportional to the blocking factor. For the single source version of Floyd's method, we can play the trick of storing only the affected row of the adjacency matrix to the disk during each iteration. In this case, the total I/O cost is $\Theta(n^2)$.

There are two variants of programs assuming the "secondary-storage model". The first variant would run on the local machine but the writes are not synced at the end of each iteration. This way, the UNIX operating system would cache the intermediate data in memory and save most (possibly all) of the iterated disk I/O operations. This execution pattern simulates the evaluation behavior of a deductive database system. The actual I/O cost would be small compared with the join cost (matrix multiplication). The caching benefit (I/O buffering) can be measured. The program's elapsed time would be an estimation of CPU join cost plus some join incurred disk I/O cost (without logging overhead). This would give a referenced estimation of an ideal implementation of transitive closure operation. We expect its elapsed time would be comparable to that of the "main-memory model" version.

The second variant would also run on the local machine where the data file resides, reading/writing data iteratively from/to the data file, but with writes being synced to disk after each iteration. This will simulate a pure relational system's way of taking transitive closures (thrashing I/O and logging). We expect it to be the most expensive one.

For the single-source version of the program we use a simplified version of Floyd's method with $\Theta(n^2)$ complexity. (This routine only calculates reachability in one row of the matrix.) The cost estimations for the single source versions of programs using Floyd's algorithm can then be deduced similarly. We omit them here for simplicity.

4 Results for Taking Whole Transitive Closures

The following is a presentation of the results for taking the whole transitive closure of relation "family". They are produced by the C program/ stored procedure versions running on

Ingres/Sybase, and by Floyd's matrix algorithm using plain UNIX files as direct data storage. Due to space limitations, the program listings are omitted.

The x-coordinates of the graphs are the number of tuples in the "family" table, the y-coordinates are the elapsed time measured in seconds spent by different programs.

Figure 1 is the graph for taking the whole transitive closure on top of Ingres. The curves correspond to the C program involving network and logging overhead (the dashed curve with small equal-length segments), the C program with logging overhead but without network overhead (the solid curve), the C program without logging and network overhead (the dashed curve with large equal-length segments) and stored procedure without network and logging overhead (the dot-dashed curve) respectively. Note that the stored procedure performs slightly worse when the size of "family" increases, the reason is that inside the loop of the stored procedure we have SQL statements that update the relations frequently, thus it is very difficult for the system to come up with an optimized plan based on the initial static statistics. Although using stored procedure saves some parsing and optimization overhead, a worse global plan would easily compromise this benefit. Also note that there is little performance difference between different versions of programs, the reason is that the dominant join cost overshadows the other overheads.

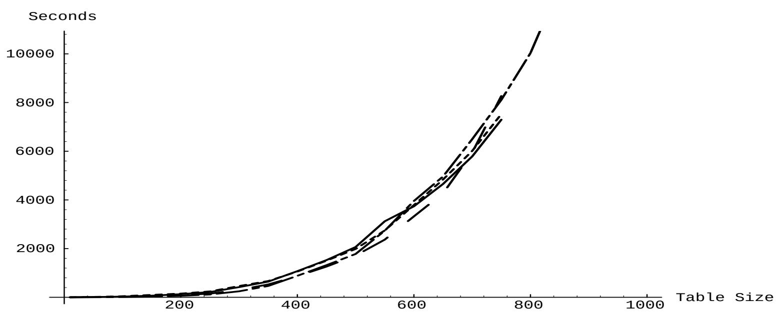

Figure 1: The Graph for taking whole transitive closure Ingres_versions

Figure 2 is the graph for taking the whole transitive closure on top of Sybase. The curves correspond to the C program involving network overhead (the dashed curve with small equal-length segments), C program without network overhead (the solid curve), and stored procedure without network overhead (the dashed curve with large equal-length segments) respectively in decreasing slope order. Since Sybase doesn't allow turning off system logging, all these three programs involve logging overhead in their results. We could observe that the network overhead is not negligible anymore, and there is also obvious performance difference between using a C program vs. using a stored procedure to perform transitive closure operation.

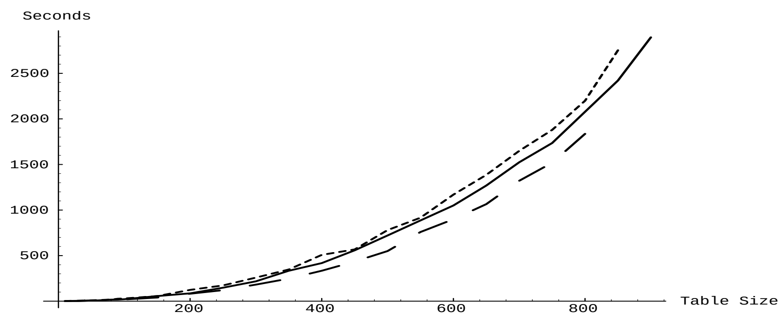

Figure 2: The graph for taking whole transitive closure Sybase_versions

Figure 3 is the graph for taking the whole transitive closure using Floyd's algorithm. The curves correspond to the C program simulating committed disk writes (writes being synced to disk during each iteration), C program with read()/write() system calls inside the program loop but writes are not synced (i.e., the program takes advantage of UNIX I/O buffering), C program without I/O operations done inside the program loop (i.e., assuming a main memory join model), respectively in decreasing slope order. We could observe that synchronous disk writes is the major bottleneck, increasing the response time by orders of magnitude.

Figure 4 is the graph for cross-comparisons: The dashed curve with small equal-length segments correspond to the C program taking the full transitive closure on Ingres. The solid curve corresponds to the C program taking the full transitive closure on Sybase. The dashed curve with large equal-length segments corresponds to the Floyd's algorithm with cached I/O cost (i.e., reading/writing data is performed but not synced to the disk). The dot-dashed curve corresponds to the Floyd's algorithm with pure CPU cost (i.e., no reading/writing data is performed inside the algorithm). We could observe a significant performance gap between an ideal implementation (avoiding those unnecessary overheads) and a relational database implementation of the transitive closure operation.

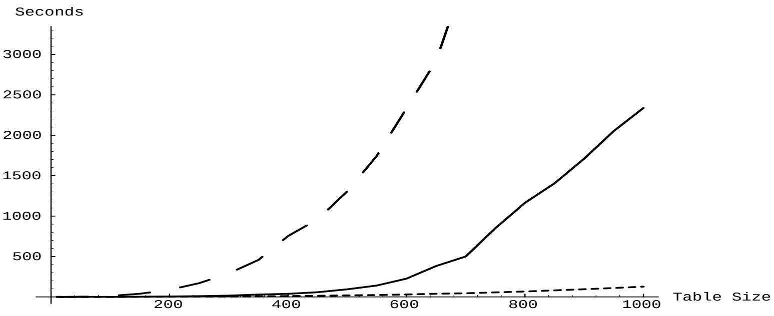

Figure 3: The graph for taking whole transitive closure Floyd_versions

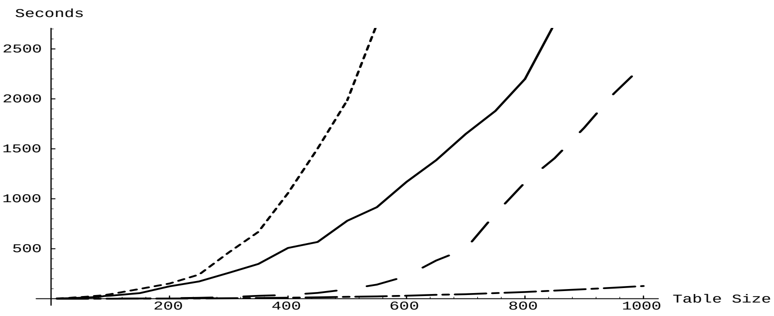

Figure 4: The graph for taking whole transitive closure cross_versions

4.1 Observations

From the above Ingres and Sybase graphs, We could observe that when the database size gets larger, the join cost will dominate the total elapsed time. Note that usually the curves will still be order-preserving (in the order of absolute elapsed time taken by them) since logging and network cost always contribute to the elapsed time. The curves for program involving network and logging overhead will always be higher than the curves of the program without these overheads.

The observations we got from the whole transitive closure experiment are as follows:

1. From our experiment, we observe that the network overhead is bigger than the logging overhead when the network load is high (due to space limitations, the data is not included) but grows slowly. Both costs are only a small portion of the total elapsed time spent in taking the whole transitive closure. This suggests that join cost is still dominant in the whole transitive closure operation.

2. The stored procedure is much faster than the C program when the size of "family" is small. The reason is that the procedure is precompiled, so the $(\#iterations) * (parsing + query\_planning\_per\_batch)$ is avoided. Since there is no need to send the query command during each iteration, $(\#iterations) * (network\_cost\_per\_batch)$ overhead is also avoided. But for larger size of "family", the C program runs faster on Ingres. Although using stored procedure suffers less parsing and query planning overhead, it is usually difficult to give a global cost-effective query plan at compile-time for a complex procedure (note the dynamic nature of those temporary relations). For the C program, the query optimizer has more accurate profile information about the participating relations during each iteration, thus the cumulative join cost savings becomes larger when the size of the base relation gets bigger.

3. From our experiment done on Sybase we also observe that the number of disk writes is always much bigger than the number of disk reads. This is partially due to the caching facility of Sybase (most reads are satisfied in the memory cache). The other reason is frequent updates performed on those intermediate relations. Since those updates are always logged, a lot of small disk writes are generated. Since logging has the blocked semantics, the introduced delay greatly affects the overall execution time of the transitive closure program. Because of frequent logging activity, the system log size grows quite rapidly. In our experiment, when the database size reaches 800 tuples, our 64MB disk space is quickly consumed. This not only consumes a lot of disk space, but also imposes disk bandwidth contention.

4. When the database size gets bigger (for Sybase), the amount of I/O time (including logging time) exceeds the CPU time. This suggests that the major cost component in a data-intensive database program is still the I/O cost.

5 Results for Single Source Transitive Closure

To simplify the discussion, we choose to compare the time spent by computing

$$\sigma_{parent=1} TC(family(parent, child))$$

So the query size is n.

Figure 5 is the graph for taking the single source of whole transitive closure on top of Ingres. The curves correspond to the C program involving network and logging overhead (the dashed curve with small equal-length segments), C program with logging overhead but without network overhead (the solid curve), C program without logging and network overhead (the dashed curve with large equal-length segments) and stored procedure without network and logging overhead (the dot-dashed curve) respectively in decreasing slope order.

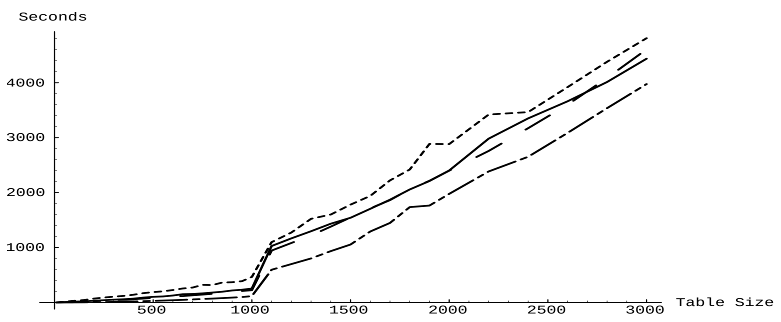

Figure 5: The Graph for taking single source of transitive closure Ingres_versions

Now the performance difference between different versions of programs become more obvious. The reason why there is a uniform bump in the cost curves when the number of tuples in "family" reaches 1000 is probably because the memory buffer size is exceeded.

Figure 6 is the graph for taking the single source of whole transitive closure on top of Sybase. The curves correspond to the C program involving network and logging overhead (the dashed curve with small equal-length segments), C program with logging overhead but without network overhead (the solid curve), and stored procedure without network (the dashed curve with large equal-length segments) respectively in increasing slope order. From the graph we could see that the effect of network cost is significant when the table size gets larger.

Figure 7 is the graph for taking the single source of transitive closure using Floyd's algorithm. The curves correspond to the C program simulating committed disk writes (writes are synced to disk during each iteration), C program with read()/write() system calls inside the program loop but writes are not synced (i.e., the program really takes advantage of UNIX I/O buffering), C program without I/O inside the program loop (i.e., assuming a main memory join model), respectively in decreasing slope order.

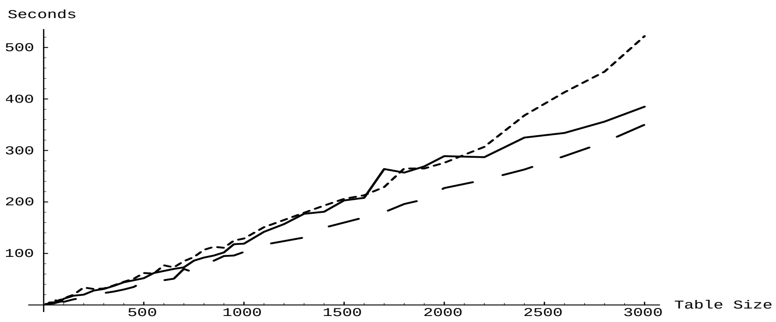

Figure 6: The Graph for taking single source of transitive closure Sybase_versions

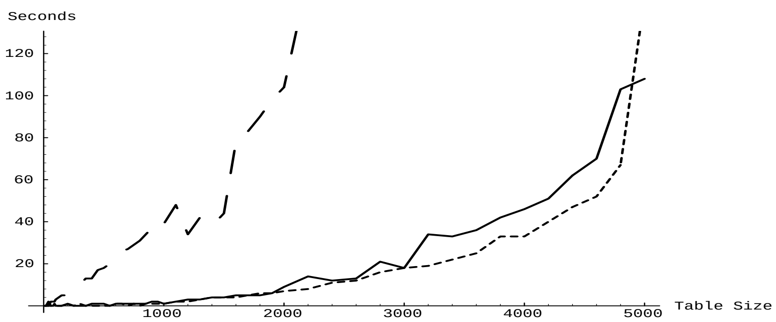

Figure 7: The graph for taking single source of transitive closure Floyd_versions

Figure 8 is the graph for cross-comparisons: The dashed curve with small equal-length segments correspond to the C program taking the single source transitive closure on Ingres. The solid curve corresponds to the C program taking the single source transitive closure on Sybase. The dashed curve with large equal-length segments corresponds to the Floyd's algorithm with cached I/O cost (i.e., reading/writing data is performed but not synced to the disk). The dot-dashed curve corresponds to the Floyd's algorithm with pure CPU cost (i.e., no reading/writing data is performed inside the algorithm). Note the significant differences between the slopes corresponding to Ingres/Sybase curves and those corresponding to Floyd's curves.

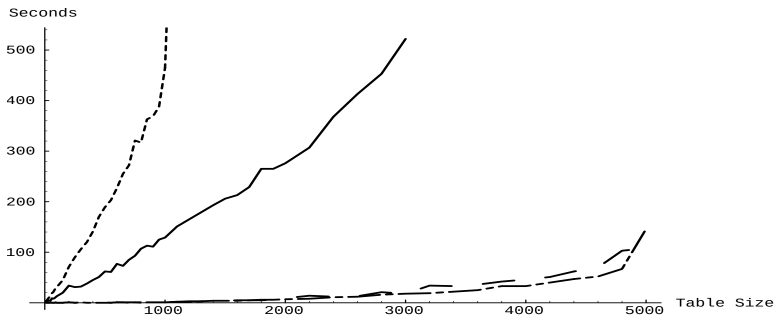

Figure 8: The graph for taking single source of transitive closure cross_versions

The observations from Section 4.1 also hold in the single-source case, with logging and network cost being more significant due to the decrease of overall join cost.

6 Other cost factors

The effect of caching can be seen from figure 9 and 10. Both curves are results of C program running on Sybase, with index built on the join columns of base relation "family". The dashed curve is configured with cache size 4.8MB, and the solid curve is configured with 12MB cache.

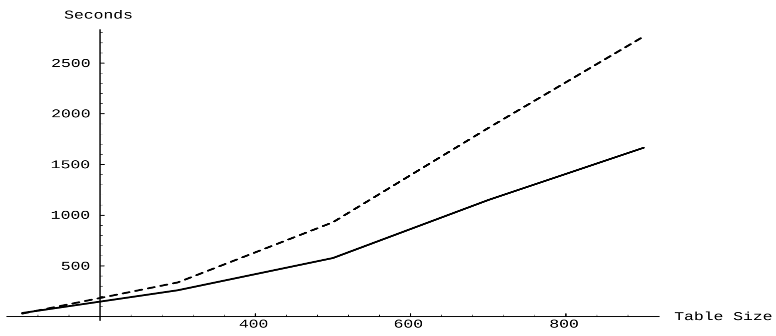

Figure 9: Caching effect in taking the whole transitive closure

Note that caching helps a lot in the whole transitive closure case, since the dominant join cost can be reduced dramatically if most of the relations can be cached in memory. However, caching has little effect for the single source case since the size of those Δ relations are very small in our example (one tuple in size during each iteration), a smaller cache is already enough to hold all the relations in memory.

The effect of clustered index on the join attribute of base relation "family" can be see from figure 11 to 12. The results are C program running on Sybase with a cache size of 12MB. The dashed curves corresponds to the indexed version, and the solid curve corresponds to the non-indexed version.

Note that for the whole transitive closure case, both joining relations are large ("family" and Δs), indexing on the join attributes improves the performance significantly, and the performance gap goes up when we increase the size of the base relation. Using an index on the base relation is especially beneficial since no updates are performed on it, thus we can avoid the penalty of slower updates and extra logging on the index. For the single source case, the Δ relations are very small, thus the saving of join cost achieved by indexing is not that obvious.

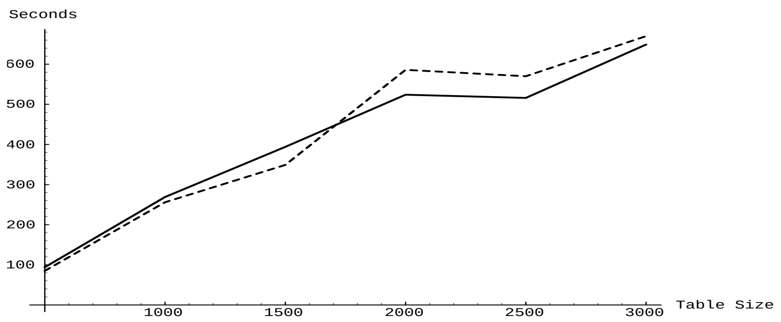

Figure 10: Caching effect in taking the single source

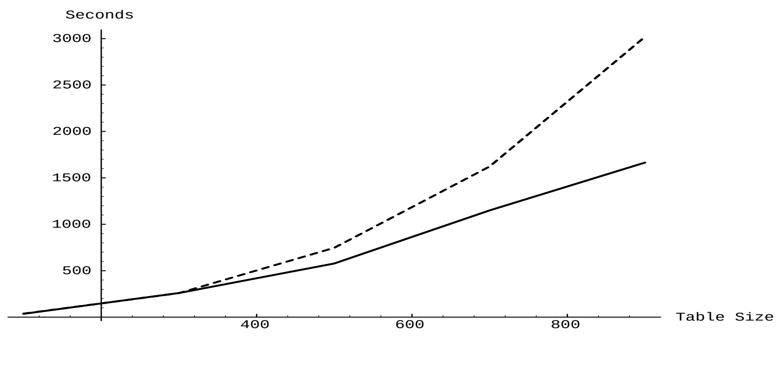

Figure 11: Indexing effect in taking the whole transitive closure

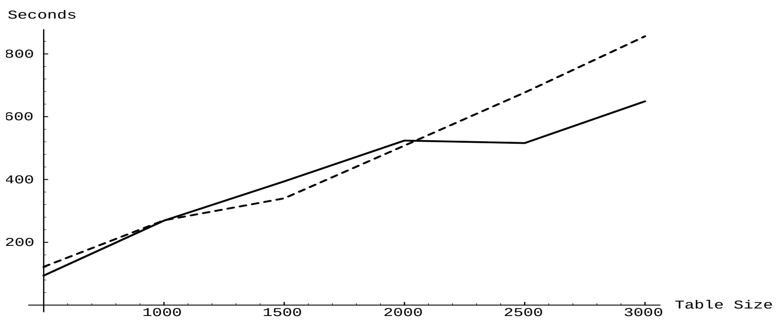

Figure 12: Indexing effect in taking the single source

7 Comparisons with Other Related Work

Some other work has done regarding the performance analysis of taking transitive closures on relational systems [5, 7, 11, 10].

In [5], both analytical and experimental results are presented to suggest some useful heuristics for efficient recursive database processing. Among them, performing selection first, making use of wavefront relations (saving previous processing results to avoid redundant processing), and grouping those joins which reduce the size of intermediate results are demonstrated to be beneficial on a WISS (Wisconsin Storage System) platform. In [7], an adaptation of Warren's algorithm [12] is proposed, its performance is compared with that of an iterative algorithm and an improved version of the iterative algorithm (logarithmic). They evaluate the performance of the algorithms for different source relation sizes, available memory sizes, join selectivities, and maximum path length. The general conclusion is that no algorithm has uniformly superior performance; the adaptation of Warren's algorithm is superior when the source and result relations can be held in main memory.

In [11, 10], a new data structure called "join index" is proposed to facilitate the relational join operation. A join index is a binary relation that captures the semantic links that exist between tuples. The idea is to apply all complex operations (join, union) on join indices and to access the data at the very end. Because the length of an index is shorter than that of a tuple, the size of the data to be iteratively joined will be reduced considerably. In [11], some analytical results are shown for two transitive closure algorithms, "brute-force" and "logarithmic". The relative performances between two versions (using join index vs. not using join index) for each algorithm are contrasted. It is shown that, for various values of parameters, applying either algorithm to a join index rather than the base data yields better performance.

Our work differs from the above in several aspects. The experiments are done using two mature commercial relational systems, thus addressing some real aspects of database systems (such as logging, concurrency control and network overhead). A much more comprehensive set of cost factors involved in evaluating transitive closure operations is identified. Previous work focused primarily on the join cost, we showed that the logging and network overheads can also be significant. One observation we made is that by using an index on the join relations, implementing a more efficient join method such as hybrid-hash join, and allocating a larger buffer cache, we could reduce the join cost significantly, but the logging overhead and the network overhead would stay constant, thus forming an overall bottleneck.

8 Conclusions

Our results indicate that both Sybase and Ingres are significantly worse than the ideal, where we are using Floyd's algorithm as a measure of what is ideal. From figures 4 and 8 it is clear that there is a large performance gap between the relational databases and the ideal in our experimental range.

The transitive closure operation is one of the essential services that distinguish a deductive database system from a relational database system. Our experiment shows the difficulty of augmenting an existing pure relational system to a deductive database system. The main issue is performance.

A relational system typically deals with a small number of large relations. Usually the updates on these relations have to be logged for future recovery if the system crashes. Since the relations are long-lived, caching them will improve the performance. In contrast, when evaluating a transitive closure query, especially when using algorithms simulate sideways information passing [9], many supplementary relations are generated. These relations are typically small and short-lived (usually their lifetime lasts only one iteration). The majority of operations performed on these relations are updates. Logging seems to be overkill since the disk I/O incurred by logging is exorbitant. An option of turning off logging at relation-level would be useful for reducing this overhead. A non-built-in transitive closure operator also suffers a severe network cost and inter-process communication overhead. An efficient implementation would try to place the temporary relations in main memory or adopt a caching policy to favor these relations, for example, by providing "pinning" operations on the frequently accessed relations such as Δ relations and the base relation "family". This way, we could reduce most of the thrashing disk I/O overhead. In our experiment, we observe the number of disk reads is much less than the number of disk writes. Thus the dominant factor in I/O cost is the disk writes (caused by an indiscriminate buffer replacement policy and logging). A caching facility for a deductive database should take this characteristic into account. In order to provide this caching facility we need to modify the existing systems.

One characteristic of the transitive closure operation is that the base relations are used iteratively in the join operation. Δ relations are typically small and only equality joins are performed on them (which means hashing would be a good choice). So hashing or indexing on base relation would be cost effective to reduce the join cost. For instance, join indices [11, 10] would give good overall performance since it only focus on the relevant join attributes and delay the actual tuple fetching until the time to assemble the final result. Since join cost would usually be a major portion of the cost of taking the whole transitive closure, a globally efficient join strategy would pay off in reducing the total elapsed time of the program. This is valid even in a distributed environment.

Some interesting issues arise during the experiment, and they are:

1. How will deductive database optimization strategies such as linear evaluation or magic sets transformation affect the performance problem?

2. How does the problem manifest itself in a distributed environment? In a wide area network configuration, the network set up time and transmission time will likely be more dominant in the total cost.

3. In a database system one might typically want to compute a transitive closure as a transaction. Data private to the transaction does not require privileged system services such as logging/locking performed on them since they are *private* to the transaction itself and temporary in nature. Also a transitive closure operation is typically a long-duration transaction (locks on the base relations are held for too long since we need to take a large number of joins) and only after a number of iterations will it commit. Other transactions accessing the same relation might be blocked for a long time, reducing transaction throughput dramatically. An interesting question is how transaction processing strategies interact with query processing strategies.

Also we would like to do the experiment on POSTGRES, in which a transitive closure operator is directly supported. However, the current version (4.1) of POSTGRES does not correctly compute transitive closures.

References

[1] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *6th ACM Symp. on Principles of Programming Languages*, pages 110–120, 1979.

[2] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM Symposium on Principles of Database Systems*, 1986.

[3] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings o the ACM SIGMOD 1986 International Conference on Management of Data*, pages 16–52, 1986.

[4] C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10:255–300, 1991. Preliminary version appeared in the 6th ACM Symposium on Principles of Database Systems, 1987.

[5] Jiawei Han and Hongjun Lu. Some performance results on recursive query processing in relational database systems. In *IEEE 1986 International Conference on Data Engineering*, pages 533–541, 1986.

[6] Ellis Horowitz. *Fundamentals of Data Structures in C.* Computer Science Press, New York, 1993.

[7] Hongjun Lu, Krishna Mikkilineni, and James Richardson. Design and evaluation of algorithms to compute the transitive closure of a database relation. In *Proceedings of the Third International Conference on Data Engineering*, pages 112–119, 1987.

[8] G. Phipps, M. Derr, and K. A. Ross. Glue-Nail: A deductive database system. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, 1991.

[9] J. D. Ullman. *Principles of Database and Knowledge Base Systems*. Computer Science Press, Rockville, MD, 1989. (Two volumes).

[10] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2):219–246, 1987.

[11] P. Valduriez and H. Boral. Evaluation of recursive queries using join indices. In *Proceedings of the First International Conference on Expert Database Systems*, pages 271–293, 1986.

[12] H.S. Warren. A modification of warshall's algorithm for the transitive closure of binary relations. *CACM*, 18(4):218–220, 1975.