The Performance and Utility of the Cactis Implementation Algorithms

Pamela Drew* and Roger King**
Department of Computer Science, University of Colorado, Boulder, Colorado 80309
Scott Hudson
Department of Computer Science, University of Arizona, Tucson, Arizona 85721

Abstract

The database system Cactis is an experiment in managing computed data in an efficient fashion. Using an incremental update approach and self-adaptive optimizations, the system attempts to minimize the amount of I/O required to update derived data values. Performance tests have been run against a wide variety of databases and transaction streams. The general conclusion is that Cactis performs well, in most cases resulting in a reduction of I/O in the range of 50 to 90 percent. We attempt to isolate various database factors (such as the complexity of the schema and of the derived data) and determine how they affect the performance of the Cactis implementation algorithms, as well as test the major optimization aspects of Cactis in isolation. Finally, we draw conclusions concerning the general usefulness of the Cactis algorithms in database systems, and try to suggest where further research should be performed.

1. Introduction

The Cactis project [HuK86, HuK87, HuK88a, HuK89] began in 1985. The driving goal was to address one specific research issue relating to the support of complex database applications such as CAD/CAM, software engineering, VLSI design, and PCB design: the maintenance of computed data. Design engineers often report that standard hierarchical, network, and relational databases do not provide sufficient support for complex forms of engineering data. For example, in a software environment, there are dependencies relating source modules, object modules, configurations, documentation, bug reports, milestones, etc. PCB and VLSI design require that the intricate constraints involving board wirings be represented.

Doing this with a traditional database system presents two problems. First, the system does not provide any substantive modeling capabilities for explicitly representing computed data, and so the application software must construct computed data structures on its own. This requires the application software to perform a major data transformation, whereby complex data objects are translated essentially into simple records. Second, as a result, it is highly unlikely that the database system will update and retrieve this data in an efficient fashion.

Cactis presents a data model and a set of implementation algorithms for maintaining computed data. The data model is essentially that of an attributed graph [ACR88]. The model is in keeping with the spirit of the numerous object-oriented database research projects, in that Cactis encapsulates a computational capability within data objects. These computations are derived attributes.

The implementation relies on two general techniques. First of all, the system is incremental, in that computed data is updated in a fashion that attempts to recompute as little as possible when an update is made. Second, the algorithms are self-adaptive, in that they learn from previous usage statistics in determining how to schedule computations. Cactis attempts to be efficient both with respect to the number of data objects needed for a computation and the management of the buffer pool. In this way, Cactis achieves its goal of limiting the amount of I/O needed to maintain derived (or computed) data.

Although numerous isolated performance tests have been performed, and preliminary results have been reported [HuK89], an integrated, extensive battery of tests had never been performed on Cactis. Feeling that many database research projects never properly validate their implementation algorithms, we felt that this was necessary to do.

We had two primary goals in running these tests. The first was to determine the limits of the Cactis techniques. In particular, as Cactis is intended to minimize the cost of computing derived data, we wanted to see precisely what kind of derived data makes the system work best. Before performing the tests, we predicted that the system would not work well when the complexity of computed data was small, and that it would operate most effectively when the database consisted of a tight graph of highly-interdependent computed data. Below, we will see that these predictions were not always right.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

*This researcher was supported by a fellowship from U S West Advanced Technologies. **This researcher was supported in part by ONR under contract number N00014-88-K-0559, and in part by a contract from AT&T.

In order to properly test the effectiveness of Cactis, we also had to isolate the two major mechanisms within Cactis: the process for scheduling updates and the data clustering facility. As the second mechanism is the most easy to adapt to other database systems, and as we predicted that it would have the biggest impact of performance, we were particularly concerned with testing the second mechanism in isolation.

Our second goal was to determine what other research must be performed in order to construct truly useful mechanisms for maintaining derived data. We found that there were a few major limitations in our attempt to validate Cactis as a system whose internal algorithms might be useful in other database systems. Further work is needed to construct metrics for categorizing derived data; as it is, we are not able to make very precise statements about what kinds of databases are best served by Cactis.

Another problem is that we have not yet tested the Cactis algorithms on databases which possess not only computed data, but information that is processed with standard, set-oriented queries. Finally, while we constructed a fairly sophisticated system for automatically building test databases, we do not know how real-life databases stand up against our test databases; it is necessary to gather metrics concerning the makeup of databases which naturally embody derived data. The performance tests reported in this paper seem to suggest that Cactis works well within a very broad spectrum; the fact that we did not find any substantial factor that limits the usefulness of the system makes us afraid that our automatically generated databases perhaps do not truly match a natural spectrum of derived data.

In the second section, we provide an overview of the Cactis model and implementation algorithms. Then we report on a set of performance tests. Since there is no standard with which to compare Cactis, we focus on validating the performance of our algorithms over a wide spectrum of databases. We also provide an analysis of the algorithm complexity of the Cactis implementation, and in the conclusion discuss how this compares with the actual performance results. We have concluded that Cactis provides a very effective way of maintaining complex computed data. We have also discovered that for certain forms of computed data, Cactis does not perform particularly well.

2. The Cactis Data Model

A Cactis database consists of a set of typed objects. Each object in the database consists of a number of *attributes*, each of which is assigned a value that may be of any type expressible in the C programming language. The type and number of attributes associated with an object is determined by the type of the object.

External structure can be created by establishing relationships between objects. In conventional object-oriented systems such as Smalltalk [GoR83], the external interface to an object consists of a set of messages that the object can respond to. However, in the Cactis data model the external interface to an object consists of a set of (typed) data values that flow into and out of the object across (typed) relationships.

Attributes may be assigned values by user operations or they may be computed, and all such values are visible only within individual objects unless they are passed through relationships to other objects. Non-computed attributes support a form of structural encapsulation. Values transmitted into an object may be used to derive computed attribute values. This supports a form of behavioral encapsulation, whereby an object may respond to changes elsewhere in the database by redefining its own derived attribute values. Values which flow out of an object may be either derived or non-derived. In the Cactis model the internal implementation of derived attributes is expressed declaratively in the form of *attribute evaluation rules*.

The Cactis data model provides direct support for rich semantic relationships. Consequently, the Cactis data model can be seen as a semantic data model [HuK87]. As in other semantic models, relationships in the Cactis model are typed and directed. The type and direction of a relationship is used to represent a semantic concept. However, relationships in a Cactis database are significantly different than those found in other semantic models such as the Entity-Relationship Model [Che76], the Semantic Data Model [HaM81], or the Functional Data Model [KeP76, Shi81]. In these models, relationships are defined with types. An object type uniquely defines the relationships that objects of that type may participate in — including the range types of those relationships.

The Cactis model, on the other hand, separates object types from relationship types. A relationship type is determined solely by the type, direction, and number of values that flow across the relationship. Consequently, any object which exports and imports the proper set of values may participate in a relationship of that type. Expressed in different terms this means that the range type of a relationship does not depend on the domain type. In this way, relationships may be moved around to connect up objects of various types. All that matters is that a relationship deliver attribute values of the correct type and number, as expected by the object at the "range" end of the relationship.

This notion of relationships has important implications. In particular, it allows much greater extensibility of the database since an object may — in fact must — remain completely ignorant of the type of any object it is related to. An object only knows what types of values flow across a relationship, not how the values are produced or consumed. This allows objects to be transparently replaced by more complex equivalents in a manner that is completely transparent to any objects they are related to.

3. Incremental Update

The Cactis data model provides a powerful mechanism for supporting derived data. However, it also presents a significant challenge for efficient implementation; this is due to the fact that such a general-purpose facility for specifying derived data could potentially cause vast areas

of the database to be read or written every time a change is made. This section describes the basic update algorithm used in the Cactis system, while the next section considers the self-adaptive optimizations developed to improve performance in a disk-based environment. Please note that Cactis is tuned only toward minimizing the I/O cost of calculating derived values; it does not support any conventional access structures (like btrees) for doing set-oriented retrievals. We also do not consider concurrency control or recovery in this paper; all our tests were run on a single user version of Cactis.

The update algorithm used by the Cactis system is incremental. When a change is made to an attribute in the database, the update algorithm at worst has to do work proportional to the set of derived attributes that are (transitively) dependent on the attribute changed. The update algorithm does not do work proportional to the whole database or the set of all objects of certain types. More importantly, if a change to attribute A causes attributes B and C to be transitively affected, and if B and C both transitively cause attribute D to be updated, then D is only updated once. With a mechanism like data driven triggers [BuC79] this is not guaranteed. This can cause exponential behavior in the worst case (see for example [Rep84] for an analysis of data driven updates of this form). The Cactis algorithm in comparison is strictly linear in its behavior.

While an update is being propagated to all dependent attributes, access to attributes which are not transitively related to a changed value can be accessed directly without additional overhead. In addition, the update algorithm is lazy. The system ensures that when values are examined, they are correct with respect to their defining equations. However, attributes are not recomputed unless and until they are actually examined. In this way, work is avoided for attributes which would have received a series of new values, but for which those new values are never actually needed by the user. This allows values of only infrequent interest to be derived at little cost.

The incremental update algorithm is best understood as a series of graph traversals on the *attribute dependency graph*. Nodes in this graph represent attributes, while edges in this graph represent (direct) dependencies (caused by derivation rules) between attributes. After each database update, the algorithm works on this graph in two phases. The first phase identifies potential work to be done — transitively dependent attributes that may need to be recomputed as a result of the update. The second phase performs the actual recomputation of attributes. Whenever an access requests one or more values, the system determines which attributes need to be recomputed and invokes the second phase of the update algorithm to recompute those values as needed. Normally, attributes are only recomputed on demand, however, the schema designer has the option of declaring that certain attributes are *important*. Important attributes are brought up to date even if their values are not used.

The first phase of the algorithm is the *mark out-of-date* phase. It starts at the point(s) of change and marks all attributes which might be affected by the change as out-of-date. This marking process is simply a traversal of the attribute dependency graph. Whenever an attribute value is requested, the system first checks to see if it has been marked out-of-date. If it has, its value is recomputed by invoking its attribute evaluation function. This is the second phase of the algorithm. The function recursively requests the values of other attributes, which may invoke the second phase of the algorithm for these attributes. In this way, the algorithm obtains a final value which is correct with respect to all transitive dependencies. Once a value has been reevaluated, its out-of-date mark is reset so that further accesses will not cause extra recomputations. Both phases of the algorithm are thus graph traversals — one following the dependency edges forward and the other backward. This point will be important to the optimizations described in the next section.

A proof of correctness and complete analysis of the attribute evaluation algorithm used by Cactis can be found in [Hud89]. To summarize this analysis: for an update, the algorithm performs total work which is, in the worst case, proportional to the set of attributes transitively dependent on the attribute(s) which were changed. Note that when an update is made, Cactis will potentially examine all dependent attributes twice (once for each phase of the update algorithm), while a trigger-like mechanism could evaluate a given attribute an exponential number of times. In order to achieve linear behavior within Cactis, attribute evaluation functions must be purely applicative (i.e., they must have no observable side-effects and must compute their result strictly on the basis of their parameter values). This limitation is not present with conventional trigger mechanisms.

Unfortunately, the Cactis algorithm is not optimal since it could conceivably do work directly proportional to the smaller set of attributes which either actually change value after an update or are directly related to an attribute which changes [Rep82]. In other words, the algorithm updates the optimal set of attributes after a change, but may perform a non-optimal amount of extra overhead work in order to do this (but never more than linear in the size of the transitive dependency set of attributes).

While an asymptotic analysis can sometimes be deceptive because large constants are hidden, performance tests on an in-memory version of the algorithm indicate that this is not a problem in this case. The basic algorithm is very simple and the constants of proportionality are very small in practice. Consequently, this analysis is in fact of practical as well as theoretical interest.

4. Self-Adaptive Optimizations

Computation of derived data on disk is typically expensive since following transitive dependency chains can potentially involve reading many different blocks from disk to access relatively few attribute values. This means that while the Cactis algorithm is asymptotically very good and in practice performs very well in an in-

memory setting, it many not actually perform well in a disk-based setting. This section describes a set of self-adaptive optimization techniques which are designed to ensure that it does. These techniques are the focus of the performance tests described in the next three sections.

Two complementary techniques are used to reduce the amount of disk I/O performed by the system. (Note that Cactis is not tuned toward minimizing CPU time; as a result our performance tests are almost exclusively concerned with I/O time.) The first of these techniques involves taking advantage of alternative orderings in the computation of derived data. In particular, scheduling parts of the graph traversals used for updates in a way designed to reduce total disk I/O. The second technique involves periodically reclustering the database to improve the locality of reference that occurs during retrieval and update of derived data. Both these techniques use statistics collected at the object level to guide optimizations. These statistics, in the form of a decaying average, track changes in access patterns and allow the algorithms to adapt to the fine-grained usage patterns that actually occur over time.

The Cactis incremental update algorithm consists of two graph traversals, one to mark attributes out-of-date, and one to compute derived attributes. The first *scheduling* optimization used by the system takes advantage of the flexibility in ordering that is allowed in these traversals. The mark out-of-date traversal can clearly proceed in any order which marks all reachable attributes. In particular, at each step in the traversal, the system is free to choose the next node to visit based on what it heuristically predicts will reduce total disk I/O.

The second traversal of the algorithm also has flexibility in the order in which it proceeds. Because attribute evaluation functions in the Cactis data model must be purely applicative, an attribute evaluation function may request its parameters in any order and will still compute the same result. This means that the second traversal may also proceed in any order so long as a given attribute is computed after all of the attributes it directly depends on are computed.

In order to optimize the order in which each of the traversals proceeds, the system maintains statistics about traversals. For each edge of the dependency graph that crosses between objects, the system keeps a decaying average of the total number of objects visited along that path. In particular, each time a traversal is completed across an edge the new reference count is averaged with the old count in an exponentially decaying fashion. For a given dependency edge, this statistic gives an estimate of how many objects must be examined in order to update the value flowing across that edge.

Given this statistic, the system uses the following heuristic to schedule updates. First, if the next step of a computation can be performed using data already buffered in memory (i.e., the next step in the traversal will visit an object buffered in memory) that computation is given priority for scheduling. Second, for computations that require I/O, priority is given to the one which is predicted to require the least I/O. This is evaluated by examining the statistics. These heuristics represent a shortest job first approach. The idea is to attempt to finish small computations quickly so that buffer space can be freed up to support computations which require more space.

The locality of reference of a database can be decreased if objects which are often used together are placed in the same physical disk block. The second optimization performed by the system is to periodically recluster the database (off-line) on the basis of usage statistics in order to increase this locality of reference. For this optimization, statistics are kept which count how many times each relationship is traversed. Relationships which are frequently traversed generally connect objects that are referenced together during the traversal process. These objects are hence candidates to be clustered in the same disk block.

Off-line reclustering of the database is done using a greedy heuristic. Clustering starts by placing the most frequently referenced object in the database in an empty block. The system then considers all relationships that go from an object inside the block to an object outside the block (which has yet to be placed in a block of its own). The object at the end of the most frequently traversed relationship is placed in the block. This process is repeated until the block is full. The system then repeats this process for new blocks until all objects have been assigned blocks.

Note that the two optimization techniques work hand-in-hand. The first tries to minimize page thrashing by choosing an update that will result in little I/O. The second attempts to maximize the utility of the pages which are fetched.

5. A Test Framework

This section describes the test framework used to explore the effectiveness of the Cactis implementation techniques. The goal of this test framework was not to determine the absolute performance of the system (e.g., transactions per second) since the system is far from optimized and there are currently few object-oriented systems which it can be directly compared with in a meaningful way. Also, the system was not designed to serve as a general-purpose DBMS; rather our intention was to build a DBMS tailored specifically toward efficient maintenance of derived data. In particular, the system does not support any set-oriented queries. Therefore, the goal of the tests was to determine the effectiveness of the implementation techniques used, to explore how this effectiveness changes under different situations, and to determine the limits of these techniques.

Also, the tests performed did not make use of a sample application or "real life" benchmark transaction stream, due to the fact that as yet, no clear characteristics of a *typical* application using extensive derived data have emerged. Instead test data sets and query streams were generated in a random but carefully parameterized fashion. This allowed specific characteristics to be varied in isolation to study their effects on system performance.

While the test results in the next section provide specific percent improvement numbers for each of the optimization techniques used by the system, these numbers are difficult to apply to specific applications. It would be even more unreasonable to attempt to apply them to database systems which use techniques similar to, but different from, Cactis. A more important result is the overall effectiveness of the specific Cactis optimization algorithms, and how this varies across important dimensions of database structure. In sum, the tests were designed to determine performance trends; the specific numbers have to be considered only within a narrow context.

In order to explore a wide range of situations, a program which generates random databases and query streams was constructed. This generator accepts a series of parameters (described below) and creates a Cactis schema, a database of test objects, and a one or more test query streams to run against the database. The parameters supplied to the database generator control the characteristics of the generated schemas, objects, and query streams. These characteristics are designed primarily to vary in the following three important categories:

- structure and complexity of connections between objects

In this category, the structure of the generated databases is varied at three levels. At the highest level, we are concerned with the total *connectedness* of the data. That is, given an object, how many other objects is it related to, either directly or through transitive relationships. This concerns relationship connections, not specifically derived attribute dependencies. At the object level, we are interested in the locality of reference of related objects; if a series of relationship connections is followed, what is the probability of returning to an object already on the relationship path. Finally, at the level of individual attributes, we are interested in the structure of derived attribute dependencies. Specifically, we vary the length of a chain starting from a non-derived object to the last derived object which, due to an attribute derivation, transitively depends on the non-derived object. For example, for a derived attribute, does it tend to have long chains of attribute dependencies, or only short chains. Similarly, do the attribute dependencies tend to fan out widely or do they tend to confine themselves to a small number of relatively linear chains.

- query characteristics

In this category, we are interested primarily in the locality of reference of the queries — do they tend to access variables in a local neighborhood of the database or spread their accesses across unrelated sections.

- buffering characteristics

Finally, in this category, we are interested in the size of in-memory buffer space. In particular, we are interested in how much of the database can be buffered in memory at one time, since this characteristic can have a dramatic effect on overall performance.

In order to generate test data sets with specific properties out of the range of possibilities given above, the database generator accepts a series of parameter values. These parameters control generation of a random database which at an informal level proceeds as follows: First, all

```
Procedure Connect(objs : set of object; cycle_bias : float);
  Var
    obj1, obj2: object;
    connected     : set of object;
    unconnected   : set of object;
  Begin
    -- initially no connections made
    connected := empty; unconnected := objs;
    -- start with one object in connected set
    obj1 := remove_random(unconnected);
    Repeat
      -- choose one end of a new relationship for within
      -- connected set
      obj1 := choose_random(connected);
          -- decide if this will introduce a cycle
          If random() > cycle_bias Then
             obj2 := remove_random(unconnected);
                connected := connected + obj2;
          Else
                obj2 := choose_random(connected);
          End If;
          -- make the relationship
          Relate(obj1,obj2);
    Until unconnected = empty; -- all objects connected
End;
```

Figure 1. Relationship Creation Within Partitions

the objects in the test database are created. The number of objects is controlled by the **total_size** parameter. These objects are then partitioned into groups that will eventually form connected components within the database (by this we mean connected via relationships, not necessarily attribute dependencies). The size of each partition is controlled by the **connected_size** parameter. This parameter controls the overall connectedness of the database — in other words how closely coupled or isolated objects are. Larger values of **connected_size** imply more relationships between objects while smaller values imply more isolation.

Once partitions have been formed, each partition is processed independently to produce a connected component of the database. This step involves creating relationships between objects. At this level, the property varied is locality of reference — the likelihood that, when following relationships, one is likely to return to the same neighborhood or instead to visit new objects. While it is difficult to exactly control this probability in a randomly generated database, it can be approximated. We do this by manipulating the properties of the undirected graph formed by objects and relationships. At one extreme, if this graph is a tree, then the locality of reference is minimized since no path ever returns to the same object twice. On the other hand, as cycles are introduced into the graph the probability of returning to the same object increases. The parameter **cycle_bias** is used to control the introduction of cycles into the relationship structure and hence the locality of reference at the object level.

The algorithm used to generate relationships for each partition is shown in Figure 1. It works by selecting one object to start the connected component then repeatedly

creating a relationship until all objects in the partition are connected. At each stage the probability of introducing a relationship which closes an (undirected) cycle is **cycle_bias** while the probability of creating a non cycle edge is 1 - **cycle_bias**.

Once a relationship structure has been established for each partition, an attribute dependency structure for derived attributes must be established. The dependency structure can be seen as a directed graph embedded within the undirected graph formed by objects and relationships. Each node in this directed graph corresponds to an attribute, while each edge corresponds to a dependency between attributes.

In order to apply fine control to the structure of the dependency graph(s) constructed, a series of templates are used. These templates give a set of dependency graphs in isolation from the outer object-relationship graph. The database generator acts by randomly choosing a template graph, then instantiating attributes and dependencies matching a copy of that graph within the outer object-relationships graph. Layout starts at a randomly selected object and proceeds by placing dependencies across randomly selected relationships. Attributes are declared, and simple attribute evaluation functions are generated to induce the proper attribute dependencies (all attributes are simple integer values and all evaluation functions are simple additions). By varying the set of template graphs used, the average path length and average fan-out can be controlled directly.

Once the database generator has constructed a test database, it proceeds to construct one or more random query streams. These query streams are a mixtures of reads and writes at a ratio determined by the **read_write_ratio** parameter. To reduce the total number of tests to be performed, all experiments reported here use a ratio of 2:1.

In addition to the ratio of reads to writes, the nature of the query stream is also controlled by the **transaction_type** parameter. This parameter is set to either **random** or **localized**. At the **random** setting, the attributes accessed at each stage of the query are chosen at random from anywhere in the database. This represents the least locality of reference within the query and the most difficult case for performance optimizations. On the other hand, if **transaction_type** is set to **localized**, transactions are generated which start at a randomly selected attribute, but which always proceed to access attributes of related objects. This represents a high degree of locality of reference and hence the best case for optimizations. In addition, when a transaction type of **localized** is selected, the query stream consists of three repeated sequences — again, the best case for the optimizations — whereas **random** query streams contain no repeated sequences — again, the worst case.

Once a test database and test query stream have been selected, actual test runs were performed using the procedure shown in Figure 2. For each test run statistics were gathered which measured the total number of disk accesses. The disk accesses with a simple first come first serve (FCFS) scheduler (i.e., no optimization) were compared against those for the optimized (priority-based) scheduler to obtain a percent improvement due to the priority scheduler. In addition, statistics were kept for the first 1/3 of the query stream (run without clustering) and compared with the final 1/3 of the query stream (run after clustering twice) to obtain an improvement due to the clustering algorithm. Clustering improvement numbers are not inflated by a poor initial clustering since the initial clustering of the database insures that all objects on the same disk block are directly related to some other object in that block (except in rare boundary cases). This is already a reasonably good clustering and is significantly better than the worst case.

6. Test Results

Using a range of parameter settings on the database generator, we created 264 different databases. Each database used a different randomly generated query stream with a 2:1 ratio of reads to writes. To keep the time required to perform the tests manageable, databases of 100 objects were used for all tests. (Creating databases with larger numbers of objects would have caused the database generator to run for very long periods of time.) However, to compensate for these relatively small databases, a correspondingly small in-memory buffer area was used. Because exact object size could not be controlled (since the process of randomly laying out attributes causes objects to be of varying size) a fixed number of objects (4) were placed in each disk block. In cases where the buffer size was held constant (i.e., results shown in Figures 4 through 9), 3 blocks were allowed for in-memory buffer space resulting in 12 object being buffered at one time.

Each graph presented in this section shows the average behavior of the system over a particular range of parameter settings. In addition each data point is depicted with a range indicating the highest and lowest results obtained at that parameter setting. As will be seen below, variances at given settings were fairly low.

```
Procedure run_test(DB: database;
                   query: query_stream; buf_size: int)
Var
  sched : scheduler_type;
Begin
    For sched ∈ {FCFS, Priority} Do Begin
    -- clear all clustering and scheduling optimization statistics
            reset_opt(DB);
            -- run first third of query stream
            Cactis(DB, query.part1, sched, buf_size);
            -- recluster the database
            cluster(DB);
            -- repeat twice more
            Cactis(DB, query.part2, sched, buf_size);
            cluster(DB);
            Cactis(DB, query.part3, sched, buf_size);
            cluster(DB);
    End;
End;
```

Figure 2. Testing Procedure

The final major parameter associated with creating a random database is the template used to determine the fine-grained attribute dependencies. All templates used for these tests were in the form of a tree like the one shown in Figure 3. This template is in the form of a modified full binary tree with an extra node inserted across each edge. The template trees themselves were parameterized in two ways. First the depth of the tree — the distance from the root to each leaf — was varied. The depth of the template shown in Figure 3 is 8. Second, the branching factor — the number of children of each node that has more than one child — of the tree was changed. The branching factor of the template shown in Figure 3 is 2. In cases where the template was held constant (i.e., the results shown in Figures 4 though 12) a depth of 10 and branching factor of 2 was used.

The graphs can be analyzed in three groups. Figures 4 through 9 measure the effects of changes in course grained structure (i.e., the **connected_size** parameter) on the algorithms while holding the in-memory buffer size and fine-grained structure (i.e., the attribute dependency template) fixed. This corresponds to varying the major structural aspects of the schema (i.e., the relationship assignments) while holding other factors steady. The second set of tests shown in Figures 10, 11, and 12 measure the effects of varying in-memory buffer size while keeping other factors fixed. Finally, the third set of tests (shown in Figures 13 through 18) indicate how the system performs under variations in the structure of fine-grained attribute dependencies. This corresponds to varying the computational dependencies between derived attributes, but holding the major structural and buffering aspects of the database steady.

All tests presented in this section depict the results for a **cycle_bias** of 30% which is a mid range value. Tests were also performed for biases of 10% and 50%. These test (not shown) produce very similar results — both the average value and variances are close to those for 30%. Furthermore, the **cycle_bias** parameter does not seem to directly determine performance since the system

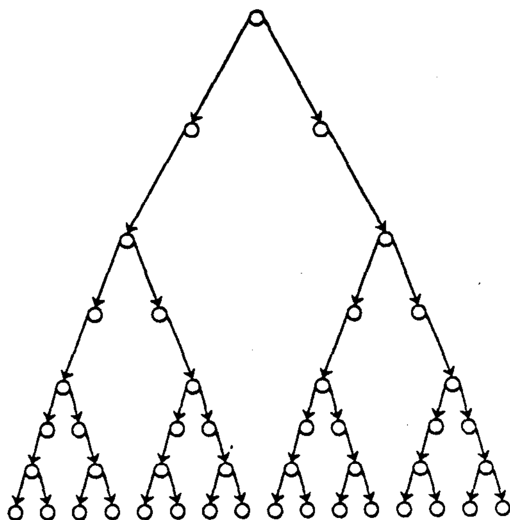

Figure 3. Sample Attribute Dependency Template

performed best for low cycles in some cases and best for high or medium cycles in others. In all cases each choice of a bias resulted in comparable results (although the variance between different biases is typically somewhat greater than that for the runs under a single bias). Since **cycle_bias** failed to be a predicting factor, test results for high and low cycles are omitted and only medium (30%) cycles are shown.

Figures 4 and 5 indicate the improvement from the priority scheduling optimization in isolation from clustering. These figures measure percent improvement of prioritized scheduling over non-optimized first come first served (FCFS) scheduling as connected component size (**connected_size**) is varied. Figure 4 shows the results for **random** transactions while Figure 5 shows the results for **localized** transactions. Both transaction types performed comparably with **localized** transactions resulting in slightly greater improvements. The general trend in both cases is for the optimization to perform best in a mid range of connectivities and poorest when connectivity is either very high or very low.

Figures 6 and 7 indicate improvement from clustering in isolation from priority scheduling (i.e., clustering with non-optimized FCFS scheduling). Figure 6 shows **random** transactions while Figure 7 shows **localized**. Again, there is not a large difference between localized and random, but as we would expect, localized tends to perform slightly better overall. Here we also see that the clustering optimization provides considerably better overall improvements than the priority scheduling optimization, and that the clustering optimization is more robust — it does not loose effectiveness at higher levels of connectivity, but instead remains relatively steady near its best performance.

Although the clustering and scheduling optimizations are largely complementary, their effects are not strictly additive. Figures 8 and 9 indicate the percent improvement of clustering done in the presence of priority scheduling.

While Figures 4 through 9 provide results for a fixed in-memory buffer size of 3 blocks (12 objects), Figures 10, 11, and 12 provide an indication of how performance changes as the amount of buffer space changes. For these tests, **connected_size** was fixed at 20, all transactions were of the **localized** variety, and the same attribute dependency template as Figures 4 through 9 was used (i.e., depth of 10 and branching factor of 2). Figure 10 shows improvement due to scheduling over a range of buffer sizes; Figure 11 shows the improvement due to clustering without scheduling; and Figure 12 shows the improvement due to both clustering when scheduling was performed. Again, as a general trend, clustering provides considerably more optimization than scheduling. In both cases, mid range buffer sizes performed best. Very small buffer sizes seem to be intrinsically prone to at least some thrashing, while large sizes allowed most or all of a connected component to remain memory resident so that the naive, unoptimized approaches still worked well.

The final group of tests depicted in Figures 13 through 18 show how varying the fine-grained structure of

attribute dependencies affects each of the optimizations. The same basic attribute dependency structure (i.e., a tree with the same shape as the one shown in Figure 3) was used for these tests. However, in this case, the depth and branching factor of the tree was varied. Other parameters remained fixed with a buffer size of 3 blocks, a **connected_size** of 20, and all **localized** transactions.

Figures 13, 14, and 15 show the effects of varying the depth of the tree (i.e. length of dependency chain from the root to each leaf) while keeping the branching factor fixed at 2. Figures 16, 17, and 18 show the effects of using a shallow dependency tree template (depth 2) and varying the branching factor. Each of these cases seems to exhibit a relatively high variance across the range of structures. In general, these results do not seem indicate a strong correlation between optimization performance and either the depth or branching factor of the attribute dependency template.

As a final test, Figure 19 shows the results of profiling the database code itself to determine the ratio of work done within the scheduling algorithm versus that done for I/O operations. For simplicity this graph measures the ratio of total function calls made as the course-grained complexity (connectivity) of the database is varied. As can be seen, the proportion of work done for scheduling only increases very slightly as the database complexity increases.

7. Implications and Analysis

At the highest level, the test results presented in the last section show that the self-adaptive optimizations used in the Cactis system do indeed act to reduce I/O costs. It can also be seen that the clustering optimization represents the "big win". It provides the biggest improvement in performance and, as it is done off-line and involves a greedy algorithm, is the simplest optimization. This optimization also has the advantage of being the least dependent on the particulars of the Cactis data model — a variation of this optimization could be applied to a variety of different object-oriented data models. Finally, clustering also appears very robust, working reasonably well (In [FJL88] the authors discuss a database system which is making use of the Cactis algorithm for clustering data.)

There are two potential drawbacks to the clustering optimization. First, it is likely to be incompatible with the kinds of clustering used for set-oriented queries. This is due to the fact that Cactis clusters data in a fashion that disregards the type structure. For set-oriented queries, it is important to be able to isolate all the objects of a given type that meet certain properties. At a minimum, the Cactis clustering algorithm assumes that most data manipulations involve the retrieval of computed attributes, not set-oriented retrievals. Second, this optimization is currently performed off-line and hence requires an interruption in the availability of the database. However, we are exploring techniques to allow this optimization to be performed in an on-line incremental fashion.

The very impressive results gathered in testing the Cactis clustering algorithm cause us to feel that database clustering is a research direction worth pursuing. On-line, interruptable algorithms for clustering data under a wide variety of situations are needed. The effects of mixing derived data with set-processed data is just one of the parameters that should be examined. Cactis assumes that all databases objects are large and of the same size. Databases with a wide variety of object sizes must be tested, and special algorithms to handle this will be needed. Also, Cactis uses a simple, greedy algorithm which is highly sub-optimal. Under certain circumstances, it will be worth the overhead of a more complex algorithm to get even better performance.

While somewhat overshadowed by clustering, the scheduling optimization also provides clear improvements. However, the scheduling optimization seems less robust — working well for most connectivity patterns but not for the entire range of databases. It is also much less compatible with other current object-oriented database implementations.

A major trend of the results is that none of the optimizations work well for databases which have only very small connectivity. This indicates that the techniques perform the job they were intended to perform — optimization for complex derived data — but do not help much in cases where data is isolated. This is a clear indication of where the Cactis optimization techniques do **not** work. This shows for example, that the optimizations probably would not be helpful for conventional applications where most access is performed by means of associative search. An interesting experiment would be to see how Cactis would perform with a mix of query types. It might be that derived attribute computations are difficult to optimize in the presence of other queries. Similarly, it would be interesting to measure Cactis' performance when a major shift in the locus of a query stream occurs. It may also be that the optimization techniques would be slow to respond to radical changes in usage patterns despite the exponential decay of the usage statistics.

The test results presented in the previous section also provided a few surprises. It was initially thought that the depth and branching factor of the attribute dependency template trees and the number of cycles in the relationship graph would have a strong correlation with overall performance. However, instead the only strongly correlated factor turned out to be the global connectedness of the data. We believe the reason for this is that for the parameters controlling the finer-grained structure of the database induce much more complicated behavior patterns that cannot be correlated with a simple linear scale of variations. More extensive tests, however, might allow more precise conclusions on the effects of various parameter settings.

Finally, one of the most significant conclusions is that the business of generating databases is a tough one. Although initially, we thought that the parameter arrangements supported by our generator would allow fine-grained control over the creation of databases and transactions streams, it turned out that we were not always able to do exactly what we needed. For example, it was difficult to control the average size and variation

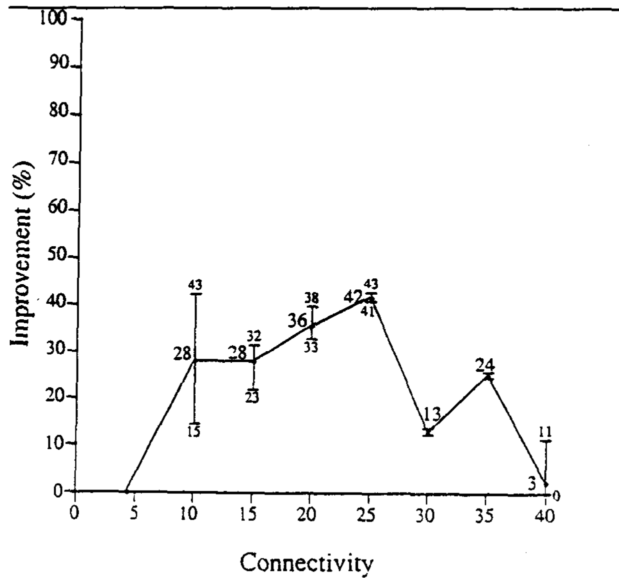

Figure 4. Improvement Due to Priority Scheduling (Random Transactions)

in size of objects. It was also hard to precisely control the layout of attributes in objects. The template system, while simple to use, only gave us very coarse control. This may have something to do with the apparent conclusion that attribute layout did not affect performance. And, our generator also made it difficult to experiment with databases containing many objects.

In sum, we don't feel that our generator is by any means a way of exhaustively searching the space of all possible derived data databases. In general, our generator does not satisfactorily simulate real-life databases designed to serve users with complex tasks. A more sophisticated tool is needed for experimenting with new algorithms for maintaining complex data in such applications as engineering design. To address this, we are currently working on a system called A La Carte [DKB89], which provides a test bed for selecting various database facilities (such as a powerful data model, a novel concurrency control technique, and a transaction mechanism oriented toward long, interactive design transactions) and then plugging in new database implementation algorithms. Only with a much more sophisticated test bed, can algorithms for supporting complex databases be properly designed and tested, without having to rebuild a large chunk of a new database each time.

8. The Future of Cactis

We are currently designing and constructing a distributed version of Cactis, called Cacti [HuK88b]. The system is targeted for a local network of Sun workstations. Our central motivation in pursuing this effort is that we envision derived data to be important in many engineering design efforts, and design engineers often work in distributed, interactive environments. The implementation of the system is being greatly facilitated by the fact that the graph algorithm in Cactis is naturally parallel, thus making it easy to adapt it to a distributed environment. In keeping with the self-adaptive nature of Cactis, the new system uses usage statistics to replicate, migrate, and recluster data around the network. We also plan to develop a better data manipulation language for Cacti, one that allows set-oriented queries and derived attribute computations to be integrated.

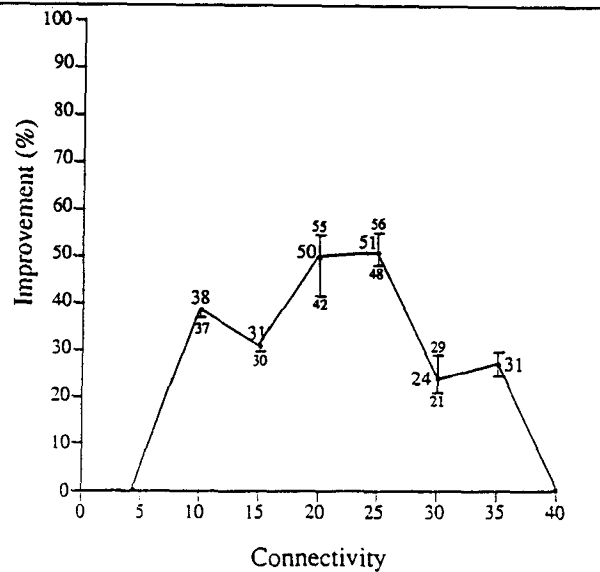

Figure 5. Improvement Due to Priority Scheduling (Localized Transactions)

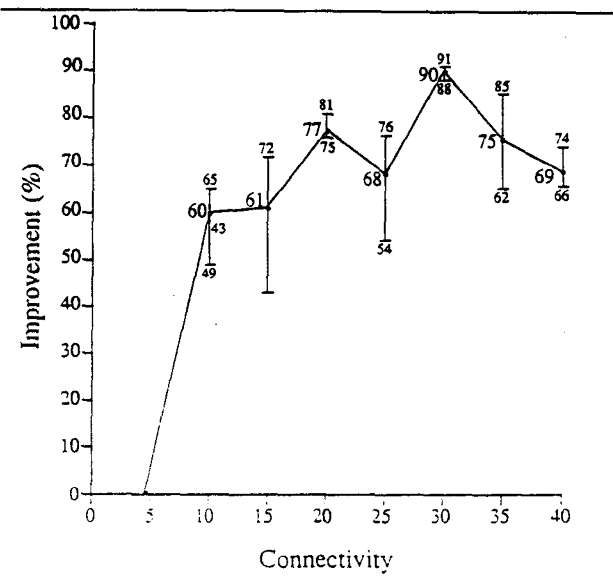

Figure 6. Improvement Due to Clustering Alone (Random Transactions)

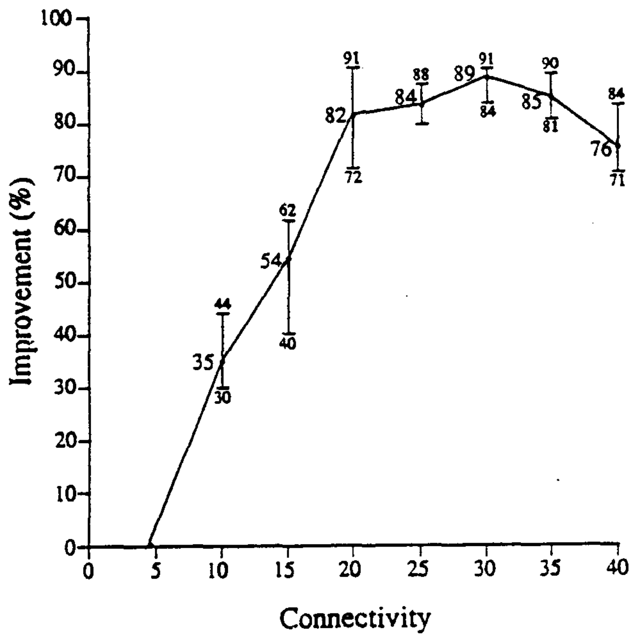

Figure 7. Improvement Due to Clustering Alone (Localized Transactions)

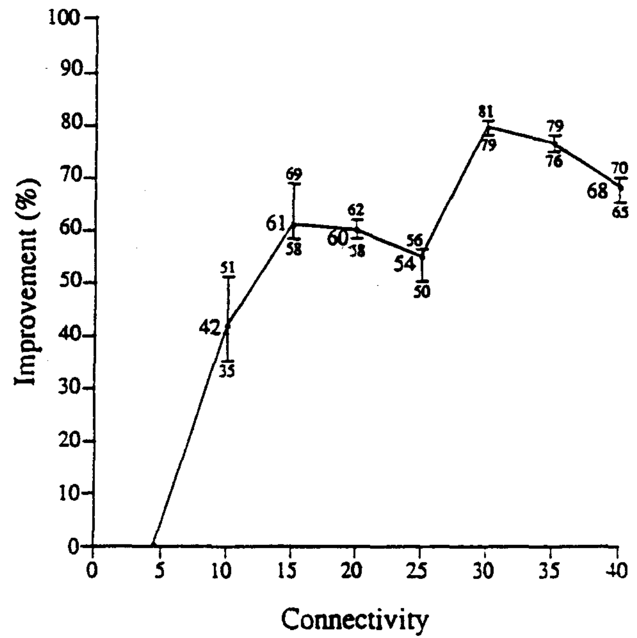

Figure 8. Improvement Due to Clustering With Scheduling (Random Transactions)

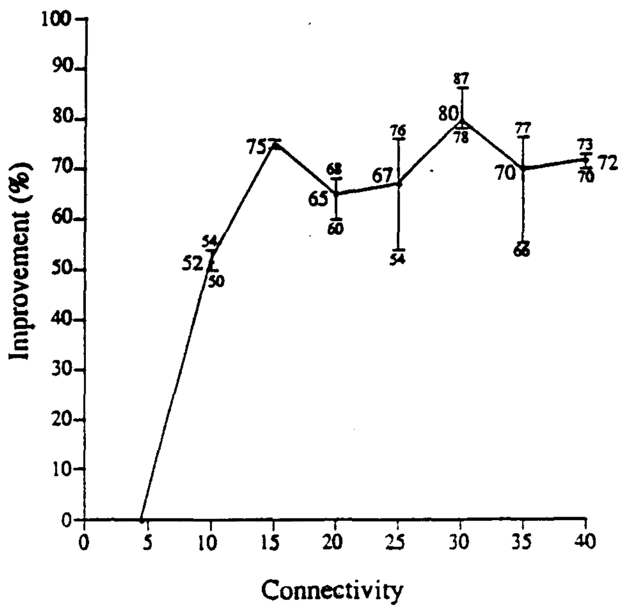

Figure 9. Improvement Due to Clustering With Scheduling (Localized Transactions)

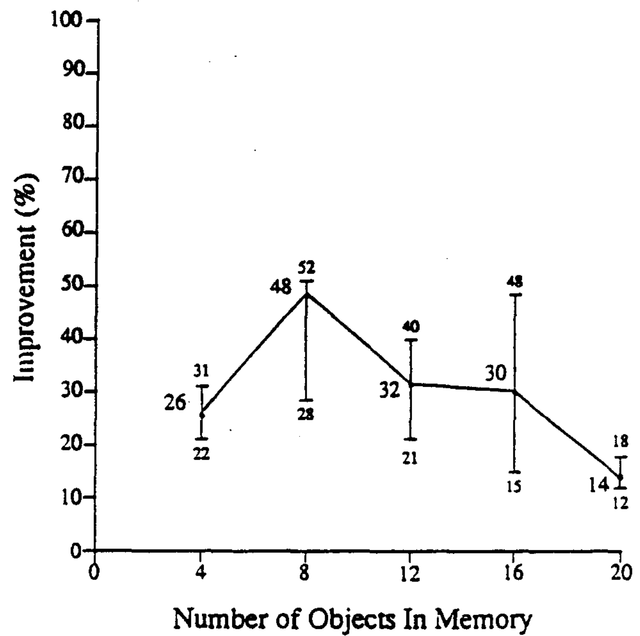

Figure 10. Improvement Due to Scheduling Under Varied Buffer Sizes

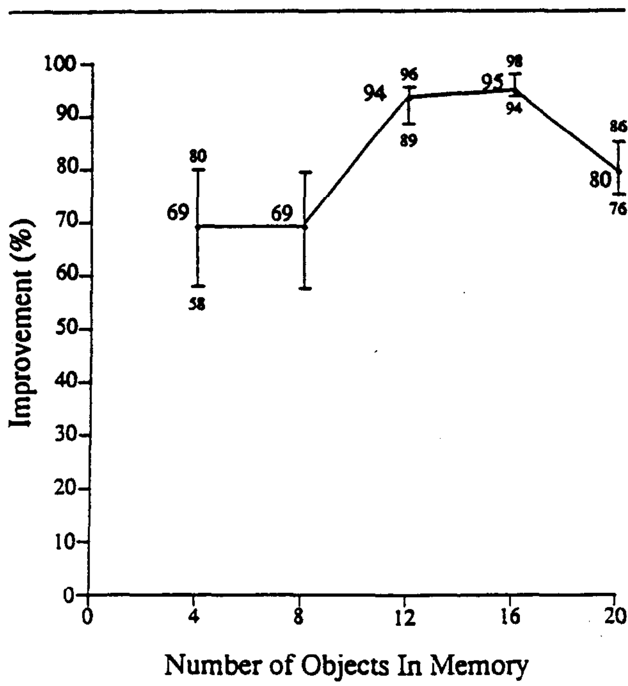

Figure 11. Improvement Due to Clustering Alone Under Varied Buffer Sizes

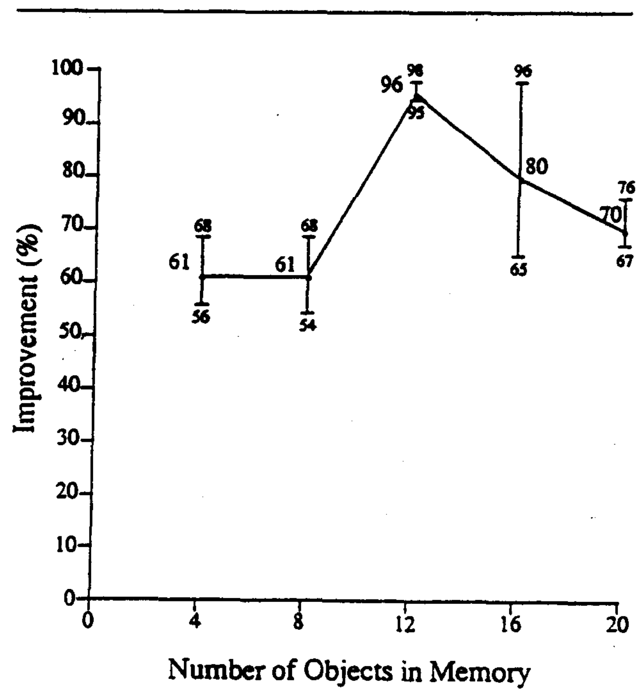

Figure 12. Improvement Due to Clustering With Scheduling Under Varied Buffer Sizes

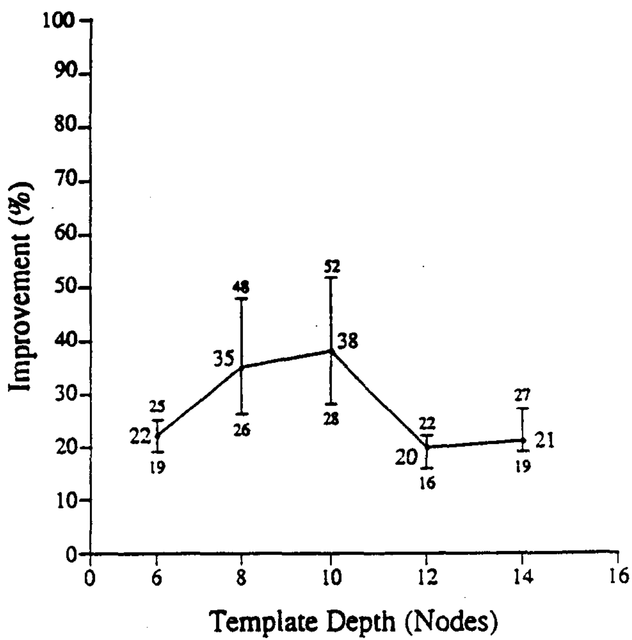

Figure 13. Improvement Due to Scheduling Under Varied Template Depth

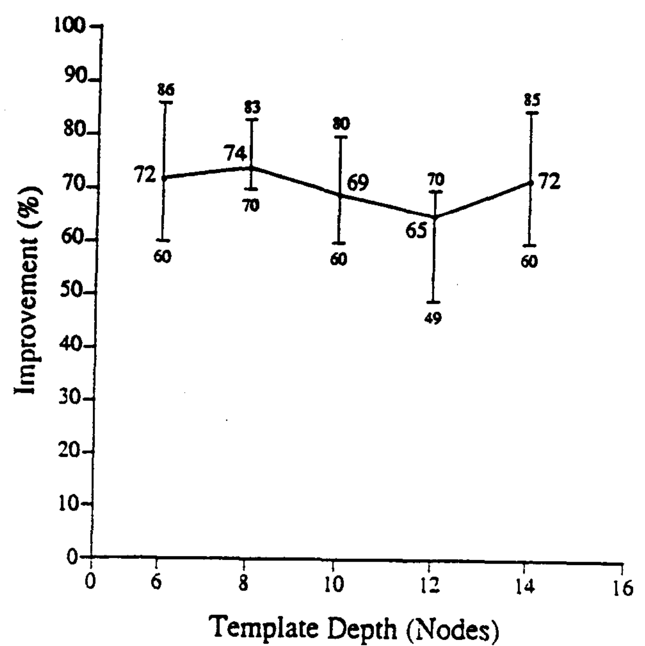

Figure 14. Improvement Due to Clustering Alone Under Varied Template Depth

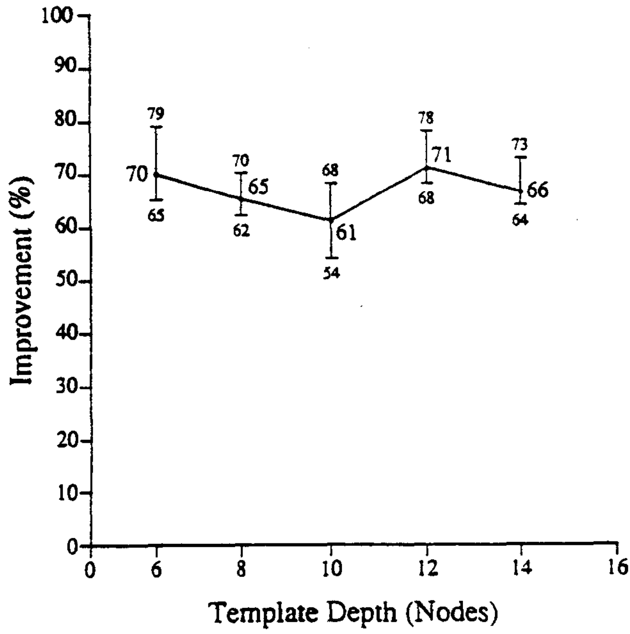

Figure 15. Improvement Due to Clustering With Scheduling Under Varied Template Depth

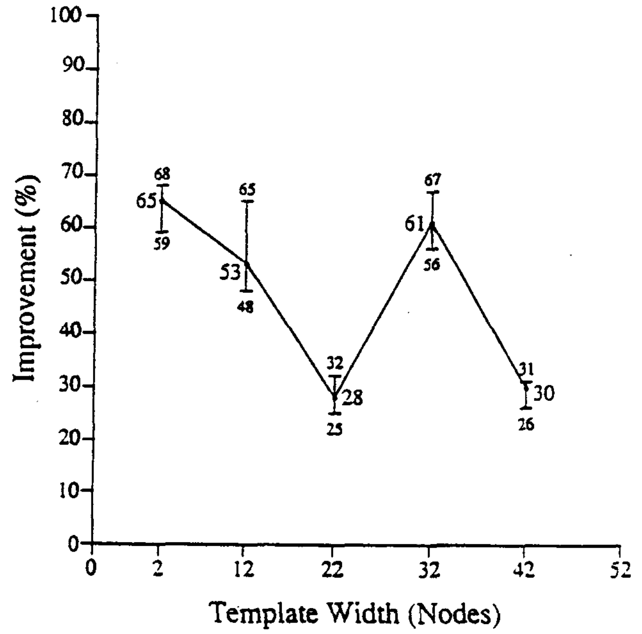

Figure 16. Improvement Due to Scheduling Under Varied Template Branching

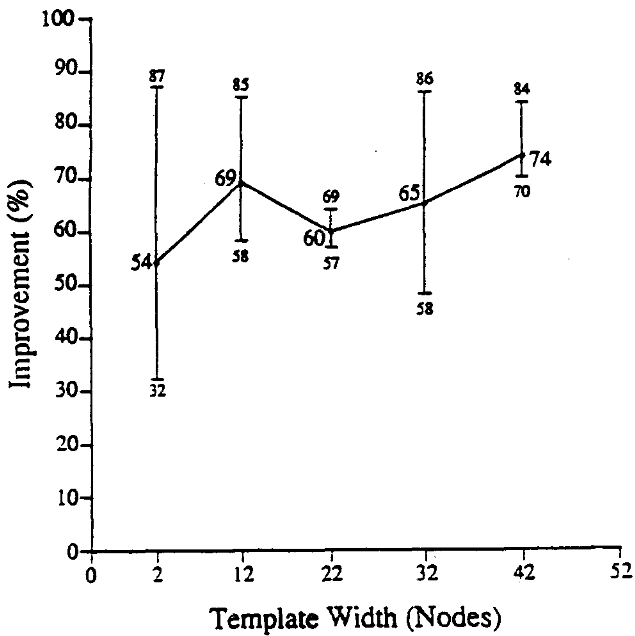

Figure 17. Improvement Due to Clustering Alone Under Varied Template Branching

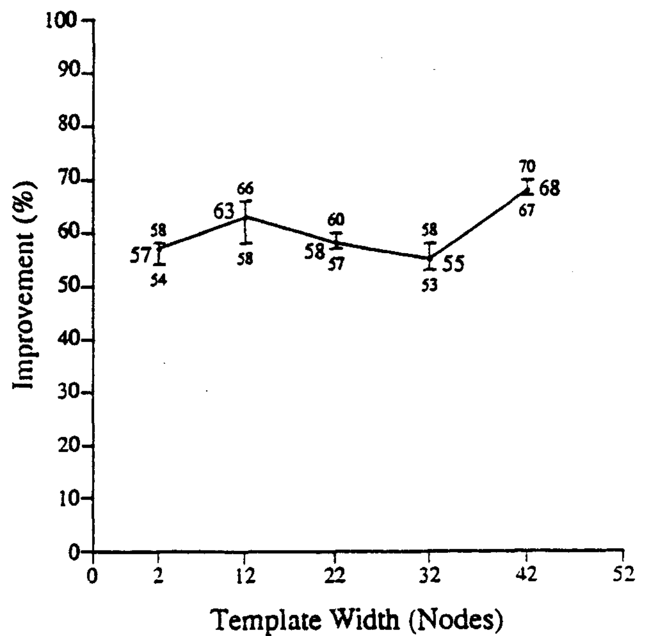

Figure 18. Improvement Due to Clustering With Scheduling Under Varied Template Branching

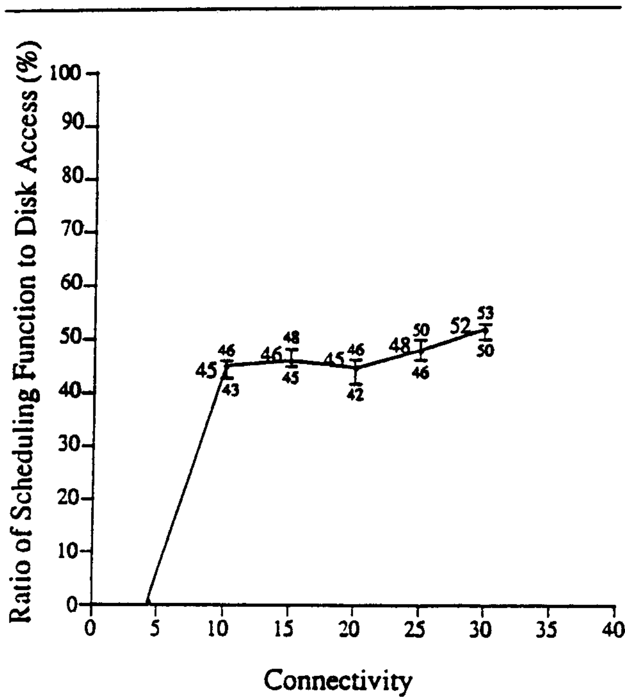

Figure 19. Ratio of Procedure Calls for Scheduling vs. Procedure Calls for I/O

References

[ACR88] B. Alpern, A. Carle, B. Rosen, P. Sweeney and K. Zadeck, "Graph Attribution as a Specification Paradigm", *Proceedings of the Symposium on Practical Software Development Environments*, Boston, November 1988, 121-129.

[BuC79] O. P. Buneman and E. K. Clemons, "Efficiently Monitoring Relational Databases", *Trans. Database Systems 4* (September 1979), 368-382.

[Che76] P. P. Chen, "The Entity-Relationship Model—Towards a Unified View of Data", *ACM Trans. on Database Systems 1*, 1 (1976), 9-36.

[DKB89] P. Drew, R. King and J. Bein, "A La Carte: A Workbench Environment for Rapid DBMS Construction and Experimentation ", *Working Paper*, 1989.

[FJL88] S. Ford, J. Joseph, D. E. Langworthy, D. F. Lively, G. Pathak, E. R. Perez, R. W. Peterson, D. M. Sparacin, S. M. Thatte, D. L. Wells and S. Agarwala, *ZEITGEIST: Database Support for Object-Oriented Programming*, Springer-Verlag , September, 1988.

[GoR83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass., 1983.

[HaM81] M. Hammer and D. McLeod, "Database Description with SDM: A Semantic Database Model", *ACM Trans. on Database Systems 6*, 3 (1981), 351-386.

[HuK86] S. E. Hudson and R. King, "CACTIS: A Database System for Specifying Functionally-Defined Data", *Proceedings of the Workshop on Object-Oriented Databases*, Pacific Grove, California, September 23-26, 1986, 26-37.

[HuK87] S. E. Hudson and R. King, "Object-oriented database support for software environments", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, San Francisco, California, May, 1987, 491-503.

[HuK88a] S. E. Hudson and R. King, "The Cactis Project: Database Support for Software Environments", *IEEE Transactions on Software Engineering 14*, 6 (June 1988), 709-719.

[HuK88b] S. E. Hudson and R. King, "An Adaptive Derived Data Manager for Distributed Databases", *Proceedings of the Second International Workshop on Object-Oriented Database Systems*, Bad Munster am Stein-Ebenburg, FRG, September 1988, 193-203 .

[Hud89] S. E. Hudson, "Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update", *University of Arizona Technical Report*, 1989. Tech. Rep. 89-12.

[HuK89] S. E. Hudson and R. King, "Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System", *ACM Transactions on Database Systems*, December, 1989.

[HuK87] R. Hull and R. King, "Semantic Database Modeling: Survey, Applications, and Research Issues", *ACM Computing Surveys*, September 1987, 201-260.

[KeP76] L. Kerschberg and J. E. S. Pacheco, "A Functional Data Base Model", Technical Report, Pontificia Universidade Catolica do Rio de Janeiro, Rio de Janeiro, Brazil, February, 1976.

[Rep82] T. Reps, "Optimal-time Incremental Semantic Analysis for Syntax-directed Editors", *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, Jan. 1982, 169-176.

[Rep84] T. W. Reps, *Generating Language-Based Environments*, MIT Press, Cambridge, Mass., 1984.

[Shi81] D. Shipman, "The Functional Data Model and the Data Language DAPLEX", *ACM Trans. on Database Systems 6*, 1 (1981), 140-173.