

Efficient Assembly of Complex Objects

Tom Keller
Oregon Graduate Institute
Goetz Graefe
University of Colorado
David Maier
Oregon Graduate Institute

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 90-023

December, 1990

Report published in *Proceedings of the ACM-SIGMOD International Conference on Management of Data*,
Denver, Colorado, May 1991.

Efficient Assembly of Complex Objects

Tom Keller
Oregon Graduate Institute
Goetz Graefe
University of Colorado
David Maier
Oregon Graduate Institute

Abstract

Although object-oriented database systems offer advantages over relational or record-oriented database systems, such as modeling facilities for complex objects, they are criticized for poor performance and query capabilities on set-oriented applications. The unacceptable performance is due in part to the object-at-a-time processing typically used by object-oriented database systems.

We believe that improved performance of object-oriented database systems depends partially on the efficient and selective retrieval of sets of complex objects from secondary storage. In this report, we present the method of complex object retrieval and assembly used in the Volcano query processing system and the Revelation project. We also present experimental results comparing set-oriented versus object-at-a-time complex object assembly.

1. Introduction

Relational database management systems provide a simple and well-understood model of data. The simplicity and theory of the relational model result in efficient implementations. However, relational database management systems are poorly suited for modeling more complex data such as those found in engineering applications. Object-oriented database systems have many advantages over traditional record-oriented database systems, most notably modeling facilities for complex objects, object identity, and encapsulated behavior. In a previous report, we have described the goals of the Revelation project and introduced a high-level vision of its query optimization scheme [1]. We believe that four concepts are crucial for the performance of object-oriented database systems. First, set-oriented processing allows leveraging expensive operations, e.g., disk seeks. Second, the retrieval and in-memory assembly of complex objects are very frequently used operations, therefore a determinant of performance. Third, query optimization and access planning, proven to be a cornerstone of relational systems performance, will gain even more importance for semantically richer queries and complex data. Fourth, parallel processing techniques can be exploited much more easily if the underlying processing paradigm uses sets that can be partitioned, rather than single object instances.

We report on set-oriented processing to improve complex object retrieval and assembly - a combination of the first two concepts listed above. We introduce an operator called the *assembly* operator, implemented on top of the Volcano query processing system. The assembly operator was designed to retrieve and assemble complex objects in a manner that outperforms non-set-oriented (*naive* or *object-at-a-time*) complex-object assembly, using physical and logical information such as object clustering and the degree of sharing between objects. In addition, the assembly operator is able to retrieve complex objects selectively, based on arbitrary selection predicates.

In the next section we briefly survey related work. Section 3 discusses background information from the Revelation Project and the Volcano query processing system. Section 4 details the assembly operator and its benefits over object-at-a-time assembly. Section 5 describes the data structure and algorithm used to drive complex object assembly. Section 6 provides a preliminary performance evaluation for the assembly operator. Directions for future research are presented in Section 7 and a summary and our conclusions are given in Section 8.

2. Related Work

Our design of the *assembly* operator was influenced mainly by the way look-up routines work for unclustered index scans, for example, the join called TID-scan in Kooi's thesis [2]. Scanning a file using an unclustered index is much more expensive than using a clustered index. One could try to avoid the seek costs of the unclustered scan by sorting the pointers retrieved from the index and looking them up in physical order. This approach, however, may require substantial sort space. We sought an operator that avoids the cost of completely sorting the pointer set, but retains the advantages of using an index. Once we had defined this operator, it was straightforward to extend the algorithm to complex objects. In this report, we put this algorithm into an extensible context to make it usable in an object-oriented database system.

Complex object assembly is closely related to the pointer-based join methods of relational database systems. Assembly resembles a functional join, linking objects based on inter-object references. An early pointer-based join optimization, join indices [3], maintained a pre-computed join by storing pointers to pairs of joining records. More recently, pointer-based joins [4,5,6] use either explicitly stored pointers or system maintained

pointers. We, however, do not require sets of objects to be confined to distinct disk files or that object references contain a physical component¹.

3. Background

The assembly operator was envisioned to meet the need for increased performance in the Revelation object-oriented query processor. Revelation types have encapsulated behavior and are a combination of *complex values* and *methods*. Objects reference other objects by embedding object identifiers (OIDs) in their state. An overview of Revelation's architecture is shown in Figure 1. A query can be executed naively within the run-time system or it can be "revealed." Revealing a query is an attempt to transform a query into its equivalent complex object algebra expression. In order to reveal behavioral information about a query, the encapsulation barrier must be broken by the *revealer*, a trusted system component. Once a query is transformed into the complex object algebra expression, it is optimized, by an optimizer generated optimizer [7,8]. Optimization includes choosing physical algebra operators, also called set processing methods, for the logical algebra (complex object algebra) operators. For example, a join operator at the logical level may be replaced by the hash-join operator

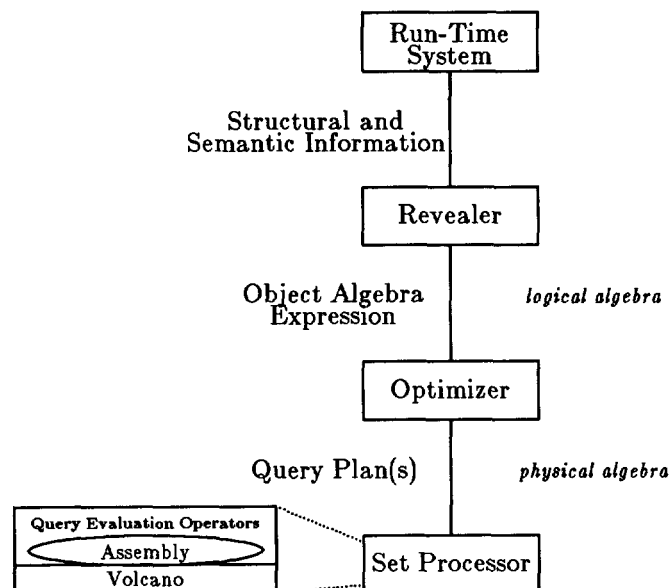


Figure 1. Revelation Architecture

¹ Only that there is a mapping from object reference to physical location.

in the physical algebra. The assembly operator is a set processing method that does not correspond to any complex object algebra operator². It is similar to a sort operator in relational systems where the operator enforces a physical property of the data that is not logically apparent (i.e. sort order). The assembly operator is used to prepare data needed by the other physical operators. It enforces the physical constraint: "The portion of the complex object needed to carry out the query is entirely in memory." By "portion" we mean part of the object's complex internal state plus fragments of referenced objects³.

Once a query has been optimized and exists as a tree of physical operators it is executed by the *set processor*. The set processor used in the Revelation project is based on the Volcano query evaluation system [9]. Volcano includes a file system with heap files, B-trees, and buffer management. The design of the Volcano query evaluation system was influenced by a number of systems, most notably WiSS [10] and GAMMA [11]. Volcano queries are composed of operators that provide a uniform iterator interface. Each Volcano operator conforms to the iterator paradigm by providing *open*, *next* and *close* calls. Other query processing systems that use the iterator paradigm, though in somewhat different ways than Volcano, are System R [12], the Ingres Corp. version of Ingres, EXODUS [13], and Starburst [14], where it is called "lazy evaluation."

4. The Assembly Operator

The purpose of the assembly operator is to efficiently translate a set of complex objects from their disk representations to a quickly traversable memory representation. *Complex object* refers to one or more objects or object fragments connected by inter-object references. Note that we do not restrict the implementation of inter-object references nor do inter-object references imply ownership as in *composite objects*. To achieve quickly traversable memory-resident complex objects, all object references (OIDs) are changed to memory pointers. This "pointer-swizzling" process results in a structure that can be scanned without the need to consult an OID-to-memory-address mapping table. Instead, complex object scanning is reduced to following memory pointers. Figure 2 illustrates a complex object where each box is an object and each inter-object reference is

² This is not strictly the case because the assembly operator does have the ability to perform a selection.

³ "Object" is used to label both application-level objects and storage-layer objects. An application-level object's state may be composed of many storage-layer objects.

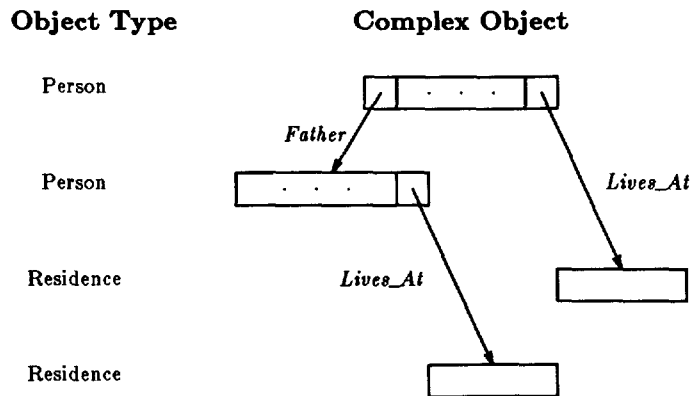


Figure 2. Sample Complex Object

shown as a directed edge. This particular figure should be interpreted as a **Person** and his/her father (who is also a **Person**) and the **Residence** of both child and father. The assembly operator has the ability to find and resolve multiple, possibly shared, object references contained within a single object. In order to do this efficiently, the operator uses physical, structural and statistical information.

We look first at a sample query on a set of complex objects of the form depicted by Figure 2: "Retrieve all people that live close to (live in the same city as) their father." A pseudo-code implementation of the top-level query and one method is shown in Figure 3. In the figure, the dot (".") is used to represent attribute selection and messages are written as C functions and are italicized. The **top-level-query** method iterates over all people. Each person in the set receives the **lives-close-to-father** message. When this query is executed naively, each complex object gets completely traversed before another is considered. Furthermore, the order that each complex object is traversed depends on how the methods were written and how the code was interpreted or

```

top-level-query:
  for each PERSON
    if ( lives-close-to-father (PERSON) )
      printout (PERSON)

lives-close-to-father:
  if ( city (self.residence) == home-town (self.father) )
    return (TRUE)

```

Figure 3. Partial Query Implementation

compiled. For example, *lives-close-to-father* may be compiled so that the *home-town* message is sent before the *city* message. If each person object is clustered with a referenced residence object, then sending the *city* message first would be more efficient. The point is, method interpretation/compilation does not order object fetches to improve efficiency.

The example above is equivalent to a set selection on a set of complex objects. Naive execution must traverse each complex object, one-at-a-time, to evaluate the query. However, an alternate way to compute the query is to assemble the complex objects and then carry out the selection. Preparing complex objects for subsequent query evaluation is the responsibility of the assembly operator. By using the assembly operator, the order that objects are fetched into memory is independent of the query implementation. That is, objects need not be fetched in the order that the query implementor fetched them. Instead, fetching order is restricted only by the structure of the complex objects.

At any stage of assembling a complex object there may be several references yet to be resolved. Returning to the previous example, after retrieving a single person either the father or the residence reference can be resolved. There are two primary reasons why this choice has a direct effect on performance. First, the physical location of objects on disk and in memory must be considered. If requested objects are contained in a single page, then only a single request should be issued to the buffer manager⁴. This situation occurs when objects are clustered together and when there is sharing of sub-objects. Intelligent scheduling of disk retrievals, based on physical location, can decrease the number of tracks covered and the total seek time. Furthermore, the larger the number of unresolved references to choose from, the greater the possibility to choose one with small seek distance. The second effect on performance arises from the existence of predicates. It is advantageous to abort the assembly of a complex object as soon as possible if it has a chance of not satisfying a selection predicate. Therefore, to prevent a waste of effort it is beneficial to retrieve sub-objects that have a high probability of failing a predicate as soon as possible [15]. For example, if the previous query was restricted to the state of

⁴ It can be argued that a second request is bound to be a buffer hit, therefore very inexpensive. Our experience shows, however, that even buffer hits can be expensive, since a table must be searched while protected against concurrent update, etc. While it is reasonable to expect that a buffer request can be serviced in less than 200 instructions if it does not result in a buffer fault, very frequent buffer hits can add significantly to overall query processing cost.

Oregon, the residence of the person should be fetched and checked before the person's father is considered.

So far we have only discussed assembling one complex object at a time. However, this approach is not likely to provide a large number of unresolved references. This problem is overcome by assembling more than one complex objects at a time. Instead of working on a single complex object, the assembly operator works on a window, of size W , of complex objects. As soon as any one of these complex objects becomes assembled and passed up the query tree, the operator retrieves another one to work on. We refer to this as a *delayed* or *sliding assembly operator*. Using a window of complex objects increases the pool size of unresolved references and results in more options, leading to a greater expected effect on optimization. A cost of using the sliding assembly operator is the need for enough buffer space to hold W partially assembled objects.

Consider a small example using three complex objects structured like the one shown in Figure 2. Suppose that the assembly operator is using a window size of 2. The three completely assembled complex objects and individual object symbols are shown in Figure 4. Assembly begins by filling the window with references to the first two complex objects. Figure 5a shows the starting condition. The assembly operator begins with a choice of two references to resolve. After resolving $A1$, and fetching the required piece of the object, two new unresolved references are added to the list. The new state is shown in Figure 5b. Next, $A2$ is resolved (Figure 5c), resulting in two more references being placed on the list. Figure 5d is the result of resolving $B1$ which places unresolved reference $D1$ on the list. After $C1$, $C2$ and $D1$ are resolved (Figure 5g), the first complex object is assembled. In order to keep a window size of 2, after the first complex object is passed to the next operator, a new reference is added to the list (Figure 5h).

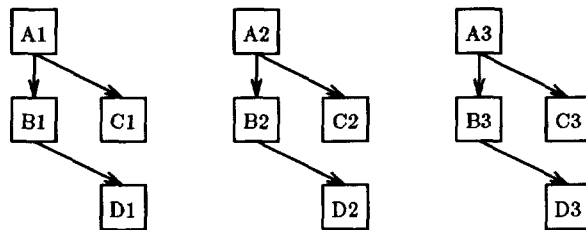


Figure 4. Three Assembled Complex Objects




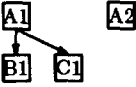

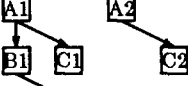

	Unresolved Reference List	Partially Assembled Complex Objects
a)	$A1, A2$	
b)	$A2, B1, C1$	
c)	$B1, C1, B2, C2$	
d)	$C1, B2, C2, D1$	
e)	$B2, C2, D1$	
f)	$B2, D1$	
g)	$B2$	
h)	$B2, B3, C3$	

Figure 5. Sliding Assembly Operator Example

The example depicts one order that results in the first complex object being assembled. There are, however, many other scheduling orders. At each step in Figure 5, an unresolved reference must be chosen. Ideally, the reference that reduces disk head movement and overall assembly time will be chosen. For example, *C2* may have been chosen from the list in Figure 5e because it was fetched when *C1* was fetched. The scheduling order is only restricted by the need to fetch objects top-down. In Section 7 we will discuss how this restriction can be overcome, allowing arbitrary scheduling order.

Assembly makes no assumptions about physical location of sub-objects. It is possible for a complex object to be partially assembled - perhaps by a previous operator. When a partially assembled sub-object is discovered, the operator finds all unresolved references within it. Suppose that objects *A1* and *B1* from Figure 4 are assembled. Assembly would begin with the following list instead of the one in Figure 5a: *A2, D1, C1*.

It may appear from the example that assembly reduces to a variant of an n-way pointer join. However, the example shows that results are produced without having to access all potentially participating objects. A

pointer join would require at least one input to be completely scanned before producing a single result. Assembly can touch a number of objects ranging from only those needed for one complex object up the entire window of complex objects. We also anticipate cases that require computations that are not algebraically expressible. For example, **lives-close-to-father** (Figure 3) may involve a distance computation based on latitude and longitude of the cities.

5. Component Iterator and Templates

In the previous discussion and example of the assembly operator, there was no mention of how the operator determines what part of a complex object to assemble, when assembly is complete or how to find unresolved references within a newly retrieved object. Such information is specific to each query and is type and structure dependent. In our design, these tasks are the responsibility of the *component iterator*, a companion routine to the assembly operator. Figure 6 shows the detailed architecture of the assembly operator and the component iterator. The component iterator uses structural and statistical information contained in a *template* to control

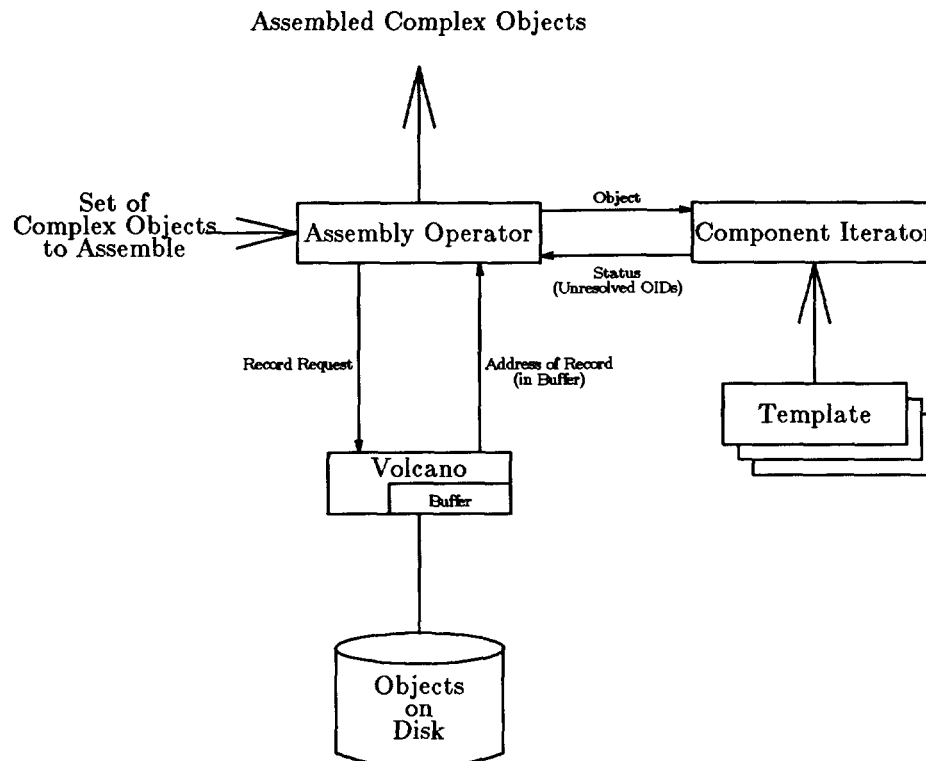


Figure 6. Assembly Operator and Component Iterator

the assembly operator. A template resembles a tree similar to the representation of a complex object shown in Figure 2. In addition to structural information, the template is annotated with statistical information. Currently the statistical information consists of the degree of sharing between objects and predicates with predicate selectivity.

The template captures two essential properties of complex objects pointed out by Batory [16]. The template allows *recursive* definitions and it indicates borders of *shared* components. Sharing information is important for three reasons. First, it will be necessary to ensure that such components are not loaded twice for two different objects into two different memory locations. Thus, some mechanism is required to determine whether shared components already reside in buffer memory. Second, a mechanism must be used to ensure that the shared component remains in memory as long as there is at least one valid reference to it from another object in memory, e.g., through reference counting. After a component is no longer referenced, it is subject to replacement using buffer replacement policies. Third, when the assembly operator runs in parallel, the original object identifier (OID) or fragments may be partitioned into disjoint subsets. Thus, shared components might be shared by objects in different partitions, and therefore introduce synchronization requirements between partitions that are unique to the assembly operator. Note that information on non-sharable objects is useful in avoiding the overhead of buffer lookup, reference counting and partitioning constraints.

The statistical information contained in the template will be used to decide the order in which component retrievals are scheduled. In particular, if the physical cost of retrieving two components is the same, it makes sense to retrieve the component that decides whether or not the other one is necessary. For example, if predicates are associated with both components, and the failure of either predicate allows abandoning assembly of the entire complex object, the component with the higher rejection probability should be retrieved first [15].

6. Performance of the Assembly Operator

The efficiency of the Volcano query processing software has been demonstrated in earlier reports, both for single-process and for multi-process query evaluation [9,17,18]. For this study, Volcano is used in single-process mode with parallelism and latching of internal data structures disabled.

Relational benchmarks such as the Wisconsin Benchmark [19] are clearly not well suited for measuring the performance of an object-oriented database system. Recent object-oriented benchmarks, the HyperModel Benchmark [20] and the Sun [21] benchmark, are better suited for our system. However, we wish to concentrate on complex object assembly and not on general query processing. Our solution was to develop a smaller group of benchmarks that focus on clustering, buffer size, window size, database size and scheduling algorithms.

Our benchmark most closely resembles the Altair Complex-Object Benchmark (ACOB) [22]. Each complex object is structured as a binary tree of 3 levels. However, unlike objects in ACOB, our objects are physically stored as a single record, not a group of seven records. Each object consists of 4 integer and 8 object reference fields equaling 96 bytes, resulting in 9 objects per page. This structure provides just enough depth and breadth to compare object-at-a-time assembly to other scheduling methods. It also provides enough structure so that clustering can be altered.

The primary consideration in deciding how to measure performance was the desire to compare object-at-a-time assembly to set-oriented assembly. Both methods have equivalent CPU costs⁵ so it is sufficient to compare the difference in I/O costs. For this reason, and the unavailability of a raw device for our use, performance is measured in terms of *average seek distance*, in pages of size 1K bytes. Average seek distance is the total seek distance divided by the total number of reads (average seek distance per read). We assume entire control over the queue of requests for the disk, making the seek time a significant cost in retrieving data from the disk [23].

As briefly mentioned above, our benchmark parameters are: clustering, scheduling algorithm, window size, buffer size and database size. Our first objective is to determine if one scheduling algorithm outperforms the others independent of clustering method, or if the scheduling algorithm needs to be adaptive to clustering. Also, as window size increases, we compare scheduling algorithm performance to object-at-a-time assembly. The second set of benchmarks test the hypothesis that information about shared sub-objects can be used to increase performance. And finally, selective assembly is tested using predicates with varying selectivities. Before presenting the results of the benchmarks we briefly describe the clustering methods and scheduling algorithms

⁵ The overhead of set-oriented assembly lies in the maintenance of a scheduling data structure (list, queue or priority queue).

used throughout the benchmarks.

6.1. Clustering

One important means of increasing the performance of queries is data clustering. A great deal of effort has been put into the research of optimal clustering methods for hierarchical objects [24,25,26,27]. However, even dynamic clustering algorithms, the most effective clustering method, while increasing performance of common queries, may result in a dramatic performance decrease of less common but still frequently used queries.

In the following discussion of clustering it will be useful to refer to Figure 7 which shows a set of n complex objects. The least restrictive form of clustering is *random* or *unclustered* data. Unclustered data is produced by randomly placing parts of each complex object on the disk. In Figure 8 and the following figures, an area boxed with a dashed line represents a cluster. Within a cluster the objects randomly placed. The simplest method of

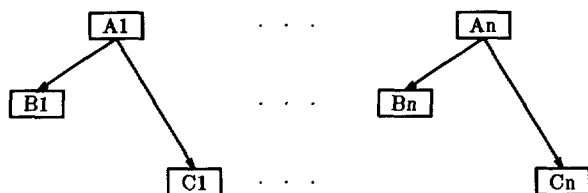


Figure 7. A Set of Complex Objects

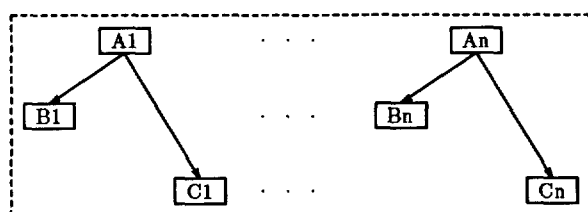


Figure 8. Unclustered Complex Objects

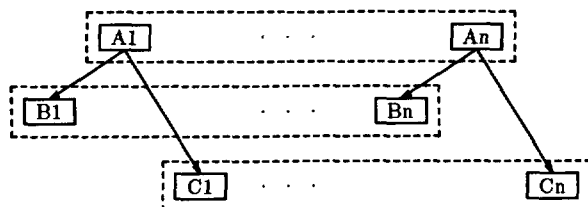


Figure 9. Inter-Object Clustering

clustering places objects of the same type, or class, together. We refer to this clustering policy as *inter-object* clustering, shown in Figure 9. Recall that there is no implied order within a cluster. The fact that object **A1** is the first object in Cluster A does not imply that object **B1** is the first object in Cluster B. Clustering some or all of the parts of a composite object together leads to the third form of clustering, *intra-object* clustering (Figure 10). This is a common form of clustering [28,29] used to increase the performance of queries that access a number of related objects at the same time.

6.2. Scheduling Algorithms

Performance is based primarily on I/O cost, so the order in which objects are fetched from disk is the key to performance. The order of object fetches is in turn dependent on the scheduling algorithm used in the assembly operator. We consider three simple scheduling algorithms in our benchmarks.

First, OIDs can be chosen for resolution by a *depth-first* traversal⁶ of each complex object. Recall the three complex objects shown in Figure 4. With a window size of 2 the objects will be resolved in the following, depth-first order: *A1, B1, D1, C1, A2, ...*. Note that depth-first scheduling is equivalent to object-at-a-time assembly, regardless of window size.

The second alternative is *breadth-first* scheduling. Using the same example, with a window size of 2, the references would be resolved in the following order: *A1, A2, B1, C1, B2, C2, D1, D2, A3, B3, C3, D3*. Note from the example that "breadth" refers to the breadth of the window and not the breadth of a single complex object.

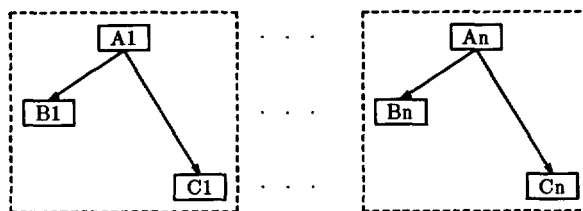


Figure 10. Intra-Object Clustering

⁶ Child order (left-to-right) is determined by the child reference storage order in the parent's state.

The third scheduling algorithm is an elevator algorithm, the SCAN scheduler, that schedules OIDs based on their physical location. This algorithm minimizes disk head movement, reducing the total seek time. Because we assume a dedicated device to store our database, there is no interference from other processing sources. Furthermore, with a sufficiently large window we can expect a large number of outstanding requests, making SCAN scheduling a reasonable choice [30].

6.3. Clustering vs. Scheduling vs. Window Size

The first group of benchmarks compare the performance of all three scheduling algorithms on all three clustering methods. There is enough buffer space to hold the largest database, so no page replacement occurs. Window sizes of 1, 50, 100, 150 and 200 complex objects are tested as well as database sizes of 1000, 2000, 3000 and 4000 complex objects. Recall that performance is measured as average seek distance (in pages) per read. We present our benchmark results when window sizes of 1 and 50 complex objects are used and show the effect of window size on scheduling algorithm performance.

6.3.1. Window Size = 1

With a window size of one complex object, all three scheduling algorithms assemble objects one-at-a-time. However, their performance is not identical, as shown in Figures 11(A-C).

The performance difference observed in Figure 11A is an artifact of the method used to assign references between inter-object clustered complex objects. Referring back to Figure 7, each object in Cluster A references an object in Cluster B, and an object in Cluster C. The clusters are placed on disk as shown in Figure 12. But each cluster is larger than the amount of valid data contained in it. Thus, in Figure 12, the shaded regions contain data and the unshaded area is unused. In fact, the cluster size is larger than any database size used in the benchmarks. Therefore, seek distance is independent of database size - shown by the flat lines in Figure 11A.

Breadth-first scheduling performs poorly for inter-object clustering because of cluster layout. Objects are fetched in the following order: Cluster A, Cluster B, Cluster C and Cluster D. However, the clusters are not physically placed in that order (Figure 12). The other two algorithms fetch from the clusters in the order they exist on disk, accounting for the performance difference in Figure 11A.

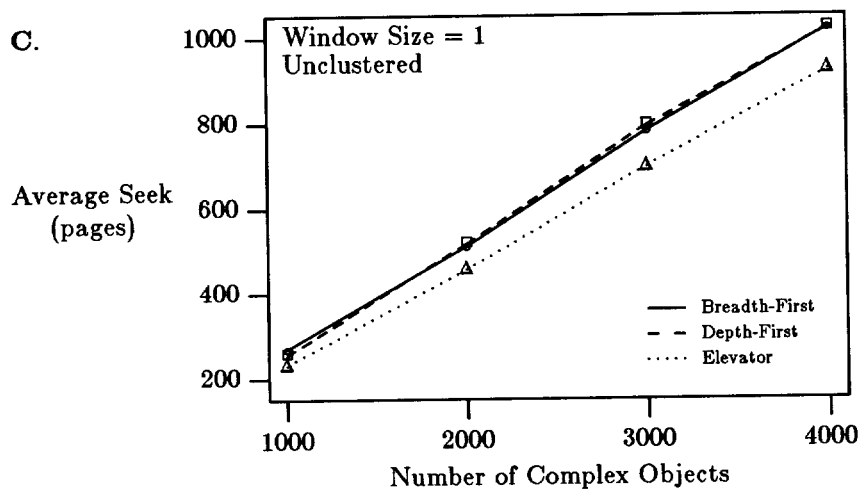
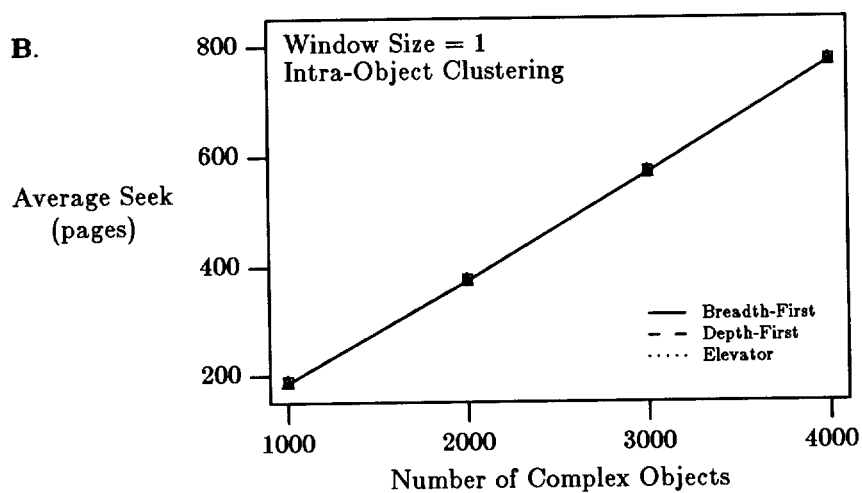
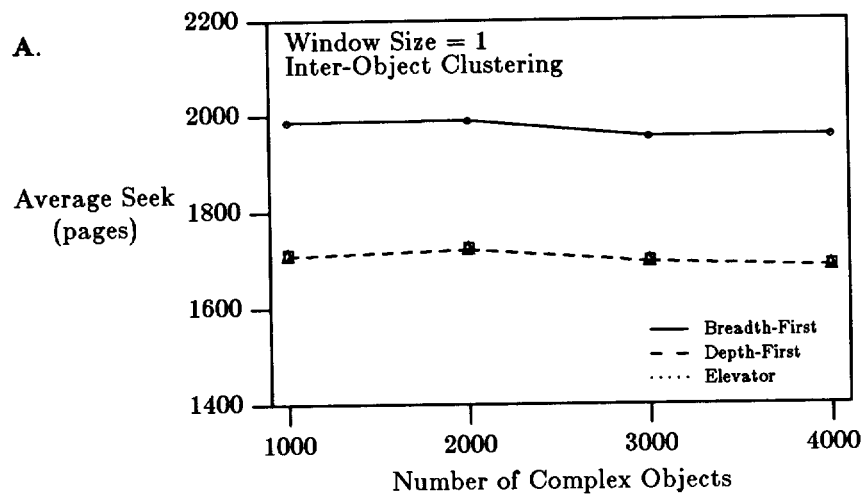


Figure 11. Scheduling Algorithm vs. Database Size (Window Size=1)

Disk:

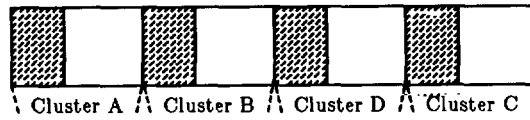


Figure 12. Disk Layout for Inter-Object Clustering

When the databases are unclustered, Figure 11C, the elevator scheduler uniformly decreases average seek distance by approximately 10%. It is possible, therefore, to gain a small performance increase on unclustered data just by scheduling object fetches based on their physical location.

6.3.2. Window Size = 50

Figures 13(A-C) show benchmark results for a window of 50 complex objects. Regardless of how the data is clustered, average seek distance is smallest for elevator scheduling.

Elevator scheduling, combined with a window of more than one complex object, orders object fetches almost identically to the "ideal" scheduling algorithm. For example, disk head movement is reduced for inter-object clustering because objects in the current cluster are fetched before those in another cluster (breadth-first scheduling). The elevator algorithm goes one step further by fetching all clustered objects in physical order. Similarly, depth-first (object-at-a-time) scheduling is suited for inter-object clustering. However, the order of assembly does not match physical order for depth-first scheduling as it does for elevator scheduling.

6.3.3. Window Size vs. Scheduling Algorithm

Under all clustering policies and window sizes, elevator scheduling is the most efficient of the three scheduling algorithms. Thus, the elevator algorithm is used to measure the effect of window size on performance. Figure 14 shows the results of varying window size for a constant database size of 4000 complex objects.

The point of diminishing returns occurs prior to a window of 50 complex objects. Window size increase beyond this point marginally decreases average seek distance while costing more buffer space (to hold partially assembled complex objects). For example, at most 7 pages are required with a window size of one complex object. When the window size is 50, up to

$$\left[6 \times 49\right](\text{pages for uncompleted objects}) + \left[7 \times 1\right](\text{pages for completed objects}) = 301 \text{ pages}$$

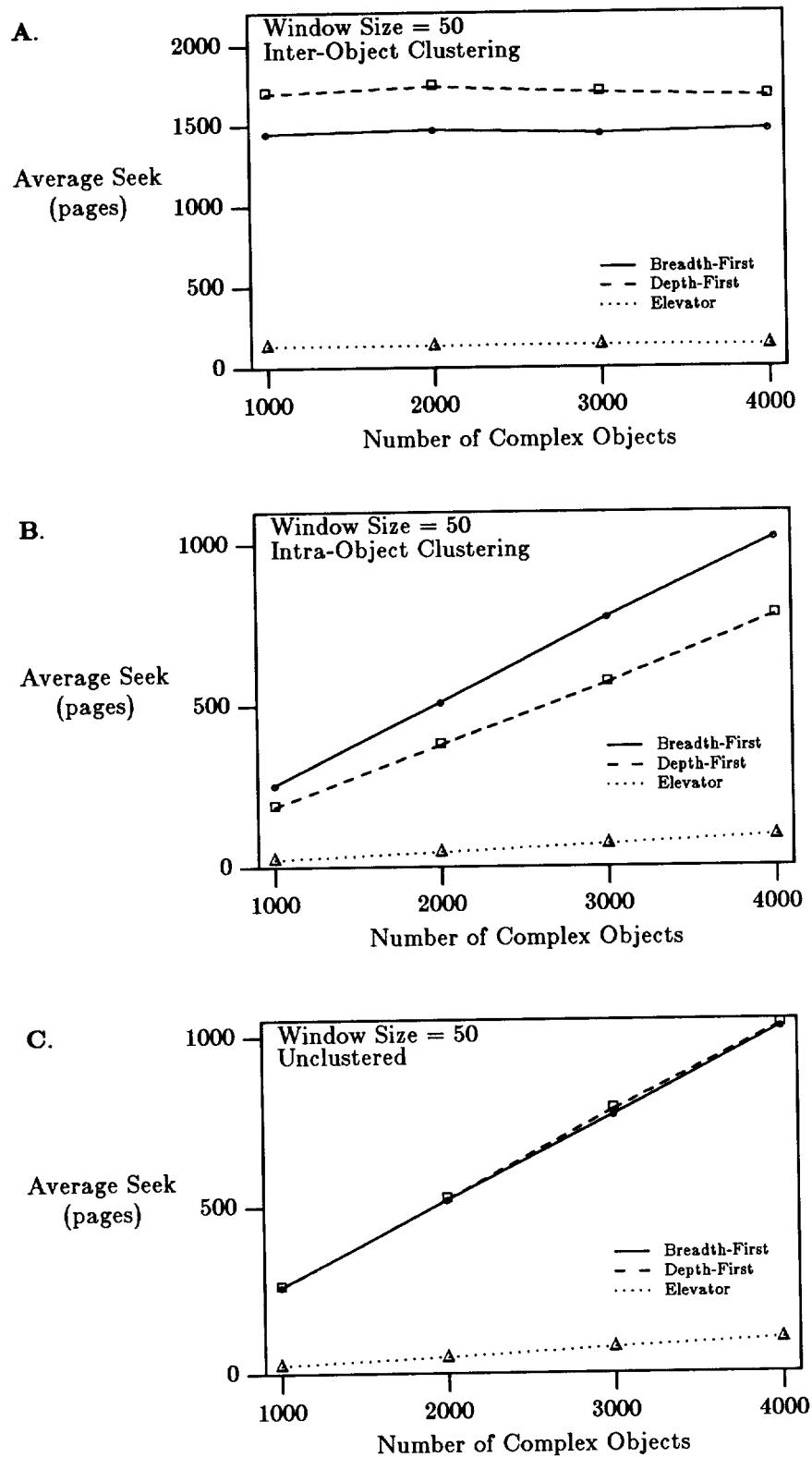


Figure 13. Scheduling Algorithm vs. Database Size (Window Size=50)

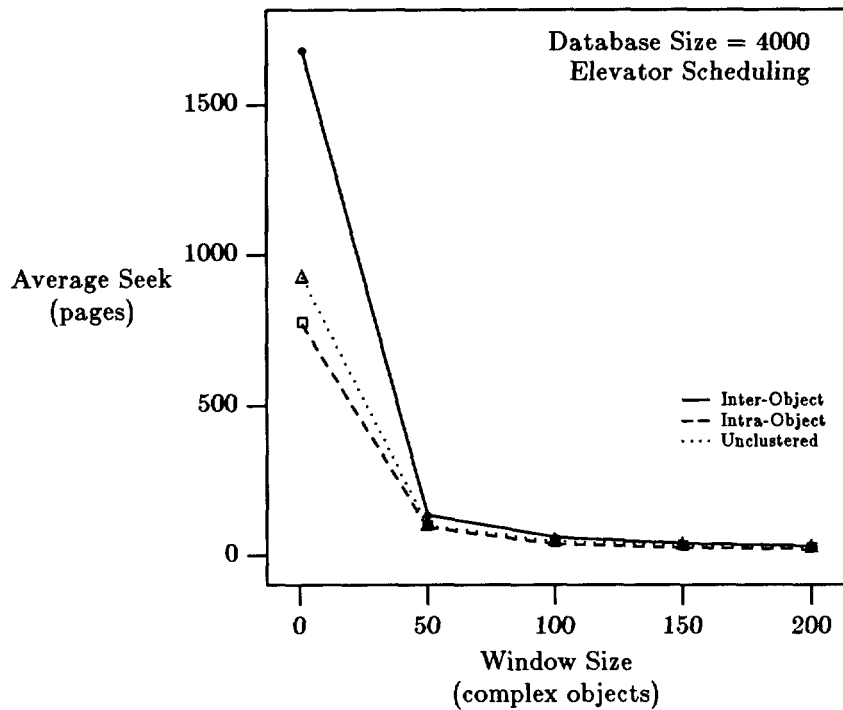


Figure 14. Database Size = 4000, Elevator Scheduling

may be needed.

6.4. Sharing of Sub-Objects

The previous benchmarks assume no sub-objects are shared by complex objects. However, this assumption cannot be made for complex objects in a realistic object-oriented database. To improve efficiency, complex object assembly uses sharing statistics contained in a template. Sharing statistics are used during assembly to predict buffer usage and prevent shared objects from being flushed out of the buffer.

As in the previous benchmark, elevator scheduling and object-at-a-time (depth-first) scheduling are compared. Inter-object clustering is used for simplicity. Sharing is the ratio of *shared* objects to *sharing* objects. For example, 100 objects sharing 5 sub-objects exhibit .05 sharing.

One of the benchmark results, using .25 sharing, is shown in Figure 15. The results shown in this figure are typical of the other benchmarks with differing degrees of sharing. Not only does the use of expected sharing statistics increase performance, it also reduces the total number of reads (not apparent in Figure 15).

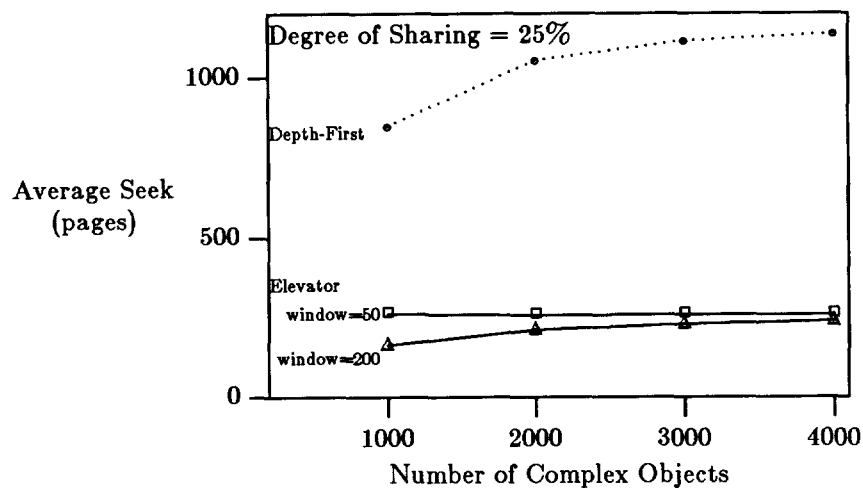


Figure 15. Performance of Databases Containing Shared Objects

6.5. Predicates and Selectivity

A key ability of the assembly operator is that it can selectively assemble complex objects. This is analogous to evaluating predicates while scanning or retrieving a relation in a relational database system. These benchmarks compare the performance of elevator scheduling to object-at-a-time assembly when complex objects must satisfy predicates of varying selectivities.

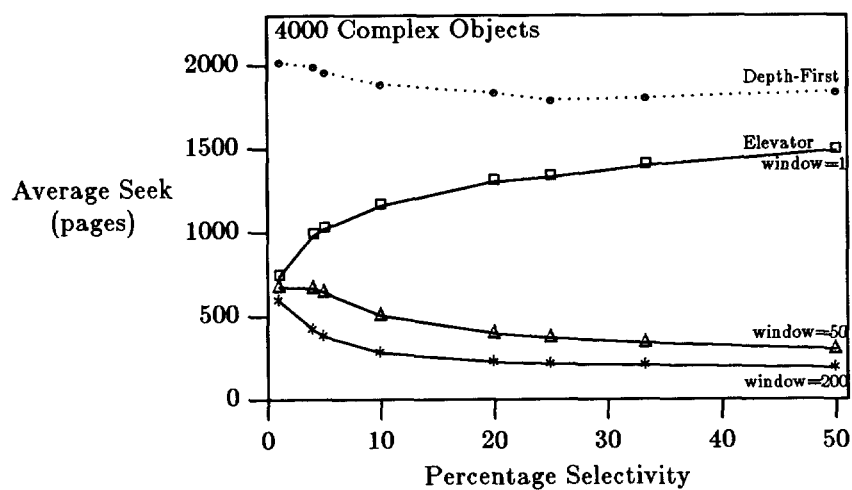


Figure 16. Predicates and Selectivities

Figure 16 shows the results of running these benchmarks. We see a decrease in average seek distance with an increase in the number of complex objects, for window sizes greater than 1. The reason, fewer reads are needed for assembling fewer objects. Object fetches other than those needed to test the predicate or completely assemble complex objects satisfying the predicate are eliminated by first fetching objects needed to evaluate the predicate.

7. Future Directions

The impact of a restricted or varying buffer size has not been explored. As explained previously, increasing the window size results in a need for more buffer space. If no more buffer space is available, then some pages will have to be released and re-read. The use of the elevator algorithm will help to prevent the flushing of useful pages by resolving all references in the current set of buffer pages. We suspect that for a given buffer size the window size can be tuned so that performance is maximized.

Currently, assembly operates entirely with one scheduling algorithm. Also, scheduling priorities based on shared sub-objects and predicates have not been integrated into a single scheduling algorithm. The primary scheduling algorithm will be the elevator algorithm modified to account for predicates, sharing and the buffer size. For example, although elevator scheduling orders object fetches to minimize disk head movement, the order may be altered to abort assembly of complex objects failing a predicate. Additional template annotations may be required to make intelligent scheduling decisions.

A further research topic is the implementation of a parallel complex object assembly operator. Every effort has been made to isolate points where locking must occur when concurrent assembly processes access shared sub-objects. Since parallelism is encapsulated in Volcano [31], it can be used for all existing operators without changing their code; we anticipate that it will also allow parallelizing the assembly operator to provide further speedup.

Implementing a parallel assembly operator poses the same obstacles as allowing multiple assembly operators in a Volcano query plan. The effectiveness of elevator scheduling depends on exclusive control of the physical device. When multiple assembly operators (or parallel invocations of a single assembly operator) are executing, each assumes sole control of the device and independently issues object fetch requests. Therefore, there are

two or more independent queues of requests for the device and the exclusive control assumption no longer holds. The situation becomes more complex when the database is stored on more than one physical device. At present, the assembly operator can only handle one device. A possible solution could involve a server-per-device architecture. Each server would maintain a queue of requests and would fetch objects on behalf of one or more assembly operators.

Previously, we stated that assembly had to occur top-down. When multiple assembly operators are allowed in a query tree, complex objects may be assembled bottom-up. Bottom-up and top-down assembly is achieved by "stacking" assembly operators. Suppose that the *B* and *D* sub-objects from Figure 4 should be assembled bottom-up. This is accomplished by using the two assembly operators shown in Figure 17. Assembly1 assembles all *B* and *D* objects according to the template and passes them to Assembly2. Assembly2 completes the assembly by fetching *A* and *C* objects and linking them with the sub-objects already assembled by Assembly1.

8. Summary and Conclusions

In this report on the Revelation project, we have outlined techniques for set processing and complex object retrieval. The set processor is based on iteration over sets, using the iterator or demand-driven dataflow paradigm as implemented in Volcano.

The assembly operator uses templates and component iterators to selectively and intelligently assemble complex objects. Once assembled, complex objects may be efficiently traversed. Elevator scheduling, combined with a *sliding window* of complex objects, reduces disk head movement compared to object-at-a-time assembly.

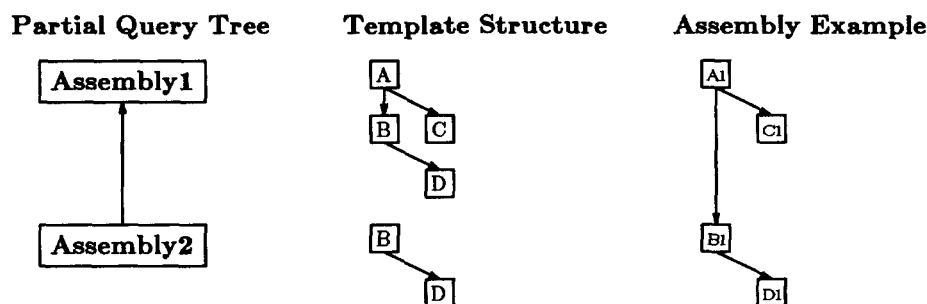


Figure 17. Combination of Bottom-up and Top-down Assembly

Furthermore, elevator scheduling outperforms *depth-first* and *breadth-first* scheduling when differing data clustering policies are used. Predicates are used to abort assembly of failing complex objects as soon as possible - reducing the number of unnecessary object fetches. Shared sub-objects are assembled and kept in the buffer as long as possible using sharing statistics.

If this technique is combined with parallelism through partitioning and asynchronous I/O, both provided as standard services in Volcano, we expect that the assembly operator will retrieve large sets of complex objects with scalable performance.

References

1. G. Graefe and D. Maier, "Query Optimization in Object-Oriented Database Systems: A Prospectus," pp. 358-363 in *Advances in Object-Oriented Database Systems*, ed. K.R. Dittrich, Springer-Verlag (September 1988).
2. R.P. Kooi, "The Optimization of Queries in Relational Databases," *Ph.D. Thesis*, Case Western Reserve University, (September 1980).
3. P. Valduriez, "Join Indices," *ACM Transaction on Database Systems* **12**(2) pp. 218-246 (June 1987).
4. M.J. Carey, E. Shekita, G. Lapis, B. Lindsay, and J. McPherson, "An Incremental Join Attachment for Starburst," *Sixteenth International Conference on Very Large Data Bases*, p. 662 (1990).
5. L. Haas, W. Chang, G. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M.J. Carey, and E. Shekita, "Starburst Mid-Flight: As the Dust Clears," *IEEE Transactions on Knowledge and Data Engineering* **2**(1) pp. 143-160 (March 1990).
6. Eugene J. Shekita and Michael J. Carey, "A Performance Evaluation of Pointer-Based Joins," *Proceedings of the ACM SIGMOD Conference*, p. 300 (May 1990).
7. G. Graefe and D.J. DeWitt, "The EXODUS Optimizer Generator," *Proceedings of the ACM SIGMOD Conference*, pp. 160-171 (May 1987).
8. G. Graefe, "Rule-Based Query Optimization in Extensible Database Systems," *University of Wisconsin-Madison, Ph.D. Thesis*, (August 1987).
9. G. Graefe, "Volcano, An Extensible and Parallel Dataflow Query Processing System," *submitted for publication, also CU Boulder Comp. Sci. TR 481*, (July 1990).
10. H.T. Chou, D.J. DeWitt, R.H. Katz, and A.C. Klug, "Design and Implementation of the Wisconsin Storage System," *Software - Practice and Experience* **15**(10) pp. 943-962 (October 1985).
11. D.J. DeWitt, R.H. Gerber, G. Graefe, M.L. Heytens, K.B. Kumar, and M. Muralikrishna, "GAMMA - A High Performance Dataflow Database Machine," *Proceedings of the Conference on Very Large Data Bases*, pp. 228-237 (August 1986).
12. M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, and V. Watson, "System R: A Relational Approach to Database Management," *ACM Transactions on Database Systems* **1**(2) pp. 97-137 (June 1976).
13. J.E. Richardson and M.J. Carey, "Programming Constructs for Database System Implementation in EXODUS," *Proceedings of the ACM SIGMOD Conference*, pp. 208-219 (May 1987).
14. L.M. Haas, W.F. Cody, J.C. Freytag, G. Lapis, B.G. Lindsay, G.M. Lohman, K. Ono, and H. Pirahesh, "An Extensible Processor for an Extended Relational Query Language," *Computer Science Research Report*, (RJ 6182 (60892)) IBM Almaden Research Center, (April 1988).

- 14.
16. D.S. Batory and A.P. Buchmann, "Molecular Objects, Abstract Data Types and Data Models: A Framework," *Proceedings of the Conference on Very Large Data Bases*, pp. 172-184 (August 1984).
17. T. Keller and G. Graefe, "The One-to-One Match Operator of the Volcano Query Processing System," *Oregon Graduate Center, Computer Science Technical Report*, (89-009)(June 1989).
18. G. Graefe, "Parallel External Sorting in Volcano," *submitted for publication, also CU Boulder Comp. Sci. TR 459*, (February 1990).
19. D. Bitton, D.J. DeWitt, and C. Turbyfill, "Benchmarking Database Systems: A Systematic Approach," *Proceeding of the Conference on Very Large Data Bases*, pp. 8-19 (October-November 1983).
20. T.L. Anderson, A.J. Berre, M. Mallison, H. Porter, and B. Schneider, "The HyperModel Benchmark," *Proc. Int'l Conf. on Extending Data Base Technology*, (March 1990).
21. R.G.G. Cattell, "Object-Oriented DBMS Performance Measurement," pp. 364-367 in *Advances in Object-Oriented Database Systems*, ed. K.R. Dittrich, Springer-Verlag (September 1988).
22. D.J. DeWitt, P. Futersack, D. Maier, and F. Velez, "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems," *Sixteenth International Conference on Very Large Data Bases*, p. 107 (1990).
23. R.A. Scranton, D.a. Thompson, and D.W. Hunter, "The Access Time Myth," *IBM Technical Report RC 10197 (#45223)*(September 1983).
24. J. Banerjee, W. Kim, S.J. Kim, and J.F. Garza, "Clustering a DAG for CAD Databases," *IEEE Transactions on Software Engineering* 14(11) p. 1684 (November 1988).
25. E. Chang and R. Katz, "Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS," *Proceedings of the ACM SIGMOD Conference*, p. 348 (May-June 1989).
26. Veronique Benzaken and Claude Delobel, "Dynamic Clustering Strategies in the O₂ Object-Oriented Database System," *Rapport Technique* 34-89, Altair (18 aouat 1989).
27. P. Drew, R. King, and S. Hudson, and 135, "The Performance and Utility of the Cactis Implementation Algorithms," *Sixteenth International Conference on Very Large Data Bases*, (1990).
28. M.F. Hornick and S.B. Zdonik, "A Shared, Segmented Memory System for an Object-Oriented Database," *ACM Transactions on Office Information Systems* 5(1) pp. 70-95 (January 1987).
29. W. Kim, J. Banerjee, H.T. Chou, J.F. Garza, and D. Woelk Composite Object Support in an Object-Oriented Database System, *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 118-125 (October 1987).
30. T.J. Teorey and T.B. Pinkerton, "A Comparative Analysis of Disk Scheduling Policies," *Communications of the ACM* 15(3) pp. 177-184 (March 1972).
31. G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System," *Proceedings of the ACM SIGMOD Conference*, p. 102 (May 1990).