Set Processing and Complex Object Assembly in Volcano and the REVELATION Project

Goetz Graefe

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 89-013

June, 1989

Set Processing and Complex Object Assembly in Volcano and the REVELATION Project

Goetz Graefe

Oregon Graduate Center
Beaverton, Oregon 97006-1999
graefe@cse.ogc.edu

Abstract

The REVELATION project aims at combining the conceptual power of encapsulated behavior in object-oriented database systems with query optimization and set-oriented processing. Volcano is an extensible query processor developed for education and research. In this report, we describe evaluation of complex expressions over sets of complex objects, and selective and efficient retrieval of a set of complex objects and their components into memory. Assembling complex objects in buffer memory is widely regarded as essential for increasing performance in object-oriented database systems.

1. Introduction

Object-oriented database systems have many advantages over traditional record-oriented database systems, most notably modelling facilities for complex objects, object identity, and encapsulated behavior. In a previous report, we have described the goals of the REVELATION project and gave a high-level vision of how its query optimization scheme might work [1]. We believe that four concepts are crucial for the performance of object-oriented database. First, set-oriented processing allows leveraging expensive operations, e.g., disk seeks. Second, parallel processing techniques can be exploited much more easily if the underlying processing paradigm uses sets that can be partitioned, not instances. 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, complex object retrieval and assembly in memory is a very frequently used operation, therefore a determinant of performance.

In this report, we focus on set processing and complex object retrieval, and detail their implementation in Volcano and the REVELATION project. Volcano is an extensible query processor developed for education and research. In the next section, we briefly survey related work. Sections 3 and 4 describe set processing in general and complex object retrieval in particular. Section 5 provides an analytical performance evaluation for the assembly operator. A summary and our conclusions are given in Section 6.

2. Previous Work

The design of the Volcano query processing system was influenced by a number of systems, most notably WiSS [2] and GAMMA [3]. Other query processing systems that use the iterator paradigm, though in somewhat different ways than Volcano, are System R [4], the RTI version of Ingres, EXODUS [5], and Starburst [6], where it is called "lazy evaluation."

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 [7]. It is well known that scanning a file using an unclustered index is much more expensive than using a clustered index. One could try to avoid the seek costs by sorting the pointers retrieved from the index and look them up in physical order. This procedure, however, may require substantial sort space and loses one of the main advantages of using indices, namely that the result is delivered in index order. We started out trying to find an operator that avoids the cost of completely sorting the pointer set, but retains the advantages 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.

3. Set Processor and File System

The REVELATION project got its name from *revealing* encapsulated behavior to a central system component. The language in which revealed behavior is expressed is an algebra, similar to relational algebra but suitably generalized for complex objects. This algebra can be optimized with an algebraic optimizer like the EXODUS optimizer generator [8, 9]. The optimization consists of two tasks, which are actually performed in an interleaved fashion. First, operators in a complex expresion or tree are reorderd. Second, these operator are mapped to algorithms. We call the former the abstract or logical operators and the latter algorithms, set processing methods, or concrete or physical operators. As an example from the relational world, *join* is a logical operator while *merge join* is a concrete operator.

In the following, we assume that the query has been optimized and is specified by an expression or tree of concrete operators. Such an expression is executed by REVELATION's *set processor* which is based on the Volcano query processing software [10]. Volcano includes a file system with heap files, B-trees, and buffer

management. Volcano's main query processing concepts are *iterators*, *streams* and *support functions*. Iterators are generalizations of program loops; in Volcano, they are processing modules that both use and provide *open*, *next*, and *close* functions. Iterators can be combined into a tree, similar to algebra expressions. When *open*ing the top iterator of an expression, it *open*s its input(s), etc. The *next* operation evaluates the expression until it can return one data item. Calling *next* repeatedly on the top iterator returns all data items selected, created, or processed by the expression.

Streams are the sets of objects that are passed between iterators. Thus, we can say that an iterator consumes zero or more streams and produces a stream. The item passed between operators is a pair of record identifier and main memory address. The data item is pinned in the buffer pool at the address given, and can be unpinned using the record identifier. Each record pinned in the buffer pool is "owned" by exactly as many iterators as it is pinned. When passing a record between iterators, the "ownership" is transferred. Thus, the producing iterator may not unpin this record in the future, and the consuming iterator must either pass it on or unpin it.

Support functions are functions invoked by iterators to perform operations specific to the request or query and to the type of data item being processed, e.g., evaluation of a selection predicate. When called, a support function is provided the object or objects to be manipulated and an argument. The argument is fixed for all items processed by the iterator, but it may be a pointer to a data structure that the support function modifies or uses for its state. It can be a constant, e.g., for comparison predicates, or code interpreted by the support function. The latter alternative allows using one general interpreter for all object manipulations by providing the interpreter's entry point whereever a support function is required, and specifying the required object manipulation code in the argument.

Since iterators use the same interface for input and output, namely streams and the open-next-close protocol, they can be nested into complex expressions. A complex query can be expressed using a multi-level tree. Query evaluation is entirely self-scheduling, since the top operator "knows" when it needs to request another data item from its inputs, and similarly, recursively down the expression tree. Using this demand-driven dataflow paradigm in which records are only pinned in the buffer as long as they are needed by some iterator

ensures that the time-space product for intermediate results is minimal. Scheduling and synchronization of two iterators costs as little as one procedure call, truly a very small overhead.

Volcano's query processing is based on dataflow and sets. Both concepts are well suited to parallel execution. Consequently, we implemented both vertical parallelism and horizontal parallelism in Volcano. Vertical parallelism is implemented in form of multiprocess pipelines. Horizontal parallelism is available as inter-operator and intra-operator parallelism. The former means that multiple operators in a bushy expression are evaluated in parallel. The latter means that multiple processes participate in an iterator, preferably on disjunct subsets or *partitions*. In Volcano, both vertical and horizontal parallelism are encapsulated in a single module called *exchange* that uses the demand-driven stream interface for input and output, but uses data-driven dataflow with flow control between processes [11].

Volcano already includes a large number of iterators, e.g., file and index scan, filter, sort, join, set operations like intersection, difference, and union, duplicate elimination, and aggregate functions. For efficient retrieval of complex objects we plan to extend it with a special iterator called the *assembly* operator.

4. The Assembly Operator

The *assembly* operator is used to materialize complex objects in buffer memory. It is an iterator like all other operators; it consumes a stream of OID's or incomplete object fragments containing OID's and produces complete or more complete objects. In this regard, it resembles resolving index entries, e.g., the TID scan in Kooi's thesis and RTI Ingres [7, 12], or the functional join proposed or implemented in other database systems, e.g., for GEM [13, 14]. However, it is much more general and powerful.

First, in order to be effective in an OODBMS, it must be able to perform more than one functional join. It must be able to resolve multiple OID's from a single object fragment, and it must be able to retrieve multiple levels of subcomponents. Second, it must be able to retrieve components selectively. For example, if objects are to be selected depending on the properties on component A, subcomponent retrieval should be abandoned immediately for an object after component A is retrieved and found not to qualify. Third, in order to achieve high performance, it is necessary to schedule component retrievals intelligently, in particular with regard to shared subcomponents and to location of components on secondary storage.

Currently, we are designing and implementing such an operator. Our design includes the concepts of an *annotated structure expression* and a *component iterator*. The structure expression captures the object structure and relates the form of original and resulting fragments. In this respect, it is a tree similar to the syntax tree of a nested structure or record definition.

Beyond the purely structural information, the annotated structure expression also contains conditions and statistical information with each of its structure-substructure links. The conditions or condition functions are tested by the component iterator before the substructure is retrieved. The statistical information includes the degree of sharing of substructures by structures and the probability that a predicate is true, and is used in scheduling heuristics.

The structure expression captures two essential properties of complex objects pointed out by Batory [15]. The structure expression allows *recursive* definitions and it indicates borders of *shared* components. Such borders of sharing are very 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 at least as long as there is a valid reference to it from another object in memory, e.g., reference counting. After a component (or its page in the buffer) is not referenced any longer, it is subject to replacement using standard buffer replacement policies such as LRU or MRU. Third, when the assembly operator runs in parallel, i.e., the original OID or fragment set it partitioned into disjoint subsets, shared components might be shared by objects in different partitions, and therefore introduce synchronization requirements between partitions that does not exists for any of the existing operators.

The statistical information contained in the structure expression will be used to decide the order in which component retrievals are scheduled. In particular, if the costs for retrieving two components are the same, it makes sense to retrieve first 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 to abandon assembly of the entire complex object, the component with the higher rejection probability should be retrieved first [16].

The component iterator is different than the other iterators in Volcano. It does not consume a stream or read a file; instead, in produces a stream of OID's from the structure expression and an object fragment in buffer memory. It is not clear whether or not we need functions corresponding to *open* and *close*. Probably, the implementation of the assembly iterator will be easier without them. The *next* function of the component iterator produces component OID's and places where such components are integrated into the buffer representation of the complex object.

The component iterator has four possible outcomes. First, it can return a new OID to be retrieved to assemble the complex object (return *OKAY*). Second, it can signal that the complex object is assembled as much as required (return *DONE*). Third, it can flag that more subcomponents will be needed, but their OID's cannot be determined until other components have been materialized in the buffer (return *SUSPEND*). Finally, the component iterator can determine that the entire object currently being assembled does not satisfy a predicate because of a component value, and instruct the operator to release all components (return *ABANDON*). We will say more on disassembly later.

For example, consider an original fragment A with components B and C, where B has a component D. A is passed to the assembly operator when it calls the *next* function on its input operator. Assembly repeatedly calls the *next* function of its component iterator for A, which produces B and C and returns *SUSPEND* for the third call. The assembly operator now retrieves the known components, namely B and C. Afterwards, it tries the component iterator again, which first produces the reference to D and then returns *DONE*. The assembly operator retrieves D, links it into the complex object, and passes the entire object to the next operator in the query tree. If there were a predicate on D, the component iterator must return *SUSPEND* to ensure that it is called again after D has been retrieved. At this point, the component iterator can return *DONE* or *ABANDON*.

At any point of time, there are (hopefully) several open references to be resolved. The order in which they are resolved has a significant impact on the performance of the assembly operator for a number of reasons. First, consider the effect of locality. If requested components reside on a shared page, it clearly is advantageous

to retrieve them together by requesting this page only once from the buffer manager[1].

Second, consider seeking on a moving-head disk. If disk requests can be serviced in a free order, rather than in one particular sequence, the number of tracks covered and the total seek time decrease. For this reason, we are interested in knowing as many unresolved references as possible at any point of time.

Third, consider the effect of predicates. Clearly, it is desirable to abort complex object assembly as early as possible if an object does not qualify under some selection predicate. Therefore, it is necessary to retrieve those components early that have a high probability of aborting the assembly before much time and effort has been spent on it [16].

Even when using the component iterator for complex object assembly, only a small number of references may be unresolved at a time. Therefore, the performance gain by intelligent scheduling may be rather limited. However, we intend to introduce a *delayed* or *sliding assembly operator*. Instead of requesting one OID or fragment, assembling the complex object, and passing it on, the sliding assembly iterator first consumes a number of objects, say W, and keeps resolving references in them until one of them is completely assembled. This one is passed on, and a new OID or fragment is requested from the input. In this fashion, W objects are assembled in parallel, and W times as many unresolved references are available for the scheduler to choose from. The disadvantage of sliding assembly is that it requires more buffer memory for partially assembled objects. In order to make best use of the sliding assembly operator, the parameter W must be tuned carefully. We intend to conduct both analytical and experimental studies with this operator.

Since parallelism is encapsulated in Volcano [11], it could be used for all existing iterators without changing their code; we anticipate that it will also allow parallelizing the assembly operator to provide further speedup. We realize, however, that multiple processes requesting pages from one disk may cause interference and decreased performance. However, in the case of assembly, parallelism cannot be entirely encapsulated because the hash table used to find objects already in memory must be protected against concurrent update.

[1] 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.

Management and protection of this hash table will be implemented in the same way using the same mechanisms already used for the tables in Volcano's buffer manager.

In the algorithm description so far, we have not considered how complex objects are represented on disk and in the buffer. For the representation on disk, we intend to use object identifiers (OID's). We anticipate that OID's will be 8- to 16-byte numbers that may or may not include hints to the physical location of the underlying object. The advantage of location hints in OID's is faster access; the disadvantage is that objects cannot be moved, the database may be cluttered with forwarding hints, or the OID's grow fairly large to include both a globally unique identifier plus a location hint.

Our goal was to design the assembly operator such that it can function properly with any of the possible representations. Other goals include a compact representation of objects on disk and efficient traversal of complex objects in main memory. Our current thoughts call for intermediate structures between object components in buffer, which we call *bridges*. The intention is to replace the OID in an object's root with a pointer to the bridge (sometimes called pointer swizzling), and to store a pointer to the component in the bridge. Further fields in the bridge structure include the OID (which was replaced in the root and must be restored before the object is written back to disk), the component's record identifier (which is the physical location that the file system understands), and a reference counter.

When following an OID reference to a component, we first determine whether the the component is already fixed in the buffer and pointed to by a bridge. In order to do that efficiently, we build a hash table with doubly-linked bucket chains of bridges. For this purpose, we also include two pointer fields in each bridge. If the OID could be found, we replace the OID with a pointer to the bridge and increment the bridge's reference count. Otherwise, we allocate a new bridge structure, translate the OID into a RID, i.e., a physical location, and load the appropriate cluster into the buffer. We deliberately left the translation mechanism from OID's to RID's unspecified; as pointed out above, we hope that our algorithm works with various object and OID schemes.

In order to ensure safe decisions whether an OID within an object in memory has or has not been replaced by a pointer to a bridge, there must be a distinctive characteristic between OID's and pointers. Our current

thought is that we will reserve one or two bits within OID's and require them to be different than they are in main memory pointers.

Clearly, a number of details are not worked out yet. First, we have not decided how exactly to schedule component retrieval, in particular how to optimize disk accesses using the statistical information on sharing mentioned above. Second, we have not defined yet how an object and its components are released in the buffer when an object assembly is abandoned because a condition failed. We anticipate that we need a *disassembly* operator that consumes a stream of root components of complex objects and either releases all components or extracts some components selectively, releasing all other components in the buffer. Third, we have not finalized persistent, disk-based and temporary, buffer resident object representations. Nevertheless, we believe that the above scheme can be much more efficient than naive, component-at-a-time object retrieval. While we do not have any "hard" experimental data to support this hypothesis, we have performed a preliminary, analytical study of the disk I/O's by our assembly operator.

5. An Analytical Performance Evaluation

The efficiency of the Volcano query processing software has been demonstrated in earlier reports, both for single-process and for multiprocess query evaluation [10, 17]. Since the implementation of the assembly operator is not completed yet, we give only an analytical analysis of its performance.

We restrict our analysis to the index look-up problem because it is a relevant special form of the general problem. The general problem can be modelled as multiple look-ups; we believe that the general problem gives more degrees freedom, and therefore more opportunities for optimization. Consider a stream of N references which refer to pages distributed over C adjacent cylinders. Furthermore, consider a sliding window of W references, i.e., instead of looking up one reference at a time, we always keep W references and look up the one that is most conveniently located. Notice that references are resolved in a different order than entered into the window; this does not preclude that the looked up objects be produced in the same order as their references. Buffer space for W objects on the average will be needed, but more at peaks. However, in the general case of complex objects, this will not be an issue because it does not matter in which order components are retrieved.

If each reference is looked up immediately without use of a window, each look-up requires seeking of 1/3 of the cylinders on the average plus the average disk rotational latency, which is the time for 1/2 disk rotation. If a window is used, however, organizing it as a heap as used to create sorted runs twice the size of memory [18], and the references are ordered by their physical location, $2W$ references are resolved with each sweep over all cylinders. The average seek is less than $C/(2W)$. If $2W>C$, we expect more than one reference per cylinder, and can optimize accesses within each cylinder to reduce rotational latencies. In fact, let us assume that $2W>C$, then we expect to seek C times to the next cylinder for each sweep and resolve $2W/C$ references on each cylinder. Thus, we can optimize latencies to resolve all references within one disk rotation, and reduce the average latency from 1 reference per 1/2 rotation to about $2W/C$ references per rotation.

Let us consider a concrete example. Assume there are N=1000 references to objects on pages distributed over C=10 cylinders. Since all seeks will cover a fairly short distance, let us use a constant seek time of S=2ms for our analysis. Let us assume an average disk latency of L=8.333ms. With the naive strategy, we will perform 900 seeks ($1/C$ of the references will be to the same cylinder as the last reference). Each of the references will be delayed by the average disk latency. Thus, the total cost is

$$N\left(1-\frac{1}{C}\right)S + N\,L = 1000\left(1-\frac{1}{10}\right)2ms + 1000\times 8.333ms = 10133ms.$$

If we used a window of W=25 references, we would expect $N/(2W)$ sweeps with C seeks each. The latency would be one rotation, equal to $2L$, per $2W/C$ references. Thus, the total cost when using a window of size W=25 is

$$\frac{N}{2W}\,C\,S + N\,/\,\frac{2W}{C}\,2L = \frac{N\,C}{2W}\,(S+2L) = \frac{1000\times 10}{2\times 25}\,(2ms + 2\times 8.333ms) = 3733ms.$$

When compared to the naive strategy, this is a speedup of almost 3. Notice that the speedup is linear with the window size, such that a window of W=100 would result in a speedup of about 11[2].

[2] The alert reader may have noticed that a window of W=1 does not provide a speedup of 1, as should be expected! The reason is that our analysis used the most conservative measure for how many cylinders actually participate in a sweep and for the latency within each cylinder.

6. 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 structure expressions and component iterators to selectively and intelligently assemble complex objects. A very preliminary analytical performance evaluation demonstrated that significant speedup can be achieved using a *sliding window* to schedule object retrievals from secondary storage. 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 acceptable and 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. 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).
3. 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).
4. 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).
5. 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).
6. 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).
7. R.P. Kooi, "The Optimization of Queries in Relational Databases," *Ph.D. Thesis*, Case Western Reserve University, (September 1980).
8. G. Graefe and D.J. DeWitt, "The EXODUS Optimizer Generator," *Proceedings of the ACM SIGMOD Conference*, pp. 160-171 (May 1987).
9. G. Graefe, "Rule-Based Query Optimization in Extensible Database Systems," *Ph.D. Thesis*, University of Wisconsin, (August 1987).
10. G. Graefe, "Volcano: An Extensible and Parallel Dataflow Query Processing System," *Oregon Graduate Center, Computer Science Technical Report*, (89-006)(June 1989).
11. G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System," *Oregon Graduate Center, Computer Science Technical Report*, (89-007)(June 1989).

12. R.P. Kooi and D. Frankforth, "Query Optimization in Ingres," *IEEE Database Engineering* 5(3) pp. 2-5 (September 1982).

13. C. Zaniola, "The Database Language Gem," *Proceedings of the ACM SIGMOD Conference*, pp. 207-218 (May 1983).

14. S. Tsur and C. Zaniolo, "An Implementaton of GEM - Supporting a Semantic Data Model on Relational Back-end," *Proceedings of the ACM SIGMOD Conference*, pp. 286-295 (June 1984).

15. D.S. Batory and A.P. Buchmann, *Molecular objects, abstract data types, and data models: A framework.*

16. M.Z. Hanani, "An Optimal Evaluation of Boolean Expressions in an Online Query System," *Communications of the ACM* **20**(5) pp. 344-347 (May 1977).

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. D. Knuth, *The Art of Computer Programming*, Addison-Wesley, Reading, MA. (1973).