ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Paolo Atzeni and Val Tannen (Eds)

Database Programming Languages (DBPL-5)

Proceedings of the Fifth International Workshop on Database
Programming Languages, Gubbio, Umbria, Italy, 6-8 September 1995

Paper:

Scheduling Resource Usage in Object-Oriented Queries

Theodore W. Leung

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

Scheduling Resource Usage in Object-Oriented Queries

Theodore W. Leung*

Department of Computer Science

Box 1910 Brown University

Providence, RI 02912-1910 USA

Abstract

Query facilities in object-oriented databases lag behind their relational counterparts in performance. This paper identifies important sources of that performance difference, the random I/O problem and the re-reading problem. We propose three techniques for improving the execution of object-oriented database queries: reuse/out of order execution, memoization, and buffer replacement policy. Schedule level optimization is introduced as our framework for integrating these techniques into query processing systems.

1 Introduction

Performance is an important consideration for the query component of object-oriented database systems. The success of the relational data model is due largely to the ability to provide suitable performance for query processing. The same may be true for the success of object-oriented databases. The nature of object-oriented databases leads to different performance characteristics than relational databases – assumptions and techniques from relational query optimization do not always apply. The work in this paper addresses one of these cases. We begin by presenting a characterization of a performance bottleneck that is not present in relational database systems, followed by an analysis of the failings of current optimization techniques. In section 4, we analyze this bottleneck and derive a suite of optimization techniques. Section 5 presents schedule level optimization, our framework for implementing the techniques from section 4.

2 Perils of Object-Oriented Queries

Object-oriented databases allow objects to reference other objects. In particular, this means that objects in one set can reference objects in another set. This leads to a situation where two objects may share a third object. Pages that contain shared objects are accessed many times during the course of one iteration through the referencing set.

$$Select(Emps, \lambda e. e.dept.size \geq 25)$$

Figure 1: A typical object-oriented query

Query 1 traverses the elements of the **source set** *Emps* using the **path-expression** *e.dept.size* to follow references to objects in the **target set** *Departments*, and eventually, to the the *size* attributes of those *Departments*. The source set is traversed sequentially, and each object or page of objects is accessed only once. We cannot guarantee that the objects or pages in the target set will be accessed sequentially. We also cannot guarantee that a shared object or page will be resident in the buffer pool when it is needed, leading to the possibility that such pages will be read from disk multiple times. These multiple accesses result because two or more objects in the source set share a single object in

* Author's current address: Taligent, Inc. 10355 N. DeAnza Blvd, Cupertino, CA 95014, Ted_Leung@taligent.com

the target set. The ability to capture this kind of object sharing is a principal feature of object-oriented data models. For the sample query, it is likely that more than one *Employee* works in each *Department*, which means that multiple *Employee* objects will reference each *Department* object. Object sharing leads to two performance problems. First, it causes a great deal of non-sequential I/O, since the target set's traversal order is determined by the source set's traversal order, and the two orders are unlikely to be similar. We'll refer to this as the **random I/O problem** throughout the rest of the paper. Second, the combination of object sharing and a finite buffer pool can cause the same object or page of objects to be retrieved from disk multiple times, since multiple objects in the source set refer to individual objects in the target set. We'll refer to this as the **re-reading problem** throughout the rest of the paper, and the "extra" reads of the same object will be called **re-reads**. These two problems are related, but much more effort has been focused at the random I/O problem than at the re-reading problem. Solving the re-reading problem also (partially) solves the random I/O problem, but solving the random I/O problem does not necessarily solve the re-reading problem. We explore this in more detail in section 3. The focus of our work is on eliminating re-reads, which takes the re-reading problem point of view. As a side effect, we will also reduce the number of random I/Os, thus addressing the random I/O problem.

3 Previous Approaches

Many techniques for improving the performance of object-oriented queries have been proposed. In this section we examine those techniques which are relevant to either the random I/O problem or the re-reading problem. We also consider how well each technique addresses each of the two problems.

Indices [12], and path indices [24, 22] in particular, are an established method of improving object-oriented database query performance. Path indices eliminate I/O operations, and most of these I/O operations are random I/O operations, so the number of random I/O's is also reduced. The indices are only useful for particular path expressions, and do not improve the performance of other portions of the query which access the same objects through a different path. Path indices cannot reduce the number of re-reads of a shared object, since the index manager has no memory of which objects have already been read, and even if it did, it would have no way of taking advantage of such information.

Clustering [6] and prefetching [27, 14] are also common performance enhancements in object-oriented databases. Clustering addresses the random I/O problem because it retrieves related objects in a single I/O operation, again eliminating I/O operations which happen to be random I/O's. Clustering is not guaranteed to improve the performance of other parts of the query that happen to use the objects retrieved by the cluster. Clustering is in the same position as indexing when it comes to addressing the re-reading problem.

Prefetching attempts to reduce I/O's (and therefore random I/O's) by ensuring that required objects are in the buffer pool when the query needs them. A system that prefetches may be able to prefetch objects that are being read again, but that doesn't really address the re-reading problem unless the object being read again can be retained in the buffer pool. Recent work [17] has noted that the effectiveness of prefetching decreases as the page size increases, and more importantly, as the degree of sharing (number of references to shared objects) increases.

Object assembly [20] controls the order in which unresolved object references are resolved. In general, the strategies employed by object assembly reduce the number of random I/O's performed by the query. Object assembly is unable to improve the performance of other parts of the query which reference objects that have already been assembled. The assembly operator does not explicitly attempt to address the re-reading problem. It does have limited effect on re-reads because it alters the order of object references which in turn determines the order in which pages are flushed from the buffer pool, but this improvement is due more to fortuitous referencing patterns than explicit plans to reduce re-reads.

One method of solving the random I/O problem is to sort the objects by their physical addresses. This is very effective at reducing random I/O's. It is not an effective solution to the re-reading problem because the objects may be paged out of the buffer pool before they are re-read. In that case, the objects will have to be read into the buffer pool again.

The usual mechanism for addressing the re-reading problem is to make use of the database buffer pool [15], and hope that objects which are re-read remain in the buffer pool between the reads. Sacco and Schkolnick's work on hot sets [29, 30] computes the number of pages required by a particular query as an aid to access-path selection. They also note that different kinds of access patterns benefit from different buffer management policies. Subsequent work in this

area has focused on further analysis of the relationship between access patterns and page replacement and on making the best use of the buffers available at query execution time [9, 25].

None of these methods directly address the re-reading problem, other than relying on the buffer manager to do a good job. They make no attempt to explicitly minimize the number of re-reads. Instead, responsibility for reducing re-reads is left to the buffer manager. The buffer manager is only able to do this as well as its replacement policy allows. One way to improve on the performance of queries with re-reads is to “extend” the size of the buffer pool by improving the page replacement policy. Chan et. al. [5] use hints to the buffer manager to improve replacement selection. These hints are encoded via user definable priorities. They do not describe any schemes that address the re-reading problem. The LRU-K [26] algorithm remembers the timestamps of the last k references to a page, in an effort to distinguish between frequently and infrequently referenced pages, which does better than LRU, but is still not the best for situations with lots of sharing, since the last k references to an object are not a good indicator of how many more references to that object will occur.

Cornell and Yu [13] described a method for integrating buffer management with the query optimizer. Their method focuses on determining which relations should be kept in the buffer pool, and using that information to prune the set of access plans under consideration. This doesn’t address any of the issues related to the re-reading problem, and in particular no reduction in re-reads occurs.

Chen and Roussopoulos [7] cache the results of queries. If the result of a query has been cached, then this technique addresses both the random I/O problem and the re-reading problem. Query result caching does not help the first time that the query is executed, nor does it help if the cache has been flushed.

Kemper and Kossman [21] propose a dual buffering scheme, where the buffer pool is divided into two segments, one dedicated to buffering pages, and another dedicated to buffering objects. Dual buffering allows useful objects on a page to be buffered “individually” if the rest of the page that they occupy is not useful. This eliminates wasteful use of memory in the buffer pool caused by internal fragmentation of buffer pages, and generally improves query performance.

4 Kinds of Performance Improvements

The re-reading problem has two major sources. The first cause of the re-reading problem is that multiple operators in the same query can refer to the same objects. An example of this situation is the case where some path expression is used in the predicate of more than one operator in a query. Depending on the execution order determined by the optimizer, the objects referenced by the path expression will be accessed twice, once for each operator. The query in Figure 2 is an example of this kind of query. The set *Departments* is accessed both as one of the inputs to the join, and as one of the components in the *Select*’s path expression. This case is often addressed by common sub-expression elimination techniques[11, 28, 16], but there may be additional opportunities for performance improvements when a subset of the objects described by the processing of the common subexpression are used in another part of a query. Common subexpression elimination is a source level analysis technique and has no notion of whether the objects that are produced by the common subexpression (or its intermediate values) will be needed by parts of the query which do not contain the source level common subexpression.

The second cause is that within a single operator over a set type, multiple objects in that set use some attribute to reference objects in another set. Multiple objects from the source set (the parameter to the query operator) may reference the same objects in the target set (specified by a prefix of the path expression). Query 1 is an example of this kind of query.

All of our optimizations share the notion of **common work elimination**, that is, we seek to eliminate all unnecessary read operations, even those that are undetectable by source level common subexpression elimination. We propose three classes of methods for providing performance improvements for object-oriented queries: reuse/out of order execution, memoization, and buffer replacement policy. Reuse and buffer replacement policy attempt to increase the effectiveness of the buffer pool, thereby eliminating re-reads and I/O operations. Memoization also has common work elimination as its goal, and is used in situations where reuse/out of order execution is not permissible.

4.1 Reuse / Out of Order Execution

In an ideal world, each object referenced by a query would be read into memory once, regardless of how it was referenced. After that, the object would be retained in the buffer, and any subsequent references would not cause additional I/O. This could only happen if the buffer pool is infinitely large. This idealized situation provides a valuable intuition for a new kind of optimization. Our intuition is that the first time an object O is read into the buffer pool, we want all operators that will perform a computation using O to perform their computations before O leaves the buffer pool. These operators are **reusing** the work done by the operator that actually caused O to be retrieved from disk. If we could find a way to allow these other operators to execute the slice of their execution related to O , then we can ensure that O will not be read from disk again in the future.

One method of realizing this intuition is to allow the query to execute **out of order**: During the evaluation of the query, we allow the flow of control to leave the execution of one operator and enter the execution of another operator. This happens to a limited extent in pipelined execution models[18], and what we propose is a generalization of pipelining. In a pipelined implementation, plan operators are implemented as coroutines, with control passing from coroutine to coroutine in a linear sequence corresponding to the ordering of the physical plan. For example, in $Join(Select(A, f1), B, f2)$, each time a tuple of A is processed, control begins with the coroutine for the *Select*, and is transferred to the coroutine for the *Join*. This is a restricted form of out of order execution. Each object or relational tuple starts at the coroutine for the innermost plan operator, and passes through all the coroutines for the plan operators enclosing that inner most operator before the next object or tuple is processed. The order of execution is “out of order” compared to an implementation where each plan operator is implemented as a procedure operating on entire sets. The order of execution is still in an order that is specified by the query, however the iteration takes place at a smaller granularity.

Our notion of out of order execution is a generalization of this idea. Pipelined implementations restrict the transfer of control to be between an operator whose output is connected to the input(s) of another operator. We generalize this by removing this restriction, allowing transfer of control between plan operators whose outputs and inputs are not directly connected. As long as the output type of one operator matches the input type of another operator, transfer of control may occur, subject to constraints regarding set overlapping and coverage. As an example, consider the query in figure 2.

$$Join(Dept, Select(Emps, \lambda e. e.dept.size \geq 25), \lambda d \lambda e. d.mgr.sal \geq e.debt)$$

Figure 2: A query amenable to reuse

We assume that the Join is evaluated via a nested loops algorithm, and the selection via sequential scan. We assume the file containing *Departments* is structured so that the *Manager* of a *Department* is clustered together with the *Department*. Furthermore, we assume that at least one *Employee* works in every *Department*. A typical physical plan for this query appears in figure 3. In this example, the collection *Department* is traversed twice, once by the *LoopJoin*

$$LoopJoin(DeptMgrCluster, LoopSelect(Emps, \lambda e. e.dept.size \geq 25), \lambda d \lambda e. d.mgr.sal \geq e.debt)$$

Figure 3: Physical plan for query 2

(since it appears as one of the join inputs), and once by the *LoopSelect* (via the path expression $e.dept.size$). If the plan is executed by executing the selection before beginning to process the join, the selection will cause all of the *Departments* to be read into memory (this is guaranteed because every *Department* has at least on *Employee* in it). If the selection is sufficiently large, those *Department* objects which were read least recently will have been flushed from the buffer pool by those *Departments* referenced more recently. Those “early” *Departments* must be retrieved from disk again to process the join.

Figure 4: Control flow for Query 2

Under out of order execution, the *Select*'s selection condition is processed when an object is read from *Emps*. Before proceeding to the next *Employee*, execution of the plan switches to the *LoopJoin* operator, which evaluates that portion of the join which can be evaluated given the *Department* object that was retrieved by the path expression in the selection. After this portion has been evaluated, execution of the selection resumes. This flow of control is diagrammed in figure 4. This strategy results in a reduction in the number of I/O operations, since the *Department* objects for a particular *Department* and *Employee* pairing in the join are only read once. Unfortunately, *Department* objects are still read more than once overall, since each *Employee* in the selection must be compared to each *Department*. We can improve this by recognizing that objects are retrieved in units of pages; when we retrieve an *Employee*, we “join” it with all the *Department* objects on all the *Department* pages in the buffer pool. A small amount of in memory bookkeeping is required to ensure the correctness of the result.

This technique is only applicable when we can guarantee that the set of objects to be traversed by out of order execution is the same as the set of objects that would be traversed by a normal order execution. Constraints on the containment relationships between sets, along with information from the schema manager of the database allow us to infer the necessary relationships at query compile time.

4.2 Memoization

Out of order execution allows us to reuse the intermediate results of computations by altering the flow of control during the execution of the query. Unfortunately, it is not applicable in all situations, because it is not always possible to determine which objects are actually being reused. Function memoization is a common technique for improving the performance of functional programs, and indexing is a special case of memoization. We can employ a form of memoization to improve the performance of those queries which cannot be improved via out of order execution. The

$$\text{Union}(\text{select}(\text{Emps}, \lambda e. \text{isPrime}(e.\text{mgr}.\text{dept}.\text{size})), \\ \text{select}(\text{Emps}, \lambda e1. e1.\text{mgr}.\text{dept}.\text{size} < 10 \text{ AND } e1.\text{wife}.\text{salary} > \$60k))$$

Figure 5: A query amenable to memoization

common subexpression *e.mgr.dept* in figure 5 seems like an ideal candidate for reuse. Assuming that the left *Select* argument to the *Union* operator is evaluated “first”, we can take the value of *e.mgr.dept* that is computed by the left arm of the *Union*, and then evaluate the right arm (*Select*(..., *e1.mgr.dept.size*)) out of order using the value of *e.mgr.dept* from the left arm. Unfortunately, this does not work, since the *Select* in the right arm also needs the value of *e1.wife.salary*, which cannot be guaranteed to be in the buffer pool at the point when we wish to evaluate

e1.mgr.dept.size. If a reusable expression is conjoined with an expression which will cause disk I/O we cannot use out of order execution, since we cannot guarantee that the I/O operation will not flush needed objects from the buffer pool.

However, we are able to use memoization to prevent the path expression *e.mgr.dept* from being read twice. When the left arm of the *Union* operator is processed, the implementation of the left *Select* operator writes a memo file for *e.mgr.dept* (even though it is evaluating *isPrime(e.mgr.dept.size)*). The implementation of the right *Select* operator reads from the memo file for *e.mgr.dept*, instead of *Emps*. This eliminates the intermediate traversal of the *Managers* during the evaluation of the path expression. In this situation, memoization involves building a path index incrementally. The memoization can be improved if the left *Select* only writes entries whose *Employees* satisfy the condition *e1.mgr.dept.size* ≤ 10 (from the right *Select*) into the memo file. The memo file then contains precisely those objects which satisfy the left conjunct in the right *Select*'s predicate.

4.3 Buffer Replacement Policy

Both reuse/out of order execution and memoization address the re-reading problem when different parts of the same query access objects multiple times. They are not effective for the case where re-reads occur in a single query operator. Changing the buffer manager's page replacement policy can be used to address the case where re-reads arise in a single operator as a result of object sharing. If a shared object can be retained in the buffer pool until it is referenced again, then the work that was done to read the object from disk is reused by subsequent accesses to that object, as long as the shared object remains in the buffer pool. This has a decidedly flavor from reuse/out of order execution and memoization. Yet it is consistent with our aim of reusing common work, since the "initial work" of retrieving an object from disk is reused by subsequent references to the object. All buffer management algorithms have this property. Our contribution is to provide a policy that is tailored to path expressions, where object sharing is commonplace.

Recall that the source of the difficulty is that multiple objects in a source collection reference a single object in a target collection. In the case where a single level of referencing is involved, we can use reference counts from the source objects to the target objects as part of the page replacement metric. For multiple levels of referencing, we simply treat each single level case in the multi-level path expression. The replacement policy computes the average reference count for a page of objects. The values of the reference counts partition the set of pages into **generations**, much like the generations that occur in generational garbage collectors [31]. Representatives of multiple generations are present in the buffer pool at any point in time. The replacement policy replaces pages on a priority basis, assigning the lowest priority to the generation with the smallest reference count. Within each generation, pages are replaced using an LRU policy. As a special case, the generation for reference count = 1 can be restricted to a single page, since we know that the only reference to that page has already occurred. This provides FIFO behavior for scan like queries. As an extension we cause the reference counts for pages in a generation to decay as the objects within it are referenced. This prevents thrashing in the lower generations and gives a more accurate estimate of the remaining references to the page.

5 Schedule Level Optimization

We address the re-reading problem and the random I/O problem by providing a framework for the three kinds of optimizations discussed in section 4. This framework introduces a new level to the optimization process, the schedule level, which takes place after both logical plan rewriting and physical plan generation. At the schedule level, each physical plan operation is expanded into a sequence of schedule level operators. Schedule level operators form an assembly language for query I/O. The operators include instructions for reading an object from a file, comparing objects, extracting object fields, etc. The implementation of physical plan operators as macros over schedule level operations allows the scheduler to have explicit control over disk I/O operations and intermediate results. It also allows multiple custom implementations of operators to exist simultaneously and provides ability to jump into and out of "the middle" of physical plan operators. We can also reorder schedule operators in order to improve the performance of the query. This notion is reminiscent of optimizations that are used in compilers, such as function inlining, peephole optimization, or instruction scheduling. The schedule level optimization process follows these steps:

1. The physical plan is converted into an intermediate representation called a schedule graph. The schedule graph emphasizes physical “partitions” of logical collections and enables transformations on those partitions. The initial conversion is accomplished by using templates which map physical plan operators onto schedule graphs.
2. The graph representation is modified using rules that allow nodes in the graph to be deleted, combined, and replaced. The rules embody transformations for reuse and memoization, and use meta-data and inclusion-relations between partitions to detect opportunities for applying the optimizations.
3. The resulting graph is used to statically allocate buffer pages to the various partitions, in an attempt to make best use of the buffer pool. Thus, each partition is assigned a buffer, and each buffer can be controlled by a different page replacement policy. This is a generalization of work relating access patterns and page replacement policies. We use our reference-count based page replacement policy to manage the buffers for partitions that participate in path expressions.
4. The graph is input to a code generation algorithm which generates an executable sequential program.

5.1 Compile-Time Buffer Allocation

Each schedule operator is allocated a private buffer pool, which may be shared with other schedule level operators. This differs from traditional database systems where all physical plan operators share a single buffer pool which holds objects of many types. Segregation of types allows us to tightly control the behavior of objects with respect to the buffer pool. The possible disadvantage of this technique is that it may fail to be responsive to global properties of the query. The allocation of buffers to the various types is determined at compile time, using a heuristic that uses the fanout of pages referenced to determine the buffer allocations.

5.2 Buffer Page Replacement Policy

When static buffer allocation occurs, the scheduler can query the database schema manager to get information about the degree of object sharing via references from a particular collection. If the reference counts for a partition exceed a threshold value, the schedule uses the RefBuffer policy to manage that partition.

5.3 Algorithmic Code Generation

The code generation algorithm takes the schedule graph, along with the buffer assignments and generates a sequential program that can be executed to evaluate the query. The program is a sequence of instructions in “an I/O assembly language”. The basic operations of this language include reading an object (page of objects) from a data structure (disk file, b-tree index, hash-table, etc.), applying a function to an object (page of objects), comparing fields of an object with some other value (including other object fields), and propagating objects based on some boolean condition.

The high level structure of the code generator is analogous to that of a compiler. We define a notion of basic blocks over the schedule graph, and use dependencies among these blocks to induce a linear ordering on them. Using this linear order, we can then generate an instruction sequence for each node in the schedule graph. At the appropriate points in the instruction sequence the algorithm inserts code to handle reuse.

When the code generator has been run on 3, the output appears as in figure 6. The basic schedule operators function like this: the *Read* operator reads an object from a file, the *Apply* operator applies an attribute to an object (possibly causing a disk read), the *ApplyBuiltIn* operator provides a mechanism for operating on basic types, the *Filter* operator produces its first argument as output if its boolean (second) input is true, and the *Output* operator sends an object to the result file. In addition, there are also less familiar operations in the schedule. The *BinaryTuple* operator produces a tuple containing its two arguments. The *BufferApply* operator applies an attribute to every object of a particular type that is currently in the buffer. In figure 6, the *BufferApply*(buffer(d),mgr) means that the mgr attribute will be applied to every *Department* object in the buffer that holds d. Likewise the *CrossApply* operator generates the cross product of its first argument with every object in the buffer for its second argument. In addition, it makes a log of every object in that buffer which participated in the cross product. This log is then made available for use by the *LogRead* operator.


```

BeginLoop
  e = Read(Emp)
  d = Apply(e,.dept)
  s = Apply(d,.size)
  b1 = ApplyBUILTIn(>=,s,25)
  e1 = Filter(e,b1)
  db = Apply(e1,.debt)
  m = BufferApply(buffer(d),.mgr)
  sa = Apply(m,.sal)
  b2 = ApplyBUILTIn(>,sa,db)
  j = CrossApply(e1,buffer(d),log1)
  j1 = Filter(j,b2)
  o = Output(j1)
  BeginLoop
    l = LogRead(Dept,log1)
    m1 = Apply(l,.mgr)
    sa2 = Apply(m1,.sal)
    b3 = ApplyBUILTIn(>,sa2,db)
    j2 = BinaryTuple(e1,l)
    j3 = Filter(j2,b3)
    o1 = Output(j3)
  EndLoop
EndLoop

```

Figure 6: Schedule for query 2

The schedule obtained by generating code for the graph without applying any schedule-level optimizations consists of a typical nested loops implementation for the join. The query in figure 6 improves on that implementation by taking advantage of its knowledge of the way that buffers are being used. This can substantially reduce the number of pages that need to be re-scanned for a nested loop join.

6 Status and Conclusions

This work has been motivated by work in the compiler literature on improving the performance of loop intensive programs [2]. Particularly relevant is the body of work on loop optimizations for data locality [1, 33, 32, 23, 4] and work on compiler directed cache management [8].

A major difference between our work and the work in compilers is that we are unable to assume array semantics for our data, since we operate on sets. While much of the work in compiler optimization focuses on arrays and makes decisions based on the values of array indexes, our work addresses the problems particular to databases, where decision making is based on the data values themselves. The notion of stopping the execution of a conceptual unit (plan operator) has no analog in compilers, the closest being loop distribution or strip mining.

A simulation of the page replacement policy has been implemented, and preliminary experiments show a reduction in page faults between 10 and 40% for queries with a high degree of sharing. A prototype implementation of the schedule level optimizer and runtime system is almost operational, and we hope to begin reporting experimental results shortly.

We have described the re-reading problem and random I/O problem, which arise in object-oriented databases because of object sharing. The sources of the two problems were explained, and we presented three classes of optimizations that can be used to ameliorate them. The notion of schedule-level optimization was introduced as a framework

that supports these optimizations, and an overview of the techniques used in our schedule-level optimizer was presented. The possibility of schedule-level optimization opens a new space of options for improving query runtime performance.

7 Acknowledgements

I wish to thank Stanley B. Zdonik for many discussions regarding this work. The EREQ group at Brown has provided a fruitful atmosphere for thinking about issues in query performance.

References

- [1] Walid Abu-Sufah, David J. Kuck, and Duncan H. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Transactions on Computers*, C-30(5):341–356, May 1981.
- [2] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. Technical Report UCB/CSD-93-781, Computer Science Division (EECS), University of California, Berkeley, 1993.
- [3] Peter Buneman and Sushil Jajodia, editors. *Proceedings of the SIGMOD International Conference on Management of Data*, Washington D. C., May 1993. ACM Special Interest Group on Management of Data, ACM Press.
- [4] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, San Jose, California, April 1994. IEEE Computer Society Association for Computing Machinery, IEEE Computer Society Press. Also appears as SIGPLAN Notices 29(11), November 1994, Computer Architecture News Vol. 22 Special Issue, October 1994, Operating Systems Review 28(5), December 1994.
- [5] Chee Yong Chan, Beng Chin Ooi, and Hongjun Lu. Extensible buffer management of indexes. In Li-Yan Yuan, editor, *Proceedings of the 18th International Conference on Very Large Databases*, pages 444–454, Vancouver, Canada, August 1992. Morgan Kaufmann Publishers, Inc.
- [6] Ellis E. Chang and Randy H. Katz. Expliting inheritance and structure semantics for effective clustering and buffering in an object-oriented DBMS. In James Clifford, Bruce Lindsay, and David Maier, editors, *Proceedings of the SIGMOD International Conference on Management of Data*, pages 348–357, Portland, Oregon, June 1989. ACM Special Interest Group on Management of Data, ACM Press.
- [7] ChungMin Melvin Chen and Nicholas Roussopoulos. The implementation and performance evaluation of the adms query optimizer: Integrating query result caching and matching. In Jarke et al. [19], pages 323–336.
- [8] Chi-Hung Chi and Hank Dietz. Unified management of registers and cache using liveness and cache bypass. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 344–355, Portland, Oregon, June 1989. ACM Special Interest Group on Programming Languages, ACM Press. Also appears as SIGPLAN Notices Vol. 24 No. 7, July 1989.
- [9] Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems. In A. Pirotte and Y. Vassiliou, editors, *Proceedings of the Eleventh International Conference on Very Large Databases*, pages 127–141, Stockholm, August 1985. VLDB, Morgan Kaufmann Publishers, Inc.
- [10] James Clifford and Roger King, editors. *Proceedings of the SIGMOD International Conference on Management of Data*, Denver, Colorado, May 1991. ACM Special Interest Group on Management of Data, ACM Press.

- [11] Sophie Cluet and Claude Delobel. A general framework for the optimization of object-oriented queries. In Michael Stonebraker, editor, *Proceedings of the SIGMOD International Conference on Management of Data*, pages 383–392, San Diego, California, June 1992. ACM Special Interest Group on Management of Data, ACM Press.
- [12] Douglas Comer. The ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.
- [13] Douglas W. Cornell and Philip S. Yu. Integration of buffer management and query optimization in relational database environment. In Peter M. G. Apers and Gio Wiederhold, editors, *Proceedings of the Fifteenth International Conference on Very Large Databases*, pages 247–255, Amsterdam, The Netherlands, 1989. Morgan Kaufmann Publishers, Inc.
- [14] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical prefetching via data compression. In Buneman and Jajodia [3], pages 257–266.
- [15] Wolfgang Effelsberg and Theo Haerder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4):560–595, December 1984.
- [16] Georges Gardarin and Rosana S. G. Lanzelotte. Optimizing object-oriented database queries using cost-controlled rewriting. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Proceedings of the Third International Conference on Extending Database Technology*, number 580 in Lecture Notes in Computer Science, pages 534–549, Vienna, Austria, March 1992. Springer-Verlag.
- [17] Carsten A. Gerlhof and Alfons Kemper. A multi-threaded architecture for prefetching in object bases. In Jarke et al. [19], pages 351–364.
- [18] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, pages 73–170, June 1993.
- [19] Matthias Jarke, Janis Bubenko, and Keith Jeffery, editors. *Proceedings of the Fourth International Conference on Extending Database Technology*, number 779 in Lecture Notes in Computer Science, Cambridge, United Kingdom, March 1994. Springer-Verlag.
- [20] Tim Keller, Goetz Graefe, and David Maier. Efficient assembly of complex objects. In Clifford and King [10], pages 148–157.
- [21] Alfons Kemper and Donald Kossmann. Dual-Buffering Strategies in Object Bases. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 427–438, Santiago, Chile, 1994.
- [22] Alfons Kemper and Guido Moerkotte. Access support in object bases. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the SIGMOD International Conference on Management of Data*, pages 364–376, Atlantic City, New Jersey, May 1990. ACM Special Interest Group on Management of Data, ACM Press.
- [23] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the Sixth International Workshop on Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes in Computer Science, pages 301–320, Portland, Oregon, August 1993. Springer-Verlag.
- [24] David Maier and Jacob Stein. Indexing in an object-oriented database. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, pages 171–182, Pacific Grove, California, September 1986. Association for Computing Machinery.
- [25] Raymond Ng, Christos Faloutsos, and Timos Sellis. Flexible buffer allocation based on marginal gains. In Clifford and King [10], pages 387–396.
- [26] Elizabeth J. O’Neil, Patrick E. O’Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In Buneman and Jajodia [3], pages 297–306.

- [27] Mark L. Palmer and Stanley B. Zdonik. Fido: A cache that learns to fetch. Technical Report CS-91-15, Department of Computer Science, Brown University, Providence, Rhode Island 02912-1910, February 1991.
- [28] Arnon Rosenthal and Paul Helman. Understanding and extending transformation based optimizers. *IEEE Quarterly Bulletin of IEEE technical committee on Database Engineering*, 9(4):44–51, December 1986.
- [29] Giovanni Maria Sacco and Mario Schkolnick. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *Proceedings of the Eighth International Conference on Very Large Databases*, pages 257–262, Mexico City, Mexico, September 1982. VLDB, Morgan Kaufmann Publishers, Inc.
- [30] Giovanni Maria Sacco and Mario Schkolnick. Buffer management in relational database systems. *ACM Transactions on Database Systems*, 11(4):473–498, December 1986.
- [31] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, pages 1–42, St. Malo, France, September 1992.
- [32] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Ontario, Canada, June 1991. ACM Special Interest Group on Programming Languages, ACM Press. Also appears as SIGPLAN Notices Vol 26, No 6, June 1991.
- [33] Michael Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM Conference on Supercomputing*, pages 655–664, 1989.