Aggregate and Grouping Functions in Object-Oriented Databases

Kjetil Nørvåg and Kjell Bratbergsengen
Norwegian University of Science and Technology
Department of Computer Science
N-7034 Trondheim, Norway
{noervaag,kjellb}@idt.ntnu.no

Abstract

Efficient evaluation of aggregate functions in object-oriented databases (OODB) can have considerable impact on performance in many application areas, like geographic information systems and statistical and scientific databases. The problem with current systems is inefficient execution of aggregate functions with large data volumes, and lack of flexibility: it is not possible to extend the systems with new aggregate and grouping functions. In this paper, we extend the concept of aggregate functions from relational databases. We introduce the concept of grouping functions, which could enhance flexibility and performance considerably. We show how this could be implemented into an OODB. We also describe how support for special kinds of aggregate queries and data structures can help in designing future high-performance systems.

Keywords and phrases: aggregate functions, object-oriented databases, database programming languages, query processing

1 Introduction

Object-oriented database systems (OODB) are attractive alternatives to traditional relational database systems. This is especially true for applications where the modeling power of relational databases is insufficient, and the language mismatch makes integration between the application programs and the database system difficult.

OODB have been an immature technology, and have only recently been put into "real work". But: in some application areas relational databases still beat them in performance. Common for many of these application areas are heavy use of *aggregate functions*, together with *grouping*. A typical aggregation query partitions a collection of objects, and evaluates one or more functions on the objects of a group. Typical examples are sum or average of an attribute in each object in a group. The result is a new collection of objects, one for each group. Although in most database systems these are not the most commonly executed operations, queries with aggregate functions possibly scan large parts of the database, making them very time consuming.

Traditionally, business applications have have been the most heavy users of aggregate functions. In the last years, new application areas have emerged. Areas of particular interest are geographic information systems, data mining, data warehousing, and statistical and scientific databases[1]. These databases contain highly structured data, very suitable for the object-oriented data model. Operations on (and analysis of) data in these systems make heavy use of mathematical and statistical operators, and in some applications, complex grouping. To be able to do efficient queries, these operators should be a part of the query system.

As of today, most OODB provide support for execution of basic aggregate functions, but often with low performance when applied to large data volumes. Also, the systems lack the desired flexibility: it should be possible to extend the systems with new aggregate and grouping functions.

[1] It should be noted that the examples described here are not necessarily independent, if you have a data warehouse, it is very likely you want to do analysis on it, by the use of data mining techniques.

In this paper, we extend the concept of aggregate functions from relational databases, and we introduce the concept of *grouping functions*. The use of grouping functions can enhance flexibility and performance considerably. We show how this could be implemented into an OODB. Finally, we describe how support for special kinds of aggregate queries and data structures can help in designing future high-performance systems.

2 Aggregate Functions in OODB Research

Much research has been done in aggregate function evaluation in relational databases, but evaluation of aggregate functions in OODB is still an immature area of research. The main reasons are probably:

1. *The importance of efficient evaluation of aggregate functions in OODB has not been recognized.* Application areas suitable for OODB, like those discussed in the previous section, should justify the importance.

2. *The advantages of storing and processing data in databases instead of files has not been recognized in all applications areas where it is appropriate [5].* The focus in scientific computing has been on doing computations with files as inputs. Especially with complex file formats as, e.g., the HDF file format[2], computations and maintenance is not trivial. It is also worth noting that important projects, as e.g. EOSDIS[3], are drifting from files to database storage of data.

3. *It is often thought that the algorithms developed for relational databases are good enough for OODB as well, if the aggregation is done on primitive attributes.* If, on the other hand, the aggregation is done on methods, the problem is thought to be equivalent in complexity to a general read-only query. One should keep in mind that the evaluation of aggregate functions form a restricted subclass of read-only queries, which gives room for considerable improvement.

Although the effort put into research on aggregate functions in OODB has been low, one should keep in mind that much has been published about topics related to aggregation. The most important work is, of course, research done on aggregate function evaluation in relational databases. The fundamentals are the same, and many results valid for relational databases applies to OODB as well. Valuable sources for aggregate function evaluation are Bitton et.al. [1], Bratbergsengen [3] and Shatdal et.al. [16]. Only recently has the grouping functions been the focus for query optimization [23, 9, 10].

3 Aggregate and Grouping Functions

Aggregation is basically partitioning a set of objects into groups, and evaluate one or several aggregate functions over the objects in the groups. The result is one set of values (from the aggregate functions) for each group.

Aggregate Functions In our notation, we will write aggregate functions as $\mathtt{A}(\mathtt{P}^A)$. For each function $\mathtt{A}$, a set of *parameters* is given from the set $\mathtt{P}^A$, which are one or more attributes from the objects that the aggregate function is applied to.

All system with query languages already support some aggregate functions, like sum and average, but with new application areas, with very different needs, it is important to have the possibility

[2]HDF is a file format for storing and transmitting scientific data sets, and a library interface for working with the data [5].

[3]Earth Observing System Data Information System

of adding new functions. Examples are statistical functions in statistical and scientific databases, spatial functions as area and perimeter in GIS/spatial databases, and special time-related functions in temporal databases.

It is useful to think of aggregate functions as state transition functions, which in e.g. POSTGRES [24] is done explicit by the use of the CREATE AGGREGATE construction. The difference between an aggregate function and an ordinary function is that an *initial state* has to exist. With the concept of objects available, we can define an aggregate function as an object class, here called *aggregate function objects*. A new aggregate function object, with its defined initial state, is created for each group during execution. An aggregate function class should contain:

1. A *constructor* that defines an initial state.
2. An *iterator* method to be called for each object aggregated to the group.
3. A *method returning the final result* of the aggregation for the group.
4. With two of the actual parallel aggregation algorithms, more than one aggregate object can exist for each group (we will come back to this in section 6.4). As a part of these algorithms, it must be possible to merge these preliminary group objects into one. To be able to do this, with general aggregate functions, it is necessary to have *merge methods* defined, as well.

Grouping Functions Which group an object belongs to, is determined by the result of one or more grouping function `G(`$\mathtt{P}^G$`)`. Each function `G` has parameters from the parameter set $\mathtt{P}^G$. $\mathtt{P}^G$ consists of one or more attributes from the objects which aggregation is applied to, together with (optional) parameters to the grouping function. The latter ones are independent of the values of the objects, they are constant values during the aggregation. Grouping functions have no states, they are just mapping functions from a possible multi-dimensional space with continuous-valued attributes to a space with discrete-valued attributes, returning a group identifying result.

In current database systems, there is really no such thing as grouping functions. Which group an object should belong to, is determined by a concatenation of the value of one or more attributes. In the context of grouping functions, we can say that this is the identity grouping function, `I(p) = p`, where output equals input. In the future, it should be possible to define grouping functions in the same way as aggregate functions.

In applications where several combinations of attributes should map into the same group, grouping functions are necessary to avoid a costly preprocessing of data. Example applications are spatial and temporal databases. If we want coordinates within an area to belong to the same group, we can have a grouping function that do this mapping. Another example is temporal grouping, where all data in a fixed interval of time should map to the same group. The described grouping functions can also be extended to grouping where one object can participate in more than one group. This can be useful for several application areas, e.g. some business data warehousing applications.

One important aspect of grouping functions that returns a group identifier, is that this kind of grouping lend itself easy to hash-based algorithms. This is important to get high performance.

4 Query Processing Operator

It is useful to explain (and implement) aggregation and other operations as language-independent *operators*. This is also the way it is done in systems as Gamma [6] and Volcano [8]. A stream of objects flows through a tree of operators. An "object" can be either a materialized object, or the object identifier. In OODBs it should be is possible to operate on methods, in the same way as on attributes. When we in the rest of the paper talk about attributes, these could be either values, or method invocations.

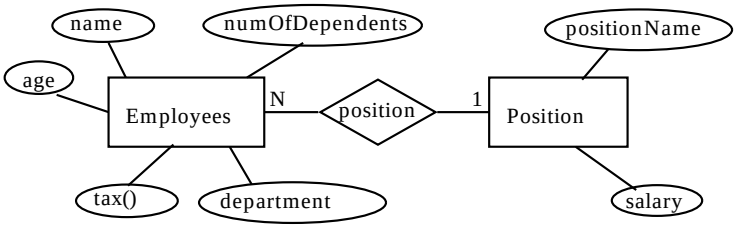

Figure 1: Example database

The aggregate operator[4] can be written as:

$$\texttt{AGGREGATE(I:A}_1\texttt{(P}_1^A\texttt{)},\ldots,\texttt{A}_n\texttt{(P}_n^A\texttt{):G}_1\texttt{(P}_1^G\texttt{)},\ldots,\texttt{G}_m\texttt{(P}_m^G\texttt{):R)}$$

where:

- `I` is the input stream. This can be a stream from another operator, e.g. `SELECT`.
- `A` defines a set of aggregate functions. For each function, a set of *parameters* is given from the set $\texttt{P}_i^A$.
- `G` is the set of grouping functions. The grouping functions are applied to all objects in the input stream, and the set of results determines for each object which group it belongs to: $\mathrm{G} = \cup_{i=0}^{m} G_i$. The concatenation of partial results from the grouping functions should probably be mapped to a group identifier. This makes hash-based aggregation algorithms easier to employ.
- `R` is the output stream. The aggregate functions `A` are applied on every set of objects belonging to the same group, thus giving one result object for each group. Each object contains `G` and the result of `A` applied on the set. Thus, the schema is $\mathrm{R}_s(\mathrm{G}, \mathrm{A}_1, \ldots, \mathrm{A}_n)$.

5 Aggregation in Query and Application Languages

Relational databases usually have support for aggregate functions in their query language (SQL), although efficiency differ. Most object-oriented databases have (or will soon have) support for aggregate functions with grouping in their query language, but lacks support in their application languages, requiring the user to explicitly call the query language from the application language, giving an impedance mismatch. In the ODMG-93 Object Query Language [4] the operations are supported by a group-by-expression similar to SQL.

To illustrate notation and languages, we will use the employee database in Figure 1 as an example. This database has two classes: The class `Employee`, with attributes `name`, `department`, `age` and `numOfDependents`, and the class `Position` with attributes `positionName` and `salary`. In addition, we have a relationship between the lasses as depicted by the figure. The method named `tax()` in the `Employee` class, returns the tax as a function of `numOfDependents` and the salary in the `Position` object. We call the extent (collection of all `Employee` objects) `Employees`. Suppose we wanted a list of the average age of people in the different departments in our example database. With the notation from Section 4, and the result put into `Result`, this can be written:

```
AGGREGATE(Employees:avg(age):I(department):Result)
```

[4]This operator is based on the aggregate operator described by Bratbergsengen in [3].

Query Language In the ODMG-93 Object Query language (Release 1.2), this can be done with the following query:

```
select department, avg_age: avg(select age from partition)
from Employees e
group by dep: e.department
```

Application Language It is important to have the aggregate operator as a part of the application language to avoid the impedance mismatch. As of today, the user is usually required to write the code himself or get the result through a call to the query processor. In that case, the query is formulated as a string interpreted at run-time. This is the case of the current version of the ODMG C++ binding [4]. If we embed the previous query into a C++-program, with the result of the query is returned in `resultSet`, we could write this as:

```
Set<Ref<DepAvgAge>> resultSet;
oql(resultSet, "select department, avg_age: avg(select age from partition)\
                from Employees e\
                group by dep: e.department");
```

Use of Methods in Queries In an OODB, it is also possible to do queries on methods, where methods should be possible to use just the same way as attributes. If we wanted a a list of the average amount of tax paid by the people in the different departments, it should be possible to do this as:

```
select department, avg_tax: avg(select tax() from partition)
from Employees e
group by dep: e.department
```

6 Execution of Aggregation Queries

In the relational data model, all tuples streaming to an aggregate operator contain atomic attributes, which can be processed immediately, and without accessing other tuples. This is not the case in the object-oriented data model. If the attributes are just values, this corresponds to retrieval of tuple attributes, but evaluation of methods is more difficult. The methods might involve computations, and possibly access to other objects. To make the aggregation part "cleaner", we separate the evaluation of methods and the aggregation by introducing a *materialize* operator. This operator outputs a restricted materialization of the objects: the required methods, and no other, are evaluated, and the results are output to the aggregate operator. In this way, the aggregate operator can apply the aggregation stream without computations and without accessing other objects. The result is that we can use a aggregate operator similar to the relational one.

6.1 The Materialize Operator

The straightforward strategy is to naively fetch an object, evaluate the methods, output the result, and continue with the next object in the stream. As long as the methods does not access other objects, this strategy works well. But if methods accesses other objects (which is similar to a functional join), this might be inefficient: if an object is accessed from more than one object in the aggregate stream, it might have been thrown out from the buffer/cache between the accesses. The object has to be re-read from disk, which is inefficient. Therefore, we want to control these accesses. This can be done by employing available techniques for functional join. Several algorithms and techniques are described by Shekita et.al. in [19, 18, 17], and Lieuwen et. al. in [14, 13].

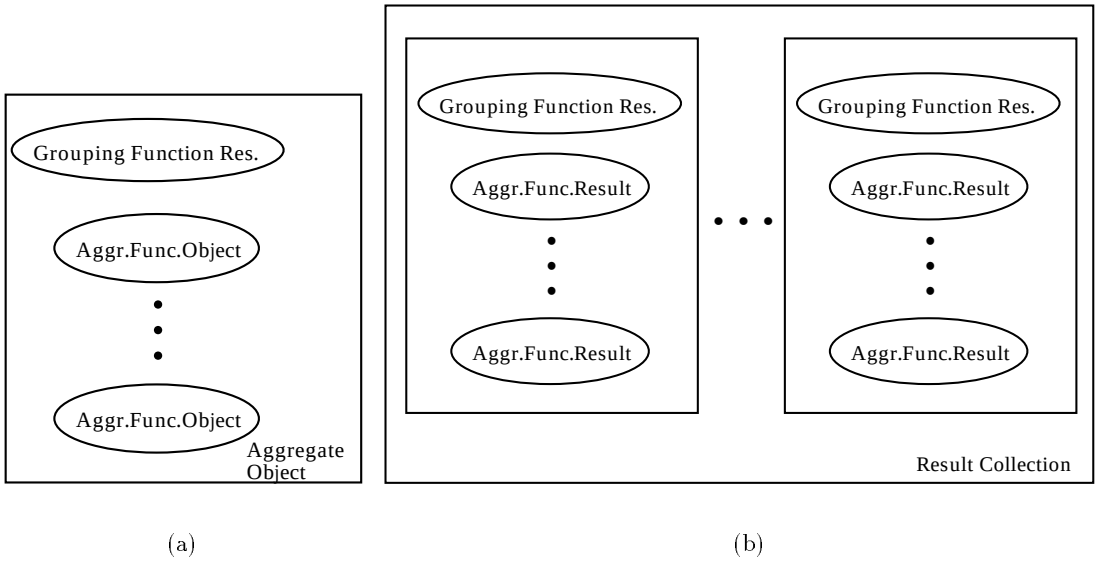

Figure 2: Aggregate object and objects in result collection

6.2 The Structure of Aggregate and Result Objects

During aggregation, a new *aggregate object* is created for each group (see Figure 2a). The aggregate objects will belong to a class with the following attributes:

1. An attribute identifying the group (the result of the grouping functions applied on an object).
2. One attribute for each A_i in the list of aggregate functions. These attributes are object instances of aggregate function classes.

The result of the aggregation is a collection with one element for each group (see Figure 2b). These objects contains the final results of the aggregate functions. The result objects will belong to a class with the following attributes:

1. An attribute identifying the group .
2. One attribute for each A_i in the list of aggregate functions. These are the final results of the aggregate functions.

6.3 The Aggregate Operator

The function of the aggregate operator is to apply the aggregate function(s) on the objects it receives, and group according to the grouping function(s). In the relational data model, all tuples streaming to an aggregate operator contain atomic attributes. With the use of the materialize operator, aggregation in OODB can be done with the traditional algorithms used in relational databases.

The *nested-loop aggregation algorithm* is the basic approach, but becomes inefficient when the size of the aggregate objects (number of aggregate objects$*$size of each aggregate object) is much larger than primary memory. In that case, one should either

- *pre-sort* the objects in the object stream based on the grouping identifier, or
- *partition* the objects in the input stream based on the grouping identifier

before aggregation. If the input stream is not sorted initially, which is common in an OODB[5], it is better to use a partition-aggregate algorithm. With un-sorted input, partition-aggregate will always perform better than sort-aggregate [2].

6.4 Parallel Aggregation

In general, it is difficult to parallelize navigational queries in OODB [11]. Set processing, on the other hand, lends itself to parallelizing. With parallel machines, distribution of work between nodes can improve performance. Several algorithms for distributed aggregation exist, with the following in common use:

- Parallel aggregation with local aggregation/central coordinator
- Parallel aggregation by redistribution
- Redistribution with local aggregation

With parallel aggregation in OODB, we get a problem that does not exist in relational databases: how to evaluate methods in objects accessed from objects to be aggregated. There are several ways to evaluate these methods, we have here the query vs. data shipping problem. This has much in common with distributed (functional) join, but the the use of complex methods complicates the situation.

Local Aggregation/Central Coordinator This algorithm is only useful for scalar aggregates, e.g., with no grouping. All nodes do local aggregation on the part of the data allocated to that node. The local aggregation can be done with one of the three algorithms described above. The resulting aggregate objects are sent to a coordinator node, which uses the preliminary aggregate objects it receives to create the final aggregate objects. With user defined aggregate functions, it is necessary to have available methods to merge the partial preliminary objects (see section 4).

Redistribution In the first phase, objects are relocated to nodes according to a hashing function applied on the result of the grouping methods. Objects belonging to a group is, in that way, guaranteed to end up at the same node. In the second phase, aggregation is performed at each node with one of the three first algorithms.

Redistribution with Local Aggregation In the parallel aggregation by redistribution algorithm, more data than necessary is moved. With large groups (many objects in each group), we can gain much by doing local aggregation before redistribution [3]. This is the way aggregation with grouping is done in e.g. Gamma [6]. In the first phase, all nodes do local aggregation on data residing on the node, just as in the local aggregation/central coordinator algorithm. In the second phase, the result is distributed according to a hashing function applied on the value of the result of the grouping methods. The global result is obtained by merging the preliminary objects (cf. section 4) as done in parallel aggregation by redistribution.

Several implementations have shown that with high-bandwidth communication between nodes, the local aggregation is not necessarily beneficial. Whether to do local aggregation or not depends on data selectivity and communication bandwidth. As suggested by [16], sampling of data should be used to decide if local aggregation should be done before redistribution.

[5]Sort-aggregate is the algorithm used in most existing commercial relational systems [7]. One reason for this, is that quite often, traditional applications want a sorted result from the aggregation (ORDER BY in SQL). In a typical OODB application, this will probably *not* be the case.

7 High-Performance Aggregation

The performance of evaluation of aggregate functions can be improved in several ways. We have already treated parallel aggregation, in addition it is useful to study optimization for special kinds of aggregate queries, special access patterns, and special data structures:

Multi-Level Aggregation Sometimes, aggregation is nested, *multi-level aggregation*. That is, aggregation done on a group of aggregate groups. This can be exploited to increase performance.

Aggregation on Multi-Dimensional Data Structures Although current databases only support aggregation on collections as sets or bags, the aggregation can be done on other multi-dimensional structures. This is useful for spatial and temporal databases.

In OODB, it is possible for the user to specify aggregate functions. While this works well on set-like structures, it is likely to give bad performance on other structures. The best alternative might be to provide new aggregate operators that work on non-set-like structures.

Temporal Aggregates Conventional aggregate algorithms are not efficient when applied to temporal databases. Algorithms for computing temporal aggregates are presented by Kline and Snodgrass in [12]. As far as we know, no one has published work on parallel algorithms for temporal aggregation, or algorithms for temporal aggregation in OODBs. Temporal databases can be viewed as a subclass of multidimensional databases, and algorithms similar to aggregation on spatial data structures can be employed.

Combining Bulk Loading and Aggregation Bulk loading is *loading a large external dataset during a single sitting* [15]. If it is known at load time what kind of aggregate functions that will be employed later on the data set, it is possible to do aggregation while loading. Combining bulk loading algorithms with aggregation will probably be beneficial, especially in databases where summary data can be exploited (e.g. in statistical and scientific databases). As a starting point, the bulk loading algorithms developed by J. Wiener [22] can be used. But, no good algorithms for parallel loading has yet been developed. This problem has to be solved, as a single-threaded algorithm will be a potential performance bottleneck.

Precomputed Results In many of the proposed application areas for aggregate evaluation, much of the data will be static. It is possible to take advantage of this by either having precomputed the query for part of the database, and/or use stored results from earlier queries. These techniques are quite similar to techniques used for view maintenance/materialized views in data warehousing [21, 9, 10].

Another approach is to have the system maintain an index to the data with statistical summary data.[6] By using a structure which makes it easy to exploit the summary data (a tree is used in the tree based access method proposed by Srivastava and Lum [20]), performance can be greatly enhanced if the data are heavily used.

Resumable Aggregation In large databases, queries involving aggregation can be a very time consuming. It is desirable that a crash during aggregation does not mean that all the work is wasted. Several approaches to avoid this can be used. Examples are aggregation checkpointing and storing partial results. The partial results is similar to materialized views, but for resumable aggregation they are completely machine generated.

[6] Statistical summary data is partial results needed for some operations. An example is a sum of some elements.

8 Conclusions

Efficient evaluation of aggregate functions in object-oriented databases (OODB) can have considerable impact on performance in many application areas. Still, current systems are not able to compete in performance with traditional relation database systems. In this paper, we have justified the need to concentrate on research in this area, and also showed why so little previous research exists.

In this paper we have extended the concept of aggregate functions from relational databases, and introduced the concept of *grouping functions*. Grouping functions are not currently offered by any system, but in this paper we have shown how this could be integrated into the aggregation process. By offering aggregation and grouping as depicted in this paper, the systems will be able to provide the desired flexibility needed in future high-performance database applications. We believe, that the way to high performance aggregation in future database systems lies in:

- More efficient algorithms. Aggregation on methods has much in common with functional join, and often functional join is part of the process too. It will be beneficial to use resources on this, and in particular on parallel and distributed functional join.
- Maintaining and employing precomputed results. When applicable, this is probably the strategy that can give most in increased performance. The implementation cost need not be too high. These precomputed results has to be automatically created when necessary by the system.
- Developing new algorithms for applications with multi-dimensional data structures. These algorithms will also be useful for evaluating temporal aggregates.
- More research in grouping functions. Grouping functions, with mapping to a discrete domain, lend itself easy to hash-based aggregation algorithms.

References

[1] D. Bitton, H. Boral, D. J. DeWitt, and W. K. Wilkinson. Parallel Algorithms for the Execution of Relational Database Operations. *ACM Transactions on Database Systems*, 8(3), 1983.

[2] K. Bratbergsengen. Hashing Methods and Relational Algebra Operations. In *Proceedings of the 10th International Conference on VLDB*, 1984.

[3] K. Bratbergsengen. Relational Algebra Operations. In *PRISMA Project Workshop, Nordwijk, The Netherlands*, 1990.

[4] R. Cattell, editor. *The Object Database Standard: ODMG-93. Release 1.2*. Morgan Kaufmann, 1996.

[5] D. J. DeWitt. DBMS - Roadkill on the Information Superhighway. Invited talk at VLDB'95, 1995.

[6] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990.

[7] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2), 1993.

[8] G. Graefe. Volcano — An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1), 1994.

[9] A. Gupta, V. Harinarayan, and D. Quass. Generalized Projections: A Powerful Approach To Aggregation. Technical report, Stanford, 1994.

[10] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-Query Processing in Data Warehousing Environments. In *Proceedings of the 21st International Conference on Very Large Data Bases*, 1995.

[11] K.-C. Kim. Parallelism in Object-Oriented Query Processing. In *IEEE Sixth International Conference on Data Engineering*, 1990.

[12] N. Kline and R. T. Snodgrass. Computing Temporal Aggregates. In *Proceedings of IEEE 11th Int'l Conference on Data Engineering, March, 1995*, 1995.

[13] D. Lieuwen, D. DeWitt, and M. Mehta. Pointer-based Join Techniques for Object-Oriented Databases. Technical Report CS-TR-92-1099, University of Wisconsin-Madison, 1992.

[14] D. Lieuwen, D. DeWitt, and M. Mehta. Parallel Pointer-based Join Techniques for Object-Oriented Databases. In *Proc. 2nd International Conference on Parallel and Distributed Information Systems*, 1993.

[15] D. Maier and D. M. Hansen. Bambi Meets Godzilla: Object Databases for Scientific Computing. In *Seventh International Working Conference on Scientific and Statistical Database Management*. IEEE Computer Society Press, 1994.

[16] A. Shatdal and J. F. Naughton. Adaptive Parallel Aggregation Algorithms. In *Proceedings of the 1995 ACM SIGMOD*, pages 104–114. ACM Press, 1995.

[17] E. Shekita. *High-Performance Implementation Techniques for Next-Generation Database Systems*. PhD thesis, University of Wisconsin-Madison, 1991.

[18] E. J. Shekita and M. J. Carey. Performance Enhancement Through Replication in an Object-Oriented DBMS. In *Proceedings of the 1989 ACM SIGMOD*, 1989.

[19] E. J. Shekita and M. J. Carey. A Performance Evaluation of Pointer-Based Joins. Technical Report 916, University of Wisconsin-Madison, 1990.

[20] J. Srivastava and V. Y. Lum. A Tree Based Access Method (TBSAM) for Fast Processing of Aggregate Queries. In *IEEE Fourth International Conference on Data Engineering*, 1988.

[21] J. Widom. Research Problems in Data Warehousing. In *Proceedings of the 4th Int'l Conference on Information and Knowledge Management (CIKM), November 1995*, 1995.

[22] J. L. Wiener. *Algorithms for Loading Object Databases*. PhD thesis, University of Wisconsin-Madison, 1995.

[23] W. Yan and P.-Å. Larson. Eager Aggregation and Lazy Aggregation. In *Proceedings of the 21st International Conference on Very Large Data Bases*, 1995.

[24] A. Yu and J. Chen. *The POSTGRES95 User Manual*, 1995.