Query Optimization using Collective I/O, OID Reordering and Migration in Parallel Object-Relational Databases

Sachin More and Alok Choudhary

Technical Report No. CPDC-TR-9802-014
©1998 Center for Parallel and Distributed Computing
January 1998

Center for Parallel and Distributed Computing
Department of Electrical & Computer Engineering
Northwestern University
Technological Institute
2145 Sheridan Road, Evanston IL-60208

Query Optimization using Collective I/O, OID Reordering and Migration in Parallel Object-Relational Databases *

Sachin More†
ssmore@ece.nwu.edu

Alok Choudhary
choudhar@ece.nwu.edu

Center for Parallel and Distributed Computing
Department of Electrical and Computer Engineering
Northwestern University
2145 Sheridan Road, Evanston IL 60208-3118 USA

Abstract

Object-relational databases extend the capabilities of the relational databases by allowing definition of new data types and methods to operate on these data types while retaining most of the relational model semantics. These extensions together with ever increasing size of the data that the database systems are expected to handle pose new challenges in query processing. Parallel object-relational database systems address these issues by storing and manipulating the data across multiple processors. In this paper, we propose a new technique to speed up method execution on large objects. It modifies I/O access pattern of the query to achieve better I/O performance. This is achieved by selectively migrating and reordering the method execution tasks. This technique part of a larger I/O optimization scheme based on collective I/O and provides as much as 600% improvement in query execution performance.

1 Introduction

Object-relational database management systems (ORDBMS) extend the relational model by facilitating definition of new data types and methods to manipulate them. The capability to define new data types is very critical for extending database technology to newer application domains like Geographical Information Systems (GIS) and multimedia databases. ORDBMS are expected to store and manipulate a wide variety of data that includes, conventional data items like numbers, strings and dates but also more exotic data items like text documents and images. The size of the data items belonging to the latter type can be anywhere from a few hundred bytes (a paragraph of text) to a few megabytes (a raster image). This leads to an overall increase in the expected amount of data stored by the database system and the computational resources needed to manipulate the data. Parallel object-relational database systems address these issues by storing and manipulating the data across multiple processors.

*This work was supported by NSF Young Investigator Award CCR-9357840, NSF CCR-9509143.

The sheer size of the data that a single query is expected to handle makes the query processing an I/O bound task. We are currently investigating various issues involved in storage and retrieval of large objects (instances of the data types whose size is more than a few kilobytes, for example raster images). In this paper, we propose a new technique called OID reordering and migration, to speed up query execution. This technique is aimed to ensure I/O parallelism. It modifies the access patterns of the query to achieve the goal.

1.1 Related Work

The SEQUOIA 2000 research project [SD92] is one of the early efforts that looks at the DBMS issues in managing Earth Science data. One of the major contributions of this work is the Sequoia 2000 Storage Benchmark [SFGM92] which is aimed at capturing the basic requirements for collection, storage and querying of Earth Science data. More recent work in this area can be found in the Paradise Project [PYK+97]. The Paradise object-relational database system is a parallel geo-spatial database system. It uses techniques developed as part of the Gamma Database Machine Project [DGS+90] and adds new techniques for parallel processing of geo-spatial database queries. While the Paradise Project is mainly aimed at parallelization of geo-spatial data processing techniques, our work is aimed at efficient storage and retrieval of ORDBMS data.

Storing data across multiple disks to achieve higher I/O bandwidth was proposed in [SGM86, PGK88] and others. Since then lot of work has been done in the area of parallel I/O and various solutions have been proposed. A lot of this work has been in the area of scientific computing where applications need to access large arrays stored on disks [dRBC93, TC96]. We extend the concept of collective I/O and other related techniques like request merging and data sieving for I/O in the database domain to achieve high performance in retrieval of large objects and their use in efficient query processing. In our recent work ([MC98]), we have demonstrated effectiveness of these techniques in I/O performance improvement. In this paper, we propose and evaluate the OID reordering and method execution migration technique. The experimental results presented in Section 4 show as much as 600% improvement in I/O performance.

The rest of this paper is organized as follows: Section 2 discusses I/O issues in parallel object-relational database systems. Section 3 gives a detailed description of OID reordering and method execution migration. Section 4 presents the experimental results. Conclusions are provided in Section 5.

2 Motivation

2.1 Why I/O is important?

The advances in database technology has paved way for its use to many more application domains. Many of these new application domains require the database system to manage and process large amounts of data. This has resulted in demand for more computational and storage resources from the database systems. The problem is more acute for storage resources than compute resources. [DG92]

remarks : "*... disk throughput was predicted to double while processor speeds were predicted to increase by much larger factors. Consequently, critics predicted that multiprocessor systems would soon be I/O limited unless a solution to the I/O bottleneck was found.*". According to the current predictions ([PYK+97]), GIS databases are expected to handle several terabytes of data in near future and the demand is will be ever increasing. The *Sequoia 2000 Benchmark* [SFGM92] puts the storage requirement of a regional data set at a little over 1GB, the national data set at 18GB and the world data set at several terabytes. Most of this storage requirement is needed to store *large objects*. For example, large objects occupy 17GB of the total requirement of 18GB for the national data set. The significant increase in the size of the datasets present new challenges.

2.2 The System Architecture

Parallel database system architectures can be classified in to three basic categories [LOT94], *Shared Everything (SE)*, *Shared Nothing (SN)* and *Shared Disk (SD)*. [Sto86] compares the above three architectures based on different issues involved in developing a parallel database system. The SN architecture scores over the rest mainly due to excellent scalability and provides near-linear speedups and scale-ups [DGS+90, Gro88]. Hence, SN architecture is a natural choice for implementation of a high performance parallel database system. Our system model consists of a set of processors, $\mathcal{P}$, each having a local disk to store the data. Each node runs a database server thread and is called a *compute node*. Each table is distributed among the compute nodes using horizontal table partitioning (Figure 1). A subset of the compute nodes is used to store the data belonging to large objects and hence they also act as *I/O nodes*.

2.3 Storage Space Organization

Given hardware storage resources (secondary storage), data layout strategy determines the organization of the data on the storage medium. For parallel relational database systems it is determined by the way the tables are declustered among processors. The local disk on a processor stores all the rows owned by the processor. However this simple strategy in many cases is not well suited for parallel object-relational database systems. To illustrate the issue here, consider a table that contains a large object in one of its columns. The rest of the attributes in a row occupy at most a few kilobytes. This means that the size of each row is now a few megabytes. The first objection to such a scheme is that a query that does not refer to attribute containing the large data type incurs significant I/O penalty. Also, rows from a table are cached by the database system to improve performance. Typical cache size is a few megabytes. The success of caching scheme will be severely compromised since the cache can accommodate only a few rows from the table. Note that these problems will be present in both sequential and parallel object-relational database systems. A problem specific to multiprocessor environment is related to inter-processor communication. In a non-trivial query plan, a rows is expected to move between the processors (*join* and *sort* operators usually need to move rows). If the size of each row is in megabytes it will result in overloading the inter-connection network. Hence large objects can

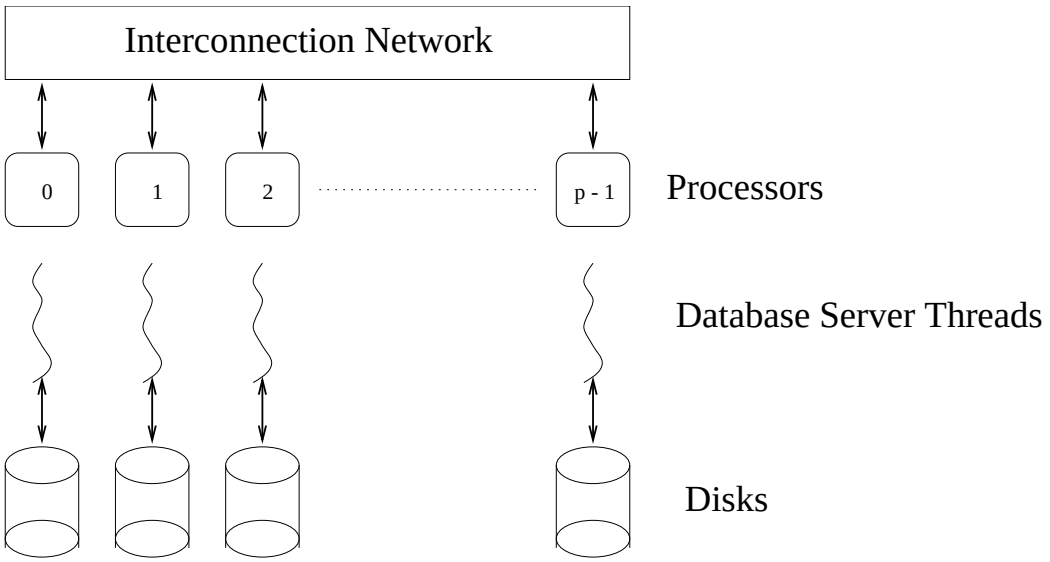


Figure 1: Computation and I/O Model

not be stored inside a row without performance problems.

Typically large objects are not stored inside their owner row [YwtPG95]. Instead an *Object Identifier (OID)*, which is a pointer to the object, is stored in the row. This gives rise to two different storage spaces in the database. A *table space* that stores the tables and an *object space* that holds the large objects. The storage strategy for the table space can be the same as described earlier where the local disk of the owner processor stores the row. Various table fragmentation policies can be used to partition the table space across multiple processors. The issues involved in the table space fragmentation have been dealt in some detail in the parallel relational database literature [JK84, SKPO88, DGS+90]. Object space on the other hand is a feature of a object-relational database which has no counterpart in the relational database model. The size of the object space at least a magnitude higher than the corresponding table space.

We use *pooled striping* [MC98] to organize and manage the object space. In contrast to *Individual Striping* where each large object is striped independently of others, pooled striping, concatenates large objects belonging to the same data type to form a *logical pool* which is then striped across the set of local disks attached to processors (see Figure 2.6). Pooled striping was chosen over individual striping for two reasons. Each large object by itself provides very little I/O parallelism benefits since it occupies a few megabytes. Also, ability to apply methods to large objects in parallel is important to achieve compute parallelism. If the large objects are striped across disks independent of each other, it is not possible to optimize the I/O access pattern that is generated when multiple large objects need to be fetched in parallel.

An important issue here is I/O load balancing. The object space is distributed over a number of

disks, each holding a fraction of data. Each processor is responsible for fetching the part of the data stored on its local disk. A skew in amount of data *accessed* by each node will mean a significant degradation of performance. I/O load imbalance contributes to performance loss both directly and indirectly. Any processor that has to do more I/O than others spends extra cycles in I/O. This effectively slows down other processors that need data stored on the heavily-loaded node. Other implication is that each processor has a limited amount of main memory available. The I/O optimization techniques discussed in Section 2.4 perform better if more main memory is available *per processor* that is involved in I/O operations. Hence, I/O imbalance will reduce the effectiveness of the I/O optimization techniques. I/O load balancing is very much dependent on the data layout strategy employed and the access pattern of the queries. A bad data layout strategy can affect the performance significantly. There is no *a priori* layout guaranteed to work for all types of datasets and queries on those datasets. The access pattern of the query also plays an important part in I/O load balance. Section 2.6 discusses this issue in more detail and in Section 3, we introduce a new technique that transforms access pattern of a query for better I/O performance.

2.4 The I/O Optimization Problem

2.4.1 Various Levels of I/O Optimization

I/O optimization problem for object space can be handled at three levels (Figure 2). The techniques used at each level vary and are directly related to the type of information available at that level.

1. *Query Executor* : This is the highest level at which I/O optimization can be done. Information available at this level is *logical*. Consider a modified version of query 2 (The *clip* method from the original query is replaced by *average* method) from the Sequoia 2000 Benchmark [SFGM92] :

   ```
   retrieve (average(RASTER.data), RASTER.time)
   where    RASTER.band = BAND
   order by ascending time
   ```

   The `RASTER` table contains a series of raster images in the `data` column. Each raster image has a timestamp which is stored in the `time` column and an associated frequency band which is stored in the `band` column. The query execution proceeds as follows. After selecting rows from the *RASTER* table that satisfy the *where* clause and then projecting the *time* and *data* attributes, the query executor has to apply the method *average* to the *data* attribute. Given the storage indirection mechanism (see Section 2.3), the query executor has a collection of *OID*s which are used to fetch the large objects. Here, the query executor knows how many objects to fetch and which processors needs to receive the results. Using this information, the query executor has to come up with a schedule that determines the order in which the large objects are retrieved, where the *average* method is executed for each object (which determines the compute load balance) and how the results reach their respective destinations.

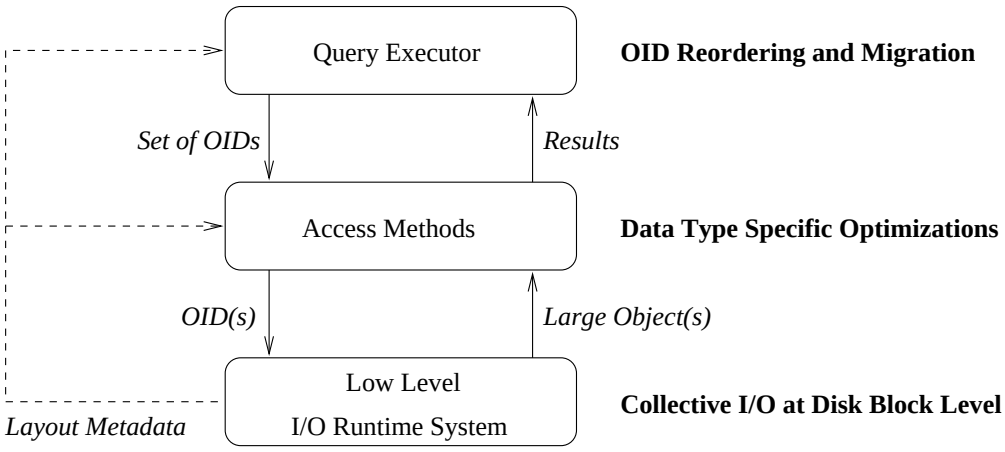


Figure 2: The figure illustrates various levels at which I/O optimizations can be done. The solid lines indicate the flow of data between these layers. The broken lines indicate the flow of metadata information that is used to compute the I/O schedule. The boldface text on the far right lists various optimization techniques employed by each layer.

2. *Methods for Large Objects* : They know the internal structure of the large objects. For example, a *clip* method which selects a section from an array, is aware of the fact that it is accessing a multi-dimensional array of a given size. Based on this information, it can carry out I/O optimization techniques that are specific to array objects. Currently implemented array methods use *data sieving* technique [TCB+96].

3. *Low Level I/O Runtime System* : It has information about the physical location of data. This information can be used to pre-process the requests (which includes transformations applied to original set of requests), schedule the low level I/O requests and finally communicate the data to the appropriate processors. This layer currently uses collective I/O at disk block level for I/O optimization.

2.4.2 Collective I/O

This is a generic technique used to optimize I/O requests in multi-processor environments. The basic idea behind the technique is to co-ordinate I/O accesses from different processors to arrive at an efficient I/O schedule. Note that an I/O schedule may require a processor to access data on behalf of some other processor which results in communication when executing the I/O schedule. Collective I/O was first proposed in [dRBC93] in the form of a *two-phase I/O strategy*. It allows processors to access arrays stored on secondary storage. In the first phase, the processors read parts of an array in a *conformal* distribution. In the second phase, the data is redistributed among the processors to

achieve the final array distribution. [TC96] describes an *Extended Two-Phase I/O Strategy* that allows processors to read arbitrary portions of the array. It first does a complete exchange of I/O requests between the processors and computes an efficient I/O schedule. Next it executes the I/O schedule followed by a communication phase to redistribute the data. Both, Two-Phase and Extended Two-Phase I/O strategies are implemented at user program level. [NK96] and [Kot95] describe collective I/O implementation at file system level and at disk level respectively.

We implemented collective I/O technique both at query executor level and at low level I/O runtime system. In the query executor, the processors carry out a complete exchange of the OIDs of the objects that need to be read. *This allows for OID reordering and method execution migration discussed in Section 3.* In the low level I/O runtime system, collective I/O is used to coordinate accesses to the physical storage. The algorithm given in Figure 3 sketches the major steps.

2.4.3 Request Merging and Data Sieving

This technique is aimed at reducing the number of I/O requests. The average time to find and transfer an arbitrary disk block is [EN94] :

$s + rd + btt$

where $s = seek\ time$ $rd = rotational\ delay$ $btt = block\ transfer\ time$

To transfer k *consecutive blocks, the estimated time is* : $s + rd + (k * btt)$

Hence, it is more profitable to give a single I/O request for consecutive blocks than to give k separate requests. The request merging technique merges two or more consecutive requests to reduce the number of I/O requests (Figure 4). Request merging is used by the low level I/O runtime system. Data sieving technique extends request merging by coalescing two non-contiguous requests. It reads *more* data than necessary in order to reduce the number of requests and throws away the unnecessary data (Figure 5).

2.5 Retrieval Strategies for Large Objects

Under collective I/O, various retrieval strategies for large objects can be employed. The huge size of data that needs to be accessed, prohibits loading all of them simultaneously. Hence, the data needs to be loaded in multiple phases and results from each stage need to be saved before loading the data for the next stage.

- *LoadN strategy*: $\mathcal{N}$ large objects belonging to a *single* processor are retrieved in a single stage (Figure 6). Since the data for the objects is striped across multiple disks, I/O parallelism can be exploited. This strategy offers no compute parallelism because the requesting processor has to process the objects in a sequential fashion.

- *LoadN4All Strategy* : $\mathcal{N}$ objects for *all* processors are loaded in a single stage (Figure 7). This strategy gives both I/O and compute parallelism. The success of LoadN4All strategy depends on

```
input : B, the set of OIDs for the objects to be retrieved

Using B, compute the set of pages P, that contains the data belonging to the objects in B

exchange P between the processors

compute the set of pages LP, that reside on the local disk of the processor and needed by some other processor(s)

compute the set of pages CP, that reside on the local disk of the processor and are needed on the same processor

compute the set of pages RP, needed by this processor but residing on some other processor

post asynchronous receive requests for the pages in RP

compute LCP = LP U CP

Apply optimization techniques (for example, request merging) to LCP to compute OLCP

for each request R in OLCP
        execute R
        for each page p in R
            if (p is in LP)
              send p to the processors that need p
            endif
            if (p is in CP)
              copy page p into the appropriate local buffer
            endif
        endfor
endfor

wait for the asynchronous receive requests for pages in RP to finish
```

Figure 3: This algorithms describes the major steps in our implementation of collective I/O at low level I/O runtime system

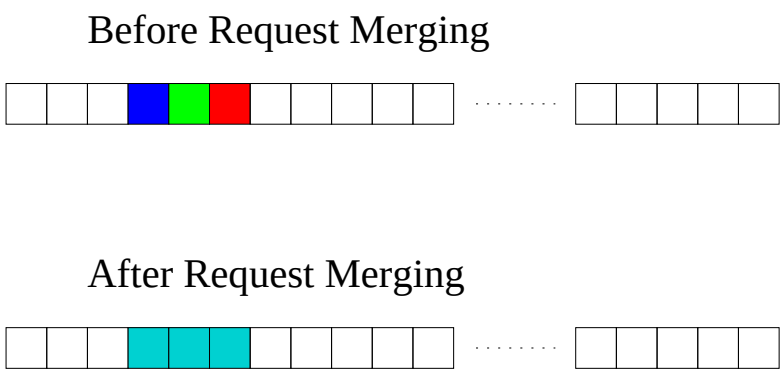


Figure 4: Request Merging : The figure shows three different I/O requests (purple, green and red) which are merged together in one request (shown in blue).

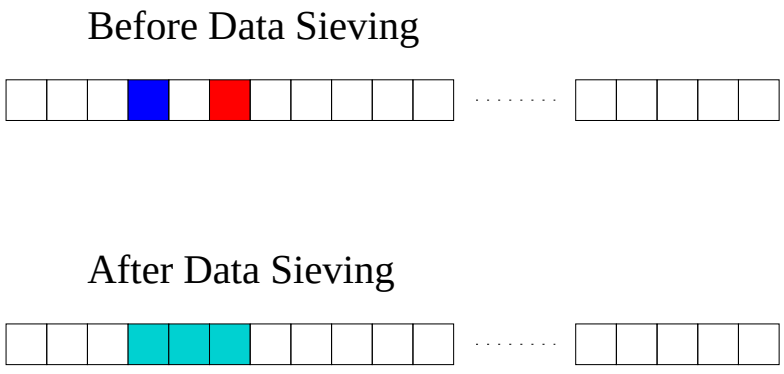


Figure 5: Data Sieving : Two I/O requests (in purple and red) which are physically apart (the white portion between them separates them) are transformed into a single I/O request (shown in blue). Note that the portion of the data belonging to the space between the original request is discarded after being read.

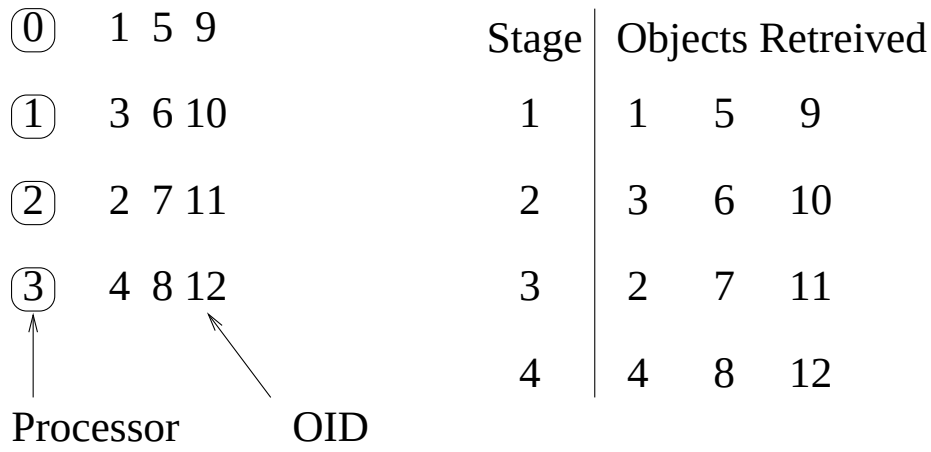


Figure 6: LoadN Strategy (N=3) : The left part of the figure lists the OIDs of the objects that need to be retrieved on each processor. On the right, is one of the possible I/O schedules under the LoadN strategy. In the first phase, OIDs 1, 5 and 9 are retrieved which belong to processor 0. The next phase retrieves objects for processor 1 and so on. Note that there could be multiple phases that retrieve objects for the same processor. For example, a LoadN strategy with N=2 will need 8 stages, two for each processor, to retrieve the complete set of objects.

distribution of number of large objects that need to be retrieved among the processors. We use *LoadOne4All* strategy in which one object per node is retrieved in each stage.

2.6 I/O Optimization and Access Patterns

Both I/O and compute parallelism are important for efficient query execution. The compute parallelism is achieved by using various table fragmentation policies like range partitioning and hashed partitioning. These policies also provide I/O parallelism for the table space. I/O parallelism for the object space on the other hand is dictated more by the *access pattern* of the query. In other words, I/O performance depends very much on the order in which the large objects are accessed. Given a storage strategy, there is always a *good* access pattern (which is better than the others) for the same set of large objects.

To see how access pattern affects performance, consider the storage layout shown in Figure 8. Each large object is declustered among two disks. Object 0 is stored on disk 0 and 1, object 1 is stored on disk 2 and 3 and so on. Now consider a query that needs to read all objects and apply a method on each one of them. Lets assume that the execution schedule consist of two phases. In each phase one object is read per processor to gain 8-way parallelism in method execution.

Now consider the access pattern shown in Figure 9. During the first phase, only 4 disks (0, 1 ... 3) are active. During the second phase only disks 4, 5, 6 and 7 are active. Hence I/O parallelism is not exploited fully. Also there is a communication bottleneck on the processors whose local disks are being

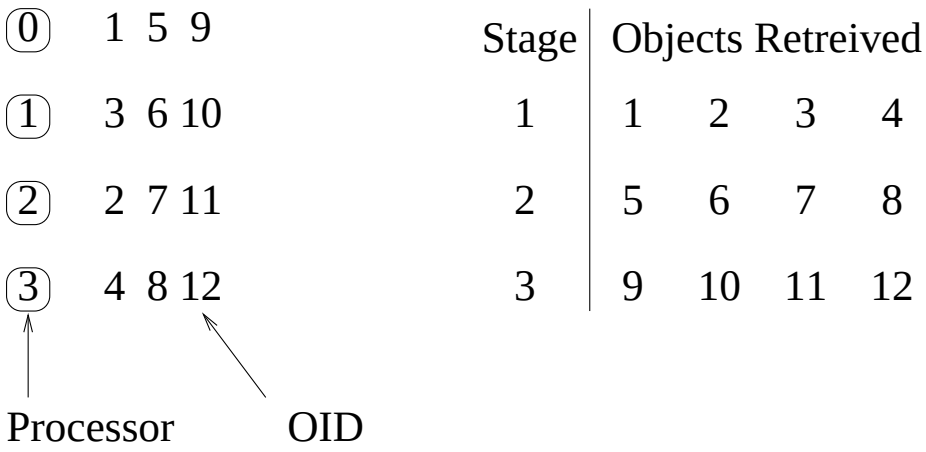


Figure 7: LoadN4All Strategy (N=1) : Left part of the figure shows the list of OIDs of the objects to be retrieved on each processor. One of the possible schedules under LoadN4All strategy is shown on the right. In each stage, N objects for each processor are fetched. In the schedule shown, four objects are fetched in each stage, one per processor. Note that the order in which the objects are fetched need not match the order in the list shown on the left. For example, stages 2 and 3 could be interchanged or object 4 could be fetched in stage 3 and object 12 could be fetched in stage 1. As we will show in the later part of the paper, such reordering may be necessary to improve I/O performance.

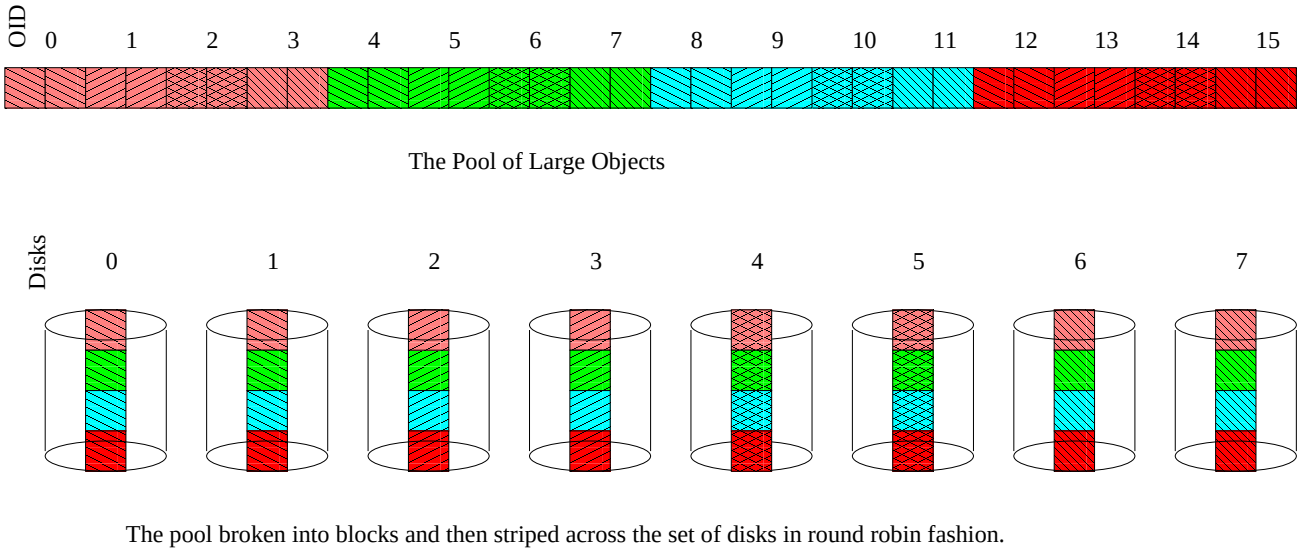


Figure 8: The figure illustrates pooled striping and the resultant storage layout. The stripe block size is equal to half the size of the objects. The pool of the objects is cut into blocks and the blocks are then distributed in a round robin fashion among the disks. Object i resides on disks $i\%4$ and $i\%4+1$.

accessed. Note that there is very good compute load balance for the method execution. Furthermore, the total amount of data read from each disk is also the same.

Now consider the access pattern given in Figure 10. It can be derived from the previous access pattern by :

- Exchanging some objects between the two stages (Reordering).
- Some objects are fetched on a different processor (Migration).

Note that the new access pattern retains other features of the earlier access pattern, namely, set of objects to be read, number of stages, the number of objects read in each stage and number of objects fetched per processor in each stage. The reason that the new access pattern will perform better is because all eight disks are active in each phase, giving maximum I/O parallelism.

3 Technique for OID Reordering and Method Execution Migration

As we saw in Section 2.6, an access pattern generated by a query play an important role in the overall performance of the query. We propose a new technique which can detect a bad access pattern and modify the execution schedule that generates a better access pattern for any object space layout. This is achieved by changing the order in which the objects are fetched and fetching an object on a different processor.

3.1 Why are bad access patterns generated?

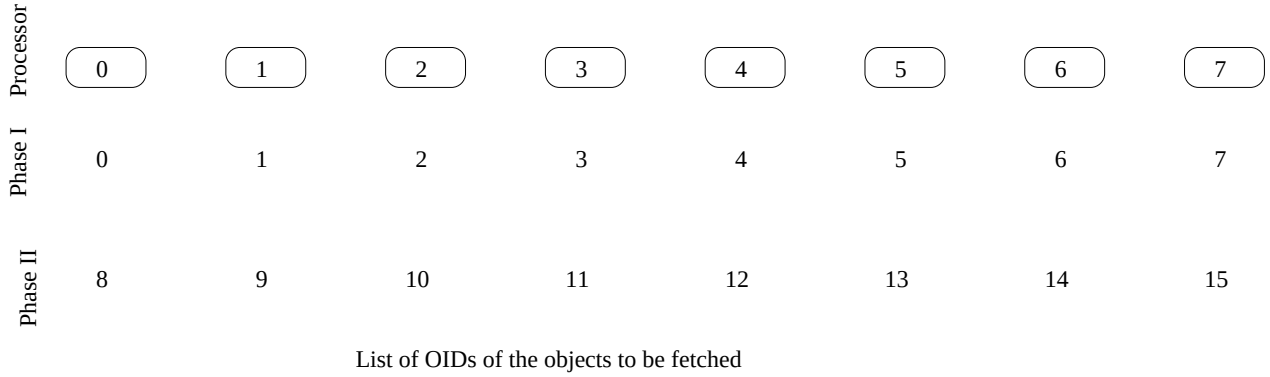
The access patterns are function of the query that generates them. Whether the access pattern is *good* or *bad* depends on the object space organization. Consider a simplified version[1] of query 10 in [PYK+97] (which is based on the Sequoia 2000 Storage Benchmark).

```
select raster.date, raster.channel
from raster
where raster.data.average() > CONSTANT;
```

This query selects the rows from `raster` table where the result of the application of the `average` method on the `data` attribute is greater than `CONSTANT`. The `data` attribute contains a raster image which is several megabytes in size (the actual size depends on the particular dataset used from the benchmark). The execution of this query is very straightforward. The `raster` table is partitioned among a set of processors. Each row of the table contains an OID that points to the actual raster image. In each phase of the execution, the next set of OIDs from each partition are selected and the corresponding large objects are read into the memory of the processor on which the row that

[1] The `clip` method in the `select` and `where` clause was removed.

Figure 9: A Bad Access Pattern for Storage Strategy in figure 8 : In phase one, objects 0, 1, 4, 5, 8, 9, 12 and 13 are read. In the second phase, objects 2, 3, 6, 7, 10, 11, 14 and 15 are read. Note that during the first phase objects from only disks 0 through 3 are being read and in the second phase objects from only disks 4 through 7 are being read.



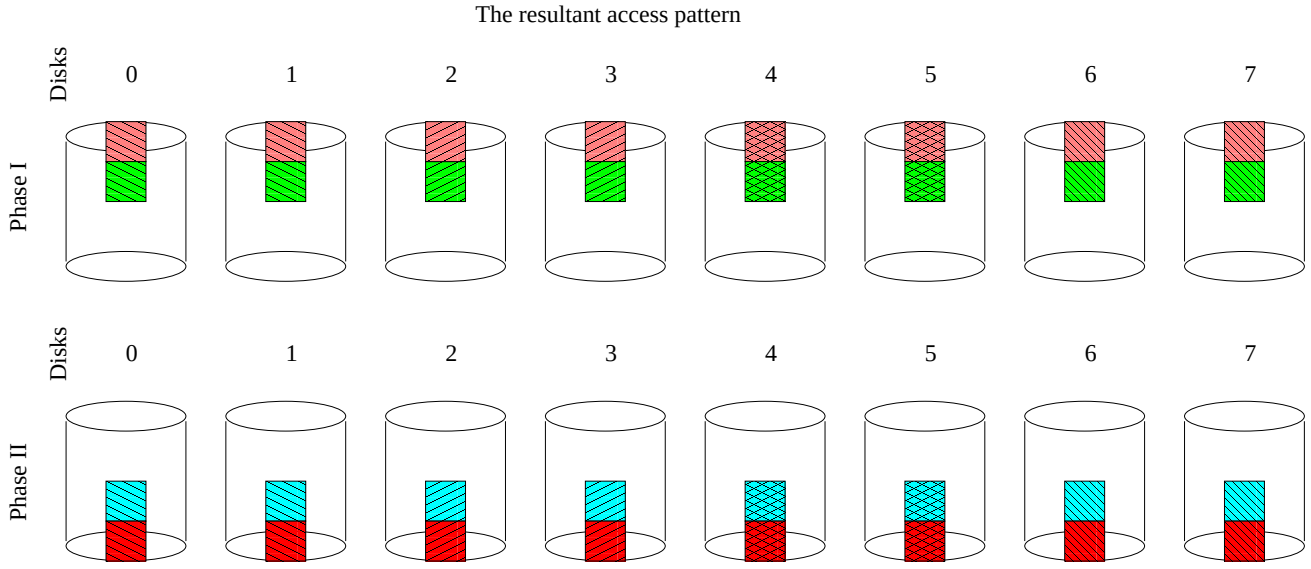


Figure 10: A Good Access Pattern for Storage Strategy in figure 8 : In phase one, objects 0 through 7 are fetched and in the second phase objects 8 through 15 are fetched.

owns the large object resides. Each processor then applies the `average` method to the large object and compares the result with `CONSTANT`. If the `where` clause condition is satisfied, then the `date` and `channel` attributes from the owner row are selected. Hence, if there are N rows in the `raster` table which are distributed in partitions of size N/P among the P processors and in each phase n large objects are loaded (assuming LoadN4All strategy), then there are N/nP phases.

Lets assume that the object space is organized the way shown in Figure 2.6 and the rows are inserted in the chronological order (sorted on the `date` field) into the table. This will result in a pool of objects in which the objects are ordered the same way as their owner records are. Striping this pool of objects across disks will result in a scenario where the smaller the difference between the `date` field of two rows (physically) closer are the `data` objects owned by those rows.

The access pattern generated by this query will depend on the way the `raster` table is partitioned among processors. Assume that the `raster` table is partitioned across 8 processors and there are 8 distinct values for the `channel` attribute and 16 distinct values for the `date` attribute. `(date, channel)` is a key on the table and there are 128 rows in the table. If the `raster` table is range partitioned on the `date` attribute with each processor holding all the rows for a particular value, then we find that a LoadN4All with N = 1 strategy will result in the access pattern similar to one shown in Figure 9. The performance of the query depends on the following characteristics of the access pattern :

1. I/O Parallelism : It measures the average number of disks that active during each I/O phase.

2. Communication Efficiency : Measures the maximum amount of data that is handled by each processor. A communication bottleneck will result if some processors are expected to send/receive significantly more amount of data than other processors.

3. Compute Parallelism : All processors should process approximately same amount data in each phase.

4. I/O Load Balance : Data read from all disks during an I/O phase should be approximately equal.

Table 1 describes the access pattern characteristics of the query under different relation (table) partitioning schemes. It appears that a range partitioning scheme based on the `channel` attribute is the best scheme for the `raster` table. Note that it is the *best scheme for the particular query* under consideration. Table partitioning is not the only source of bad access patterns. Queries that are more complex than the one under consideration can move the rows among the processors which will result in a bad access pattern.

The proposed reordering scheme is based on the proximity of the objects in the logical pool. This has a dual advantage. If data for two objects is next to each other on the same disk, then *request merging* technique can merge the two separate requests to gain better performance. For example, if objects 0 and 4 in Figure 8 are fetched simultaneously will result in 2 I/O requests (one on disk 0 and one on disk 1) which are executed in parallel. The other advantage is that objects close to each other in the logical pool can be potentially fetched in parallel (for example objects 0 and 1 in Figure 8).

	Range Partitioning		Hash Partitioning	
	on `date` attribute	on `channel` attribute	*function1*	*function2*
I/O Parallelism	Worst	Good	Good	Bad
Communication Efficiency	Yes	None	Small	Yes
Compute Parallelism	Good	Good	Good	Good
Overall I/O Load Balance	Balanced	Balanced	Balanced	Balanced

$function1 = date * 10 + channel$ $function2 = channel * 100 + date$

Table 1: Comparison of Various Table Partitioning Schemes for the Query in Section 3

Note that reordering will preserve the semantics of a query. Even if there is an order imposed on the final results (due to a `sort` or `group-by` operator), it does not imply any restrictions on the order in which the results are calculated.

Given an optimal access pattern for a set of large objects, the next task is to schedule the execution of the method on the objects retrieved during each phase. A simple strategy of *owner-computes rule* (the processor where the final result is needed executes the method) may not always work. Referring to Figure 9, the optimal access pattern for the objects is 0, 1, 2, ... 15. If the owner-computes strategy is used then only a subset of the processors can be kept busy sacrificing compute parallelism. A better approach is to selectively *migrate* the method execution. Under this approach, the large object data is directly sent to the processor where the method execution for the object has migrated. The processor then executes the method and sends the result back to the processor on which it is needed.

To summarize how the reordering and migration process works :

1. The processors exchange information about the large objects on which a method needs to be executed. This information consists of the OID of the large object, the method (with corresponding parameters if any) that needs to be executed on the object, the processor that holds row that owns the large object and the row id of the row.

2. An optimal I/O schedule for this set of large objects is generated based on the object space layout.

3. The schedule is annotated by generating the migration information if the optimal schedule does not allow necessary compute parallelism.

4. The schedule is divided in phases depending on the availability of the memory resource and is annotated appropriately.

4 Experimental Results

The experimental evaluation of the technique was carried out on an IBM SP2. Each processor has 128MB main memory and a 2GB (512MB available for user applications) local disk and are interconnected using the high performance switch.

We used the dataset template from the Sequoia 2000 Storage Benchmark [SFGM92] for the results presented here. The schema for the table used is :

```
create RASTER(time = int4, location = box, band = int4, data = int2[][])
```

time is a four byte integer and denotes the half-month over which the raster data was accumulated. The *location* attribute specifies the bounding box for the raster data and is represented by the coordinates of its top-left and right-bottom corners represented by four 4-byte integers. *band* specifies the wavelength band at which the data was captured. *data* is a raster image of size 8MB.

An experimental parallel object-relational database engine developed by us was used. The salient features of this engine include support for fixed and variable length data types, dynamic operator/method loading for the data types and built-in transparent support for large objects. Using this database engine, the `raster` table was partitioned among 8 processors and the pool of `data` objects was striped across local disks attached to these processors. The striping strategy used has the stripe block size equal to the size of the object. Four different table partitioning schemes were used :

1. Range partitioning on the `date` attribute
2. Range partitioning on the `channel` attribute
3. Hash partitioning using the hash function `(date * 10 + channel)`
4. Hash partitioning using the hash function `(channel * 100 + date)`

Four variations of the query described in Section 3 were used.

1. ```
   select raster.date, raster.channel
   from raster
   where raster.data.average() > CONSTANT;
   ```
   This query touches all large objects in the dataset hence accesses around 1GB of data.
2. ```
   select raster.date, raster.channel
   from raster
   ```

```
where raster.data.average() > CONSTANT and
raster.date > MINDATE and raster.date > MAXDATE;
```

In this query `MINDATE` and `MAXDATE` cover half the range of the `date` attribute resulting in accessing 512MB of data.

3. ```
   select raster.date, raster.channel
   from raster
   where raster.data.average() > CONSTANT and
   raster.channel > MINCHAN and raster.channel > MAXCHAN;
   ```

   In this query `MINCHAN` and `MAXCHAN` cover half the range of the `channel` attribute resulting in accessing 512MB of data.

4. ```
   select raster.date, raster.channel
   from raster
   where raster.data.average() > CONSTANT and
   raster.date > MINDATE and raster.date > MAXDATE and
   raster.channel > MINCHAN and raster.channel > MAXCHAN;
   ```

   In this query `MINCHAN` and `MAXCHAN` cover half the range of the `date` attribute and `MINDATE` and `MAXDATE` cover half the range of the `date` attribute resulting in accessing 256MB of data.

Table 2 summarizes the experimental results. Without the reordering and migration optimization, the execution times for the same query vary widely which is attributed to the different access patterns generated by the same query under different table partitioning schemes. Using the reordering and migration technique, ensures consistent and better performance for all table partitioning schemes.

Under the storage organization used, for query 1 and 2 the data to be accessed is spread across all 8 disks. For query 3 and 4 the data is spread across 4 disks. Comparing queries 1 and 2, query 2 accesses data half the size of query 1 and the data is spread across same number of disks as in query 1, hence query 2 performance is better than query 1 given that other parameters (table partitioning scheme and the optimizations used) remain the same. Between query 2 and query 3, the amount of data accessed is the same but the data needed by query 3 is spread across only half as many disks, resulting in lower performance for query 3, other factors being equal. For query 3 and query 4, the latter accesses half the size of data than the former with both needing data that is spread across 4 disks, making query 4 execute faster than query 3, under similar conditions.

Now lets consider the effect of table partitioning scheme on the execution time. Under range partitioning on `date` attribute without the reordering and migration we find that in each I/O phase the data that is fetched lies on only a subset of disks. For example, for query 1 even though the data needed by the query is across all 8 disks, in each I/O phase only one disk is active, that is data fetched during that phase comes entirely from one disk. Similar behaviour happens under the hash partitioning scheme that uses the hash function `channel * 100 + date`. The other two partitioning schemes shown in table 2 do not exhibit this skew in disk accesses.

	Range Partitioning		Hash Partitioning	
	on `date` attribute	on `channel` attribute	*function1*	*function2*
Unoptimized Query 1	558.00	83.25	87.55	193.04
Optimized Query 1	66.14	75.67	75.98	85.65
Unoptimized Query 2	351.59	24.34	24.44	79.78
Optimized Query 2	31.75	23.11	27.12	29.47
Unoptimized Query 3	319.38	72.17	92.54	100.43
Optimized Query 3	56.85	62.82	69.94	56.66
Unoptimized Query 4	219.35	20.48	22.55	45.03
Optimized Query 4	14.64	18.73	23.49	19.17

function1 = `date` * *10* + `channel`

function2 = `channel` * *100* + `date`

Table 2: Experimental Results : The rows labeled *unoptimized* report the query execution time when all I/O optimization techniques except reordering and migration were used. The rows labeled *optimized* use reordering and migration as well as other optimization techniques. The columns report query execution times under different table partitioning schemes.

Important thing here to note is that for each query the data needed by the query is equally distributed across the disks (8 disks for query 1 and 2, 4 disks for queries 3 and 4). When the reordering and migration technique was used, it successfully rearranged the I/O scheduling to maximize the number of disks that are active in each phase. This not only allowed higher I/O bandwidth but also removed the communication bottleneck.

5 Conclusions

This paper presented collective I/O, OID reordering and method execution migration techniques for improving I/O performance in parallel object-relational database system. The I/O optmization problem is tackled in query executor, methods for large objects and in low level I/O runtime system. This work is part of the ongoing research aimed at developing I/O optimization techniques based on collective I/O for parallel object-relational databases. The OID reordering and method execution migration technique is aimed at improving the access patterns of queries to achieve better I/O performance. Our experiments have shown as much as 600% improvement in the query execution time when using this technique. The peper also presented other related techniques namely request merging and data sieving. The plan for future work includes development of communication optmization techniques for collective I/O and other I/O optimization techniques like data prefetching.

References

[DG92] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[DGS+90] D. DeWitt, S. Ghandeharizadeh, D. A. Shneider, A. Bricker, H. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[dRBC93] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved Parallel I/O via a Two-Phase Runtime Access Strategy. In *Proceedings of the Workshop on I/O in Parallel Computer Systems at IPPS '93*, pages 56–70, April 1993.

[EN94] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*, pages 805–807. Addison-Wesley Publishing Company, second edition, 1994.

[Gro88] Tandem Performance Group. A benchmark of non-stop sql on the debit credit transaction. In *Proceedings of the 1988 SIGMOD Conference*, Chicago, Ill., June 1988.

[JK84] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[Kot95] D. Kotz. Expanding the Potential for Disk-directed I/O. Technical Report PCS-TR95-254, Dept. of Computer Science, Dartmouth College, March 1995.

[LOT94] Hongjun Lu, Beng-Chin Ooi, and Kian-Lee Tan, editors. *Query Processing in Parallel Relational Database Systems*, chapter 2. IEEE Computer Society Press, 1994.

[MC98] S. More and A. Choudhary. Extended collective i/o for efficient retrieval of large objects. Technical Report CPDC-TR-9802-009, Center for Parallel and Distributed Computing, Northwestern University, February 1998. URL= `http://www.ece.nwu.edu/cpdc/TechReport/1998/02/CPDC-TR-9802-009.ps.gz`.

[NK96] Nils Nieuwejaar and David Kotz. The Galley Parallel File System. Technical Report PCS-TR96-286, Department of Computer Science, Dartmouth College, 1996.

[PGK88] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 109–116, June 1988.

[PYK+97] Jignesh Patel, Jie Bing Yu, Navin Kabra, Kristin Tufte, Biswadeep Nag, Josef Burger, Nancy Hall, Karthikeyan Ramasamy, Roger Lueder, Ourt Ellmann, Jim Kupsch, Shelly Guo, Johan Larson, David DeWitt, and Jeffrey Naughton. Building a scalable geo-spatial dbms: Technology, implementation and evaluation. In *Proceedings of SIGMOD '97*, 1997.

[SD92] M. Stonebraker and J. Dozier. Sequoia 2000: Large capacity object servers to support global change research. Technical Report SEQUOIA 2000 Technical Report No. 1, Electronics Research Lab, March 1992.

[SFGM92] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The sequoia 2000 storage benchmark. Technical report, EECS Department, University of California, Berkeley, 1992.

[SGM86] K. Salem and H. Garcia-Molina. Disk striping. In *Proceedings of the 1986 IEEE Data Engineering Conference*, February 1986.

[SKPO88] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout. The design of xprs. In *Proceedings of the 14th VLDB Coneference*, pages 318–330. VLDB Endowment, Inc., 1988.

[Sto86] M. Stonebraker. The case for shared nothing. *Database Eng.*, 9(1):4–9, 1986.

[TC96] R. Thakur and A. Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. Technical Report CACR–103, Scalable I/O Initiative, Center for Advanced Computing Research, Caltech, Revised May 1996. (To appear in *Scientific Programming*).

[TCB+96] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and K. Sivaramakrishna. Passion: Optimized i/o for parallel applications. *IEEE Computer*, June 1996.

[YwtPG95] Andrew Yu and Jolly Chen (with the POSTGRES Group). *The POSTGRES95 User Manual*, version 1.0 edition, September 1995.