Extended Collective I/O for Efficient Retrieval of Large Objects [*]

Sachin More [†] Alok Choudhary[†]

Abstract

Object-relational databases management systems (OR-DBMS) 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. In this paper, we examine issues related to parallel processing of queries in object-relational model with respect to efficient storage and retrieval of large objects. We extend the concept of collective I/O and other related techniques like request merging and data sieving in the database domain to achieve high performance in retrieval of large objects. We deal with the I/O optimization problem in the query executor, access methods and the low level runtime system. We also propose a new technique called pooled striping for efficient storage of large objects on multiple disks. The results presented in this paper clearly show the effectiveness of the proposed I/O optimization techniques in handling large amounts of data in a parallel object-relational database system.

1. Introduction

Object-relational databases extend the capabilities of the relational databases by allowing the user to define new data types and methods to operate on these data types while retaining most of the relational model semantics. The size of the user-defined data types can be anywhere from a few bytes (a complex number data type) to several megabytes (a two-dimensional array that contains a million double precision floating point numbers). Hence user-defined data types can be broadly classified in two categories, *small data types* and *large data types*, depending on the size of the data type. Instances of large data types are called *large objects*. While traditional data handling techniques used in relational database systems are sufficient for small data types, specialized techniques are needed to handle the large data types. In this paper, we examine issues related to parallel processing of queries in a parallel object-relational database system with respect to efficient storage and retrieval of large objects.

Management of large objects consists of storage, retrieval and processing of large objects. The large amount of data involved necessitates employing some form of I/O parallelism (storing the data across multiple disks) to achieve high I/O bandwidth. Also, the compute intensive nature of methods that act on large objects make them an ideal candidate for parallel processing by using multiple CPUs. The retrieval process is responsible for generating an efficient I/O schedule that allows exploitation of I/O and compute parallelism to the fullest. We believe that I/O optimizations based on exploiting *only* I/O parallelism (for example, a RAID system) are not sufficient; one also needs to take into account the compute parallelism to fully optimize the I/O accesses. Also, there is a need to manipulate the computation process that allows both I/O and compute parallelism to be exploited at the same time.

A database system provides a unique scenario where the user (using a 4GL language like SQL) provides information as to *what* needs to be done. The database system decides *how* the task should be done. For example, a query only specifies a set of large objects and a method or operator to be applied on each of the large objects. It does not usually specify order in which the large objects should be processed. This gives the database system the freedom to schedule the method/operator execution for the best possible performance. Also the database schema provides useful hints about temporal locality for data items. For example, a table definition containing a large data type indicates that the large objects belonging to the table will be accessed together. A query on that table usually processes a subset of the large objects. This information can be used to design better storage strategy for large objects.

This paper presents a new storage strategy called *pooled striping* and a collection of I/O optimization techniques for accessing large objects. It also gives an experimental evaluation of the I/O optimization techniques. Pooled striping exploits the semantic relationship between large objects of the same type by enforcing spatial proximity among them on the storage media. Temporal locality during query execution (large objects of the same type tend to be accessed together) together with spatial locality guaranteed by pooled striping allow better I/O optimization during query processing. The I/O optimization techniques are integrated into the query execution engine. This allows the query execution engine to manipulate the query schedule to enhance the effectiveness of the I/O optimization techniques. Four different I/O optimization techniques are presented here. *Collective I/O* provides the basic framework for the other three opti-

[*]This work was supported in part by NSF ASC-9707074, NSF-CCR-9796027 and NSF-CCR-9796029.

[†]Center for Parallel and Distributed Computing, Department of Electrical and Computer Engineering, Northwestern University

mization techniques; it facilitates fetching and processing multiple data items in parallel. Both, *request merging* and *data sieving* reduce the number of I/O accesses when accessing two or more data items that exhibit spatial locality. *OID reordering* manipulates the query schedule to exploit spatial locality among data items.

1.1. Related work

Storing data across multiple disks to achieve higher I/O bandwidth was proposed in [8] 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 [1, 14]. The SEQUOIA 2000 research project [10] 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 [11] which is aimed at capturing the basic requirements for collection, storage and querying of Earth Science data. We use this benchmark to present the experimental results in this paper. More recent work in this area can be found in the Paradise Project [7]. 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 [2] 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.

The rest of the paper is organized as follows. Section 2 gives an overview of the I/O issues in parallel object-relational databases. Section 3 describes the basic framework on which the I/O optimizations are built. Section 4 discusses the proposed I/O optimization techniques. Section 5 describes the storage strategy. Section 6 discusses the computation scheduling scheme. Section 7 presents experimental results. Section 8 concludes the paper.

2. Parallel object-relational database systems

2.1. Parallel database architectures

Parallel database system architectures can be classified in three basic categories [4]: *Shared Everything (SE)*, *Shared Nothing (SN)* and *Shared Disk (SD)*. [9] 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 [2, 13]. 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

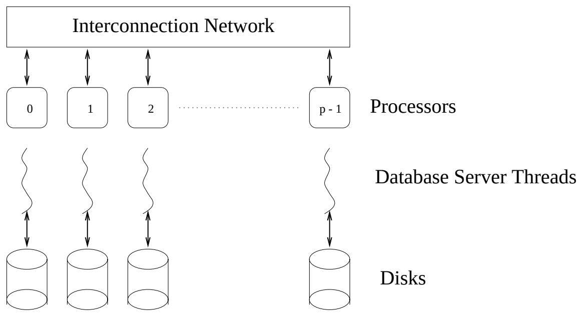

Figure 1. Computation and I/O Model

partitioning as shown in Figure 1. Processors are connected using a fast interconnection network. A processor that needs to access data on a remote disk contacts the owner processor of the remote disk. The owner processor does I/O on behalf of the requesting processor. The interconnection network is used to transfer the data between the requesting processor and the owner processor. 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*.

2.2. I/O issues in parallel database systems

Given hardware storage resources (secondary storage), data layout strategy determines the physical organization of data on the storage media. For parallel relational database systems, data layout is determined based on how the tables are declustered among processors (e.g., round-robin, range partitioning or hash partitioning). For parallel object-relational database systems, other factors like storage for large objects need to be considered.

To illustrate this issue, consider a table containing a large data type as one of the attributes. If the large object is stored inside the row, the size of the row becomes several megabytes. A query that does not refer to the large data type attribute can incur significant I/O penalty for the following reasons: (1) Database systems do I/O in terms of pages instead of rows for performance reasons. Hence, the large attribute will be filtered out (using *projection*) only *after* the row has been read. (2) Rows from a table are cached by the database system to improve performance. The success of the caching scheme will be severely compromised since the cache can accommodate only a few rows from the table. (3) In a non-trivial query plan that processes large objects, a row is expected to move between processors (e.g., *join* and *sort* operators usually need to move rows). Storing the large objects inside the row will result in overloading the inter-connection network and tremendous overheads.

Typically large objects are not stored inside their owner row [16]. 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 large objects. An important issue in object space organization is I/O load balancing. The object space is distributed over a number of disks associated with different processors, 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 the amount of data accessed by each node can mean a significant degradation in 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 pre-processing*. This effectively slows down other processors that need data stored on the heavily-loaded node. Another implication is that the processor(s) doing more I/O need larger buffers and/or require more steps in the I/O schedule. Hence, I/O imbalance can reduce the effectiveness of I/O optimization techniques. I/O load balancing is dependent on the data layout strategy employed. There is no *a priori* layout strategy that is guaranteed to work for all types of datasets and queries on those datasets. Rather one has to choose an initial layout strategy based on heuristics and then coordinate I/O accesses to avoid I/O imbalance.

I/O for table space can be handled by traditional techniques [3, 12, 2]. Object space on the other hand needs specialized I/O optimization techniques. Note that object space I/O optimization problem is different from the I/O optimization and tuning issues in a RAID-type parallel disk system; a RAID is a tightly coupled system whereas a set of local disks attached to multiple processors is a loosely coupled system.

3. I/O optimization in the object space

The database system can handle the I/O optimization problem at three different levels. The techniques used at each level vary and are directly related to the type of information available at that level.

3.1. 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 from the Sequoia 2000 Benchmark:

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

The *clip* method from the original query is replaced by *average* method. After selecting rows from the *RASTER* table that match 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 2.2), 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 individual destinations.

3.2. Access methods for large objects

Access methods know the internal structure of the large objects. For example, the *clip* method in query 2 from the Sequoia 2000 Benchmark is aware of the fact that the large object its dealing with is a two dimensional array of size 1600X2560 with each element of size 2 bytes. Based on this information and the *RECTANGLE* parameter, it can carry out I/O optimization techniques that are specific to array objects [15].

3.3. Low level I/O routines

The low level I/O routines have information as to the exact physical location of the 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 prepare the communication schedule to channel the data to the appropriate processor. [6] describes a multi-phase I/O strategy which can be extended for I/O optimizations at this level.

4. I/O optimization techniques

4.1. Collective I/O

Collective I/O is used to optimize I/O requests in a multiprocessor environment. The basic idea behind this technique is to coordinate I/O accesses from different processors. The processors exchange information regarding what data each of them needs to access. This information is used to derive 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.

4.2. Request merging and data sieving

This technique is aimed at reducing at number of I/O requests. The average time to find and transfer an arbitrary disk block is $s + rd + btt$, where s is the seek time, rd is the rotational delay and btt is the 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 2). It 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 3).

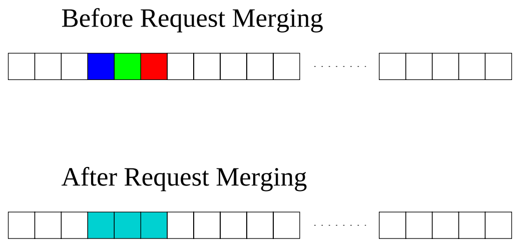

Figure 2. Request Merging : Three distinct I/O requests are converted into one request.

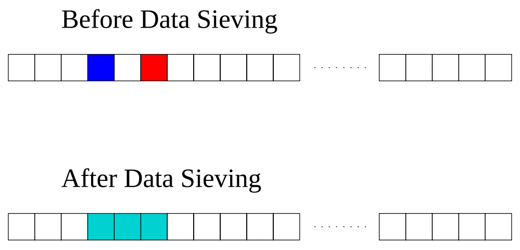

Figure 3. Data Sieving

We implemented the collective I/O technique both at query executor level and at low level I/O runtime system. We do not use collective I/O at the access method level because a single large object is usually processed by a single processor which reduces the effectiveness of the technique. In the query executor, the processors carry out a complete exchange of the OIDs of the objects that need to be read. This allows the query executor to reorder the OIDs based on *object proximity*. A partial reordering of the OIDs is done based on the *logical proximity* of the large objects they point to in the object pool (Figure 4). The figure shows logical pool of objects at the top, the numbers indicating their OIDs, indicating spatial proximity between the objects. On the left is shown the original schedule of accesses. For example, processor 1 needs to access objects with OIDs 10, 3 and 6. According to this schedule the first phase of I/O will access objects with OIDs 1, 10, 7 and 4 (one object per processor). The second phase will access objects with OIDs 9, 3, 11 and 8 and the rest are accessed in phase three. Note that accesses according to the original schedule result in I/O requests for objects that are distributed randomly in the pool. The reordering technique shuffles the order in which the objects are accessed so that in each I/O phase objects that are close to each other are accessed. On the right is shown the I/O schedule after reordering. The reordering technique allows the query executor to execute I/O requests for large objects that are closer to each other. This increases the effectiveness of the *request merging* and *data sieving techniques* because the logical proximity of the objects implies physical proximity of the data due to the round-robin striping strategy used as discussed in Section 5.

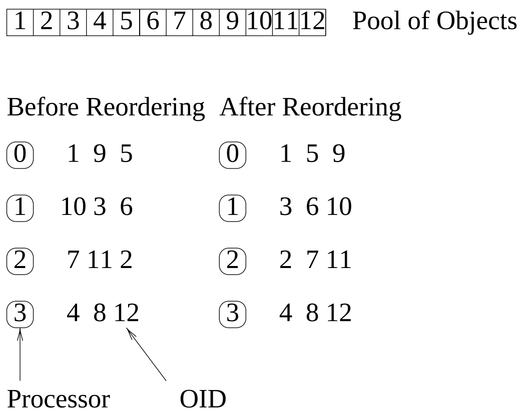

Figure 4. OID Reordering

5. Object space organization

This section discusses the object space organization in more detail. Consider a set of disks $\mathcal{D}$ each containing $\mathcal{B}$ blocks of size $\mathcal{BS}$. $\mathcal{D}$ is also called the striping factor. To store data of size $\mathcal{S}$, the data is broken up in blocks of size $\mathcal{BS}$ giving total $\lceil \frac{\mathcal{S}}{\mathcal{BS}} \rceil$ blocks. These blocks are then distributed over the $\mathcal{D}$ disks in a round-robin fashion. Note that the block size $\mathcal{BS}$ used here may not be same as the *physical disk block size*, but the former is usually set to an integer multiple of the latter for performance reasons. The exact value of the multiple will depends on the average request size. Since block size sets the lower limit on the size of the I/O request generated by the database system, a value much higher that average request size will result in lower performance (Section 7). The stripe factor, on the other hand, is more difficult to predict. A value too low will mean low bandwidth, whereas, too a high value will mean more messages (Section 7).

5.1. Motivation

Storage strategy plays an important role in the success of the I/O optimization techniques. A storage strategy should strive for:

I/O parallelism: Data items should be stored in such a way that I/O accesses should find the relevant pieces of data equally distributed across as many disks as possible. This allows for reading the data on the disks in parallel, resulting in higher I/O bandwidth. In an ideal storage strategy that stores data across $\mathcal{D}$ disks, any I/O request that fetches $\mathcal{N}$ bytes of data will find $\frac{\mathcal{N}}{\mathcal{D}}$ of the data on each disk.

Spatial locality: Accessing data from a single disk should result in as few disk I/O requests as possible. To achieve that, the data should be contiguous on disk. As noted in Section 4.2, the request merging and data sieving techniques can exploit spatial proximity of the data to accessed to reduce the number of I/O requests. In the ideal case, if $\mathcal{N}$ bytes of data is accessed from a single disk, then there should be a single disk access.

5.2. Pooled striping

We designed a new technique called *pooled striping* 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 disks. Since the pool is striped across *all* available disks, one can exploit I/O parallelism to the fullest. Any contiguous I/O request on the pool bigger than $\mathcal{BS} \times \mathcal{D}$ bytes will ensure that all disks are active and the amount of data fetched from each of them is approximately equal. Also the large objects of the *same type* exhibit *temporal locality*. This is because a query operating on a table that refers to them selects a subset of the objects contained in the pool. During query execution, this temporal locality is translated in spatial locality because all the objects accessed by the query are together in the same pool.

Note that pooled striping is *not* just individual striping with the starting disk for each object follows the last disk for the previous object. Pooled striping introduces the notion of a set of related objects. It combines *temporal locality* (objects that are accessed in the same query) with *spatial locality* (by putting them together in the same *pool*). This allows us to apply I/O optimization techniques on a group of objects at the same time instead of applying them individually on each object. Each large object by itself provides very little I/O parallelism benefits since it occupies a few megabytes reducing the effectiveness of the optimization techniques discussed in Section 4. Also, the 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.

5.3. I/O cost estimation

The I/O cost estimation needs to consider not only the time needed to access secondary storage but also the communication overhead associated with I/O optimization techniques and the main memory usage. We use the following metrics to compare two query execution plans:

(1) total amount of data accessed
(2) total number of I/O requests
(3) total amount of data communicated
(4) total number to messages generated
(5) maximum amount of memory needed
(6) maximum amount of data accessed
(7) maximum amount of data sent
(8) maximum number of messages sent
(9) maximum amount of data received
(10) maximum number of messages received

The first four metrics give an overall picture of how optimal the I/O execution plan is. For example, the first metric only looks at the total amount of data accessed without taking into consideration other factors like from how many disks it was accessed and how much data was accessed from each disk. Hence they give only a rough estimate of I/O efficiency. The fifth metric together with the amount of physical memory available, restricts the number of feasible query plans. For example, if 5 objects per processor need to be accessed from 4 processors and each processor can hold maximum two objects at the same time, one can eliminate query plans that perform object retrieval in less than three phases (Section 6). The remaining metrics give finer details about I/O load balancing and how efficiently the interconnection network is used. These metrics identify I/O and communication bottlenecks. For example, if metric 6 is much larger than metric 1 divided by number of disks, it indicates a problem with either storage strategy (lack of sufficient I/O parallelism) or query plan (bad I/O scheduling). Section 7 uses these metrics to explain the experimental results.

6. Computation scheduling

Next we consider retrieval strategies for large objects. The query executor needs to retrieve multiple large objects on each node and process them in parallel. As noted in Section 2, typical datasets occupy several gigabyte of data which prevents keeping the entire dataset in the main memory. The amount of data that needs to be accessed, which varies depending on the query, may prohibit loading all the data at the same time. Hence, the data needs to be loaded in multiple phases and results from each stage need to be saved (if needed) before loading the data for the next stage. In the case where data accessed by a query fits in the main memory, it should be loaded in a single phase. Given these constraints, the query executor needs to schedule large object processing in multiple phases. In each phase, a processor is permitted to load data that fits in its main memory.

LoadN strategy: This is a naive strategy where $\mathcal{N}$ large objects belonging to a *single* processor are retrieved in a single phase and the process is repeated till all objects belonging to all the processors are processed. The left part of Figure 5 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 $\mathcal{N}$=2 will need 8 stages, two for each processor, to retrieve the complete set of objects. 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 retrieved objects in a sequential fashion.

LoadN4All Strategy: A better strategy is to load $\mathcal{N}$ objects for *all* processors in a single phase This strategy gives both I/O and compute parallelism. The I/O parallelism is

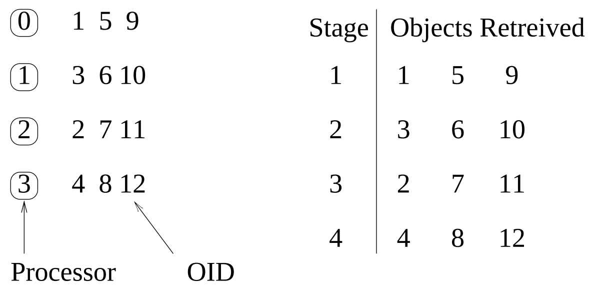

Figure 5. LoadN Strategy (N=3)

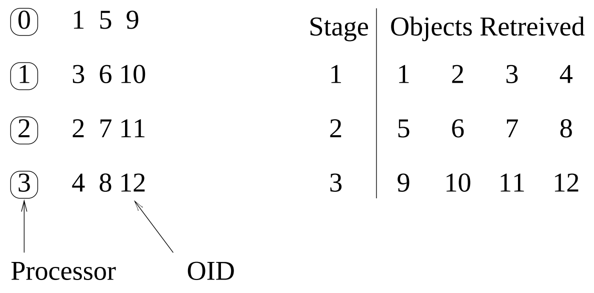

Figure 6. LoadN4All Strategy (N=1)

obtained by retrieving multiple large objects in parallel. The compute parallelism results from processing these objects in parallel, since they belong to different processors.

The left part of Figure 6 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, $\mathcal{N}$ 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.

The LoadOne4All strategy not only offers both I/O and compute parallelism, it also fits the framework provided by collective I/O technique. The each phase, each processor generates I/O requests for the objects that it needs to process in that phase. These I/O requests are then collected together to generate an I/O schedule for that phase. The success of the strategy depends on the selection of objects to be retrieved in each phase. We do a partial ordering of the objects based on their spatial proximity. This allows us to generate a compute schedule in which objects retrieved in each phase exhibit spatial locality. Details of this technique are not presented here due to space constraints ([5]).

7. Implementation results

The results were taken using a parallel object-relational database engine we developed. The details of this engine are not presented here due to space constraints. We chose the Sequoia 2000 Storage Benchmark for the results presented in this section. The benchmark provides three separate datasets, *regional*, *national*, and *world*. We use the regional dataset which is a little over 1GB in size and covers an area of $1280kmX800km$ comprising of the states of California and Nevada at spatial resolution of $0.5kmX0.5km$. A subset of the queries from the benchmark, ones that manipulate raster images, is used for the results presented here. The schema for the tables used in these queries is:

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

time is a four byte integer and denotes the half-month over which the raster data was accumulated. The data set contains data for one year, with the values $1, 2, \ldots, 26$. 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; the values range over $1, 2, 3, 4, 5$. *data* is a two dimensional array of size $2560X1600$. Each point is a two byte integer. There are 150 records in the data set. The RECTANGLE constant is set to a region covering an area of $100kmX100km$. The BAND constant is set to 1.

The experiment was carried out on a 16 processor (12 of these are available for application programs) IBM SP2. Each processor has 128MB main memory and a 2GB (512MB available for user applications) local disk. Different storage layout were evaluated by varying the number of disks used to store the large objects as well as the block size.

The experiment uses query 2 from the benchmark.

```
retrieve (clip(RASTER.data, RECTANGLE),
RASTER.time) where RASTER.band = BAND
```

This query needs to access 26 `data` objects. The total amount of data needed from each object is $80KB$. The query returns about 2.1MB of data. First the `select` operator is applied to the `RASTER` table which selects rows whose `band` attribute is equal to `BAND`. This result is stored in a temporary table, `T1`. Table `T1` contains 26 rows. Then the `project` operator is applied to table `T1` which projects the `time` and `data` attributes and stores the result in a temporary table `T2`. Table `T2` contains 26 rows. Table `T2` is then sorted on the `time` attribute to create table `T3`. Table `T3` also contains 26 rows. The `clip` operator is then applied to table `T3` to compute the final result. This step retrieves 26 partial large objects, one per row of table `T3`.

Figure 7 gives summary results for the execution of the *clip* operator. Since the large objects are accessed only during the execution of the `clip` operator, the execution time is reported only for that operator. The experiments were carried out for blocks sizes from 8MB (where entire raster image fits inside one block) to 64KB (where each block accommodates $\frac{1}{128}$ of the raster image). We find that 512MB (each block accommodates $\frac{1}{16}$ of the raster image) is the most optimal block size.

The number of nodes across which the data is spread also affects the overall timing. Larger the number of I/O nodes, greater is the potential I/O bandwidth. The effect of varying the total number of nodes is mixed. One reason for this behavior is that the `clip` method is not compute intensive, and therefore, compute parallelism cannot be exploited effectively. The only gain that the increased number of compute processors provides is the increase in the amount of main memory available which results in more data being fetched in a single step of the I/O schedule. For a given number of disks there is an upper limit at which they can supply data. Thus, for a small number of I/O nodes, increasing the number of compute processors does not result in any improvement in performance. Increase in the number of I/O nodes ensures that the disks can supply data at sufficient bandwidth, allowing better performance for higher number of compute nodes.

Next we analyze the results in terms of components of the total execution time. For the same number of compute and I/O nodes, the I/O time decreases as we decrease the block size till the optimal block size is reached. Any further reduction in block size does not reduce the I/O time. Since I/O operations are carried out on blocks, if either block size or I/O request size is not a multiple of the other, extra data is read which is eventually discarded. For block sizes larger than the I/O request size, a reduction in block size results in less amount of useless data being read. For block sizes smaller than I/O request size, more than one block is read per request which result in more communication overhead.

Another factor that affects the I/O cost is the number of I/O nodes which are actually involved in I/O. For very high block sizes (comparable to the size of the object being read) *inter-object I/O parallelism* (multiple objects stored on different disks which are read in parallel) is dominant and a uniform distribution of objects among the disks guaranteed by pooled striping keeps all I/O nodes active. For low block sizes, *intra-object I/O parallelism* (different parts of the object being read in parallel) ensures that all I/O nodes are active. Also at very low block sizes the setup cost of the I/O operation (Section 4.2) starts dominating the total cost of the operation, which results in I/O cost remaining relatively the same for different block sizes. For the same number of compute nodes and block size, increasing the number of I/O nodes decreases the I/O time since the data being read is distributed evenly among the I/O nodes and more I/O nodes means greater I/O bandwidth. For the same number of I/O nodes and block size varying the number of compute processors has little or no effect on the I/O time. The communication time depends on the number of messages exchanged and the amount of data exchanged. If the sender is delayed due to load imbalance (compute, I/O or communication) then the receiver is delayed too). As the block size is decreased the amount of data transferred between processors also decreases. But the number of messages exchanged between processors increases due to an increase in the number of blocks and meta-information about the blocks that is exchanged between the processors. Consequently, the send operation time decreases as the block size decreases but it starts increasing again due to extra message overhead for low block sizes. Increasing the number of I/O nodes while keeping the number of compute nodes and the block size constant reduces the send operation time because the predominant users of the send operation are the I/O nodes which need to send the data to other processors. Since the same amount of data is being read in all cases, more senders means less amount of data to be communicated per sender. It also increases the amount of data that an I/O node reads which is intended for the same processor resulting in a local memory copy operation which is cheaper than a send operation. For small block sizes, an increase in the number of compute nodes means an increase in the number of messages carrying the block meta-information. For very high block sizes the amount of data exchanged between processors increases which results in a heavy load on the interconnection network.

8. Conclusions

The work presented in this paper mainly focussed on extending the collective I/O and other related techniques to the ORDBMS domain. We also evaluated two new techniques: pooled striping for better storage management and OID reordering (the results for this technique are not presented here due to lack of space). Collective I/O which is a generic technique provides the framework necessary to allow multiple processors to coordinate their I/O requests. Data sieving and request merging techniques were used to optimize the number of disk accesses. The most important factor that determines available speedups is the number of distinct objects accessed. Inter-object I/O parallelism is found to be more conducive for better speedups than intra-object I/O parallelism. This proves the basic concept behind pooled striping, that storage strategies in a parallel database systems should try to optimize storage for a collection data items rather than storage for individual data items. The results also validate the effectiveness of collective I/O in terms of speedups in query execution times. We are currently performing more experiments to evaluate various other I/O optimization techniques including OID reordering.

References

[1] 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.

[2] D. DeWitt et al. The Gamma Database Machine Project. *IEEE Trans. Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[3] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[4] H. Lu, B.-C. Ooi, and K.-L. Tan, editors. *Query Processing in Parallel Relational Database Systems*, chapter 2. IEEE Computer Society Press, 1994.

[5] S. More and A. Choudhary. Query Optimization using Collective I/O, OID Reordering and Migration. Technical Report CPDC-TR-9802-014, Center for Parallel and Distributed Computing, Northwestern University, February 1998.

[6] S. More, A. Choudhary, I. Foster, and M. Xu. MTIO A Multi-threaded Parallel I/O System. In *Proceedings of IPPS'97*, 1997.

[7] J. Patel et al. Building a Scalable Geo-Spatial DBMS: Technology, Implementation and Evaluation. In *Proceedings of SIGMOD '97*, 1997.

[8] 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.

[9] M. Stonebraker. The Case for Shared Nothing. *Database Eng.*, 9(1):4–9, 1986.

[10] 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.

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

[12] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout. The Design of XPRS. In *Proceedings of the 14th VLDB Conference*, pages 318–330. VLDB Endowment, Inc., 1988.

[13] 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.

[14] R. Thakur and A. Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming*, 5(4):301–317, Winter 1996.

[15] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and K. Sivaramakrishna. PASSION: Optimized I/O for Parallel Applications. *IEEE Computer*, June 1996.

[16] A. Yu and J. Chen. *The POSTGRES95 User Manual*, version 1.0 edition, September 1995.

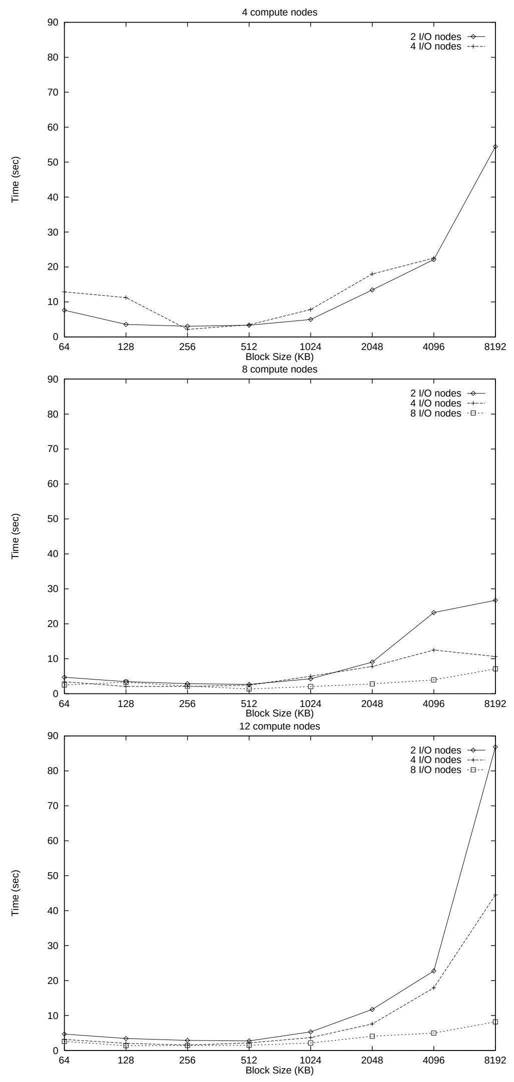

Figure 7. Summary results for the experiment