Extended Collective I/O for Efficient Retrieval of Large Objects

Sachin More and Alok Choudhary

Technical Report No. CPDC-TR-9802-009

January 1998

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

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

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

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. 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 for I/O in the database domain to achieve high performance in retrieval of large objects and their use in efficient query processing. The I/O optimization problem is dealt in query executor, access methods as well as in the low level runtime system. We also propose a new technique *pooled striping* for efficient storage and retrieval of large objects on multiple disks.

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

1 Introduction

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. While this adds a good amount of flexibility to the relational model, many new issues regarding efficiency arise. One important difference between relational and object-relational model is the amount data that needs to be handled while processing a query. For example, Geo-spatial datasets typically need to store raster images whose sizes are in the range of several megabytes and are appropriately called *large objects*. 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.

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. The I/O optimization problem is addressed in query execution, access methods as well as in the low level runtime system. We also propose a new technique called *pooled striping* to store and access the large objects on multiple disks.

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. We use this benchmark in this paper to present the experimental results. 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.

Section 2 gives an overview of the I/O issues in parallel object-relational databases. Section 3 discusses the proposed I/O optimization techniques. Section 4 describes data layout strategies. Section 5 presents experimental results. Section 6 concludes the paper.

2 Parallel Object-Relational Database Systems

There is an ever increasing demand on the database systems to be able to store more data than ever. [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 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.

Parallel database system architectures can be classified in to three basic categories [LOT94], *Shared*

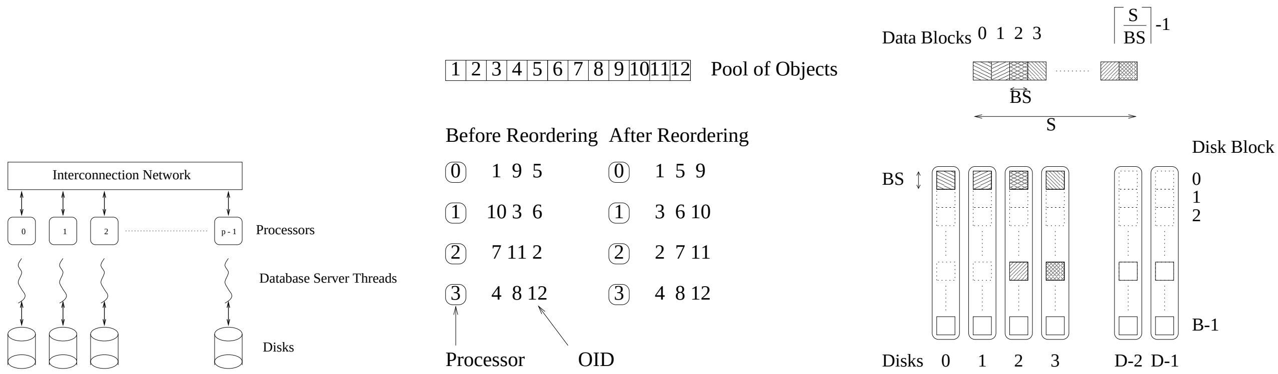
Everything (SE), *Shared Nothing (SN)* and *Shared Disk (SD)*. [Sto86] compares the above three architectures based on different issues involved in developing a parallel database system. The SN architecture scores over the rest mainly due to excellent scalability and provides near-linear speedups and 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)). 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*.

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 using either 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 that contains and stores a large object in one of its columns, making the size of the row in megabytes. One problem is that a query that does not refer to an attribute containing the large data type can incur significant I/O penalty. Also, rows from a table are cached by the database system to improve performance. 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 both in sequential and parallel object-relational database systems. A problem specific to a multiprocessor environment is related to inter-processor communication. In a non-trivial query plan, a rows is expected to move between processors (*join* and *sort* operators usually need to move rows). Huge size of each row will result in overloading the inter-connection network and tremendous overheads. Hence, large objects can 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 large objects. An important issue in object space organization 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 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 preprocessing* (collective I/O, a technique discussed in section 3 needs such 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 will need either bigger buffers than the rest and/or need more steps in the I/O schedule. Hence, I/O imbalance can reduce the effectiveness of the 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 optimize I/O accesses during query execution.

I/O for table space can be handled by traditional techniques [JK84, SKPO88, DGS+90]. Object space on the other hand needs specialized I/O optimization techniques. I/O optimization problem can be handled at three levels. The techniques used at each level vary and are directly related to the type of information available at that level. *Query Executor* is the highest level at which I/O optimization can be done. Information available at this level is *logical*. Using this information, the query executor



(a) Computation and I/O Model (b) OID Reordering (c) Data striping

Figure 1: OID Reordering and Data Striping

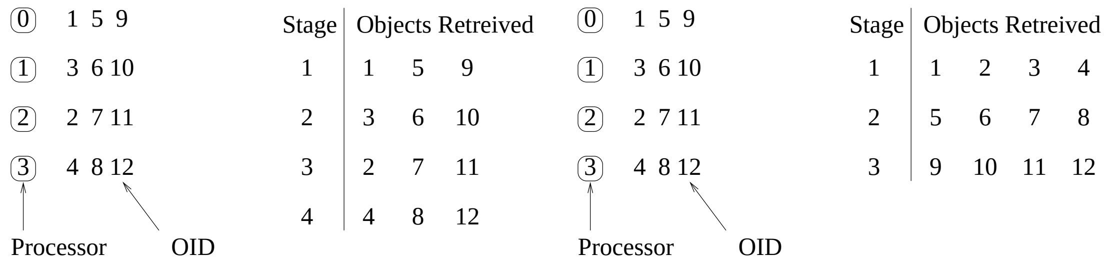
has to come up with a schedule that determines the order in which large objects are retrieved and the computations are scheduled. *Access Methods for Large Objects* know the internal structure of the large objects. For example, a *clip* method which selects a section from an array, is aware of the fact that it is accessing a multi-dimensional array of given size. Based on this information, it can carry out I/O optimization techniques that are specific to array objects [TCB+96]. The *Low Level I/O Runtime System* have 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.

3 I/O Optimization Techniques

Collective I/O This technique is used to optimize I/O requests in multi-processor environment. The basic idea behind the technique is to co-ordinate I/O accesses from different processors. The processors exchange information regarding what data each of them needs to access. This information is used 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.

We implemented 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 1(b)). The reordering allows 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* discussed in section 3 because the logical proximity of the objects implies physical proximity of the data due to the round-robin striping strategy used (section 4).

Next we consider retrieval strategies for large objects. At this point of time, the query executor needs to retrieve multiple large objects on each node. The amount of data that needs to be accessed prohibits loading all of them simultaneously. Hence, the data needs to be loaded in multiple phases



(a) LoadN Strategy (N=3) (b) LoadN4All Strategy (N=1)

Figure 2: Object Loading Strategies

and results from each stage need to be saved before loading the data for the next stage.

In *LoadN strategy*, $\mathcal{N}$ large objects belonging to a *single* processor are retrieved in a single stage (figure 2(a)). 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. In *LoadN4All Strategy*, $\mathcal{N}$ objects for *all* processors are loaded in a single stage (figure 2(b)). This strategy gives both I/O and compute parallelism. The success of LoadN4All strategy depends on 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.

In the low level I/O runtime system, collective I/O is used to coordinate accesses to the physical storage. Either one processor (for LoadN) or multiple processors (for LoadN4All) need to access one or more (depending on the value of N) large objects. These requests are then translated into *page (or block) requests*. These page requests are then exchanged between the processors to come up with an efficient I/O schedule. Among other things, *request merging* (section 3) technique is used here to compute the schedule. Since the amount of data handled is very large and is distributed across processors, communication optimization is also necessary.

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 [EN94] :

$s + rd + btt$

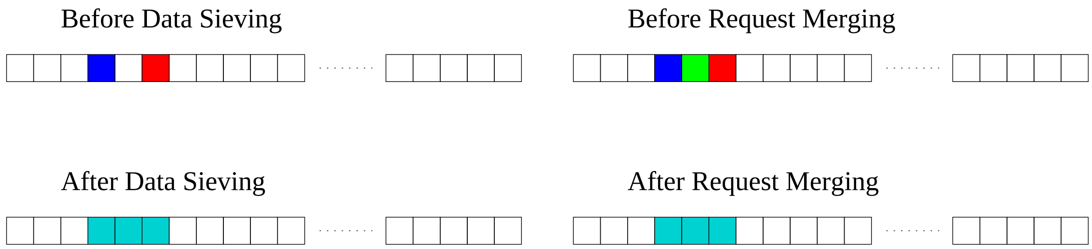
$where \quad s = seek\ time \quad rd = rotational\ delay \quad btt = block\ transfer\ time$

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

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 3(b)). 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 3(a)). It is usually used to access sections from large multi-dimensional arrays [CBH+94].

4 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 $\mathcal{D}$



(a) Data Sieving (b) Request Merging

Figure 3: Data Sieving and Request Merging

disks in a round-robin fashion (figure 1(c)). 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.

Striping each large object independently may not give sufficient speedup because each large object by itself occupies only a few megabytes, reducing the effectiveness of the optimization techniques discussed in section 3. Also, during query execution, I/O requests for different large objects appear randomly distributed at the file system/disk level. These drawbacks of instance striping can be overcome by using the proposed *pooled striping*. In pooled striping, large objects that belong to the same data type are concatenated together to form a *pool* of large objects. This pool is then striped across disks. We use pooled striping for storing large objects.

A good estimate for block size is integer multiple of disk block size. One also needs to consider the average request size for partial retrieval. 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 5). 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.

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. Following metrics are used 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 from any single disk
(7) Maximum amount of data sent from any processor
(8) Maximum number of messages sent by any processor
(9) Maximum amount of data received by any processor
(10) Maximum number of messages received by any processor

The first four metrics give an overall picture of how optimal the I/O execution plan is. The fifth metric together with the amount of physical memory available, restricts the number of feasible query plans. The remaining metrics give finer details about I/O load balancing and how efficiently the interconnection network is used. Section 5 uses these metrics to explain the experimental results.

5 Implementation Results

Experiment Plan The results were taken using a parallel object-relational database engine developed by us. The details of this engine are not presented here due to space constraints. We chose the Sequoia 2000 Storage Benchmark [SFGM92] 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 $1280km X 800km$ comprising of the states of California and Nevada at spatial resolution of $0.5km X 0.5km$. 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(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 data set contains data for one year, with 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 $2560 X 1600$. 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 $100km X 100km$. The BAND constant is set to 1.

The experiments were 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 and are interconnected using the high performance switch. The experiments were carried out on 4, 8 and 12 processors. Different storage layout were evaluated by varying the number of disks (used to store the large objects) as well as the block size.

Table 1: Summary Results for Experiment 1 : The results are sorted on compute nodes, I/O nodes and blocks size in that order for a row-wise traversal.

Compute nodes	I/O nodes	Block Size	Time (sec)	Compute nodes	I/O nodes	Block Size	Time (sec)	Compute nodes	I/O nodes	Block Size	Time (sec)	Compute nodes	I/O nodes	Block Size	Time (sec)
4	2	64K	7.63	4	2	128K	3.57	4	2	256K	3.05	4	2	512K	3.34
4	2	1024K	4.99	4	2	2048K	13.45	4	2	4096K	22.13	4	2	8192K	54.45
4	4	128K	12.88	4	4	256K	11.26	4	4	512K	2.15	4	4	1024K	3.48
4	4	2048K	7.83	4	4	4096K	18.01	4	4	8192K	22.56				
8	2	64K	4.71	8	2	128K	3.45	8	2	256K	2.87	8	2	512K	2.67
8	2	1024K	4.27	8	2	2048K	9.04	8	2	4096K	23.22	8	2	8192K	26.73
8	4	64K	3.44	8	4	128K	2.07	8	4	256K	2.09	8	4	512K	2.43
8	4	1024K	5.01	8	4	2048K	7.79	8	4	4096K	12.50	8	4	8192K	10.65
8	8	64K	2.52	8	8	128K	3.27	8	8	256K	2.20	8	8	512K	1.37
8	8	1024K	2.06	8	8	2048K	2.82	8	8	4096K	3.95	8	8	8192K	7.12
12	2	64K	4.70	12	2	128K	3.44	12	2	256K	2.90	12	2	512K	2.75
12	2	1024K	5.33	12	2	2048K	11.76	12	2	4096K	22.75	12	2	8192K	86.85
12	4	64K	3.18	12	4	128K	2.05	12	4	256K	1.56	12	4	512K	2.17
12	4	1024K	3.69	12	4	2048K	7.60	12	4	4096K	17.93	12	4	8192K	44.49
12	8	64K	2.59	12	8	128K	1.39	12	8	256K	1.44	12	8	512K	1.49
12	8	1024K	2.15	12	8	2048K	4.07	12	8	4096K	4.98	12	8	8192K	8.21

Experiment 1

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

The first experiment uses query 2 from the benchmark. 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`.

Table 1 gives summary of the results. Since the large objects are accessed only during the execution of the `clip` operator, the execution time is reported only for that operator. First consider the effect of the block size on the query execution time. The experiments were carried out for blocks sizes from 8MB (entire raster image fits inside one block) to 64KB (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, which in turn reduces total number of steps in the I/O schedule. A single disk can provide only so much bandwidth. Hence 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 bigger 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 3) 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. The receive operations time is also affected by the amount of synchronization between the participating processors (A receive operation has to wait for the sender to start sending the data. 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 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 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 that they read 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. The number of compute nodes does not affect the send operation time except at very low and very high block sizes. For small block sizes, the increases in the number of compute nodes means an increase in the number of messages carrying the block meta-information. This effect is not so dominant in mid-range block sizes. For very high block sizes the amount of data exchanged between processors increases which results in a heavy load on the interconnection network. The receive operation times follow the same pattern as the send operation times. Block size affects the receive time due to different amount of data being received. Note that the receive operation times are the worst for the nodes that are *not* involved in I/O (either because the node is not an I/O node or the data being read does not reside on that node). Major portion of the time is spent by these nodes waiting for the I/O nodes to finish reading the data. Hence, a lower number of I/O nodes implies worse receive operation times.

Table 2: Summary Results for Experiment 2 : The results are sorted on compute nodes, I/O nodes and blocks size in that order for a row-wise traversal.

Compute nodes	I/O nodes	Block Size	Time (sec)	Compute nodes	I/O nodes	Block Size	Time (sec)	Compute nodes	I/O nodes	Block Size	Time (sec)	Compute nodes	I/O nodes	Block Size	Time (sec)
4	2	64K	16.14	4	2	128K	15.54	4	2	256K	15.12	4	2	512K	15.55
4	2	1024K	15.25	4	2	2048K	14.94	4	2	4096K	15.08	4	2	8192K	16.00
4	4	64K	13.89	4	4	128K	14.99	4	4	256K	15.09	4	4	512K	14.37
4	4	1024K	14.51	4	4	2048K	14.95	4	4	4096K	15.74	4	4	8192K	15.55
8	2	64K	15.03	8	2	128K	13.52	8	2	256K	13.16	8	2	512K	14.10
8	2	1024K	13.98	8	2	2048K	13.90	8	2	4096K	13.37	8	2	8192K	14.39
8	4	64K	11.65	8	4	128K	11.36	8	4	256K	11.33	8	4	512K	11.53
8	4	1024K	11.62	8	4	2048K	10.97	8	4	4096K	13.07	8	4	8192K	15.11
8	8	64K	10.31	8	8	128K	10.25	8	8	256K	10.66	8	8	512K	10.77
8	8	1024K	11.21	8	8	2048K	10.87	8	8	4096K	13.28	8	8	8192K	12.23
12	2	64K	14.98	12	2	128K	13.45	12	2	256K	13.92	12	2	512K	13.41
12	2	1024K	13.26	12	2	2048K	12.58	12	2	4096K	14.21	12	2	8192K	13.89
12	4	64K	12.75	12	4	128K	11.78	12	4	256K	13.22	12	4	512K	12.08
12	4	1024K	11.92	12	4	2048K	11.15	12	4	4096K	11.04	12	4	8192K	12.46
12	8	64K	11.20	12	8	128K	11.04	12	8	256K	10.90	12	8	512K	12.18
12	8	1024K	10.40	12	8	2048K	11.13	12	8	4096K	10.97	12	8	8192K	11.00

Experiment 2

retrieve raster-avg{RASTER.data} where RASTER.time = TIME

The second experiment is based on the query 3 from the benchmark. The original query is modified to apply the `raster-avg` operator on the whole raster image rather than a part of the image. This modification was done in order to test the efficiency of retrieving whole objects. The query accesses 5 objects having the same time stamp. Since each image occupies 8MB, the total amount of data read is 40MB. The query computes the average of data values across the raster images and returns a single raster image of size 8MB. First the `select` operator is applied to the `RASTER` table which selects rows whose `time` attribute is equal to `TIME`. This result is stored in a temporary table, `T1`. Table `T1` contains 5 rows. Then the `project` operator is applied to table `T1` which projects the `data` attribute and stores the result in a temporary table `T2`. Table `T2` contains 5 rows. The table `T1` is deleted. The `raster-avg` operator is then applied the table `T2` to compute the final result.

We use a two-step approach similar to one described in [SN95] for the `raster-avg` operator. Table 2 gives summary of the results. Variation in the block size keeping other factors same has no effect on the execution time. Increasing the number of I/O nodes improves the performance only marginally. So for practical purposes the query does not exhibit any speedups. This is because the query accesses only $\frac{1}{26}$ of the total number of objects. For a 4 compute node configuration, the best possible distribution of objects is $\{2, 1, 1, 1\}$. For higher number of processors, no parallelism is available and the operator executes serially. Additional overhead is incurred when a node has no objects to process but still needs to convey a null result. The request merging optimization (section 3) ensures that the total number of I/O requests issued is constant across various block sizes keeping the actual I/O time constant. The communications times follow the trends exhibited by the I/O times because the I/O schedule is consists of only one phase.

6 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). 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. We are currently performing more experiments to evaluate the effectiveness of various other I/O optimizations.

References

[CBH+94] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. PASSION: Parallel and Scalable Software for Input-Output. Technical Report SCCS–636, NPAC, Syracuse University, September 1994. Also available as CRPC Technical Report CRPC–TR94483–S.

[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.

[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.

[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.

[SN95] A. Shatdal and J. F. Naughton. Aggregate processing in parallel rdbms. In *Proceedings of the ACM SIGMOD Conference*, May 1995.

[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.