Tuning Large Sorts

Sort Direct Writes, Asynchronous I/O, and Sort Area Size

Oracle Release 7.2 (7.3)

Prabhaker "GP" Gongloor
Basab Maulik
Sameer Patkar

Center of Expertise
Worldwide Customer Support
Oracle Corporation

Copyright © Oracle Corporation 1996

All rights reserved. Printed in the U.S.A.

Research Analysts: Prabhaker “GP” Gongloor

Contributors: Jeff Cohen, Anjo Kolk, Roderick Manalac, Basab Maulik, Sameer Patkar, Brian Quigley

Technical Writer: Brian Quigley

TEST.FRAME: Douglas Chandler, Darryl Presley

TRADEMARKS

Oracle is a registered trademark of Oracle Corporation. Oracle7 is a trademark of Oracle Corporation.

All other products or company names are used for identification purposes only, and may be trademarks of their respective owners.

DOCUMENTATION ERRORS

Questions about the technical content of this report should be directed to the research analyst responsible for this paper.

Please report any documentation errors found in this document to:

Manager of the Center of Expertise
Center of Expertise, Worldwide Customer Support
Oracle Corporation
500 Oracle Parkway
Redwood City, CA 94065
U.S.A.

Fax: (415) 506-7200

Introduction

Audience and Related Documentation

You are an Oracle employee familiar with Oracle7 and are interested in tuning large sort operations. Feature descriptions in this report supplement the Oracle7 Server V7.2 Documentation Set, Part Number A32920.

The Center of Expertise has conducted research focussing on asynchronous reads, a feature used in this project. Findings for the asynchronous reads research are described in the following technical report: Asynchronous Reads, Performance Implications, Oracle Release 7.3.

CAUTIONARY NOTE: Be cautious when interpreting the details in this report with respect to other platforms. Our findings are based on tests conducted in a specially created test environment. Interference from normal application activity is not factored into our findings. Different platforms may show very different outcomes compared to our test results; in particular, if a platform uses SCSI devices with different operating characteristics compared to our test devices.

Overview and Terminology

This report presents results of tests intended to discover some best practices regarding the tuning of large sort operations using release 7.2. Specifically, we are interested in the *sort direct writes* feature introduced in release 7.2, the effect of *asynchronous I/O*, *sort area size* and the utilization of system resources. We also briefly describe the behavior of sort direct writes in release 7.3. The tests for this project were run with COMPATIBLE=7.2 and complement testing by Server Technology development.

Figure 1 on page 4 of (Chapter 2) compares synchronous I/O and asynchronous I/O when a process reads from a file and processes that data.

Synchronous I/O -- when a process must wait for an I/O request to complete before continuing processing.

Asynchronous I/O -- when a process can continue processing after issuing an I/O request. Because processing and I/O can occur at the same time, asynchronous I/O should provide performance gains. The process must still check whether an asynchronous I/O completed successfully. Performance gains for any I/O operation will depend on whether data from one asynchronous I/O is processed before the successful completion of an outstanding asynchronous I/O. Asynchronous I/O is a feature available on most current UNIX systems.

A *conventional read* occurs when a server process reads a data block into the buffer cache. A *conventional write* occurs when DBWR writes a (dirtied) data block from the buffer cache to a datafile.

With release 7.2, an Oracle process may use *direct reads* and *direct writes* for certain operations. A *direct read* occurs when an Oracle server process reads a data block from a data file into private process memory without placing a copy in the buffer cache. A *direct write* occurs when an Oracle process writes a data block to a data file without placing a copy in the buffer cache and without using the services of DBWR. Direct reads and direct writes are said to *bypass* the buffer cache. Transient data blocks, such as those in sort segments, exist for the duration of a single operation and by entering the buffer cache may age out data blocks that are still being used by other processes; that is, transient data blocks cause unwarranted buffer cache contention. The purpose of direct reads and direct writes is to avoid placing transient data blocks in the buffer cache. Disk sort performance should be improved by using direct reads and writes because buffer cache processing is avoided. Furthermore, DBWR is less likely to bottleneck on I/O because part of the I/O burden is removed by the sort server processes.

A disk sort occurs when a sort cannot sort all data in the sort area. In a disk sort, a subset of the data is sorted in the sort area and the resultant sorted data is written as a sort run to disk as temporary segments. The process is repeated until all the sort records are sorted. At this point, the sort runs are merged to obtain a final ordered set by examining the leading record in each of the sort runs and outputting the record with the greatest (or smallest key). In the merge phase, when there are a large number of sort runs, there may not be sufficient memory available to do a single pass m-way merge of m sort runs. In such cases, repeated merges occur on subsets of the sort runs to yield fewer large sized runs. The optimal merge width for these intermediate merges is determined by the multiblock read factor (SORT_READ_FAC), the number of buffers to be merged, and the merge width.

The following description of the SORT_READ_FAC initialization parameter was taken from the source code.

For an external disk sort, say with *m* runs, during the merge phase it is possible that there is insufficient memory to perform a single m way merge. In such cases Oracle will repeatedly merge subsets of the runs. Oracle optimizes the sort multiblock read factor (SORT_READ_FAC) against merge width for intermediate merges.

SORT_READ_FAC (multi-block read factor for sort), is calculated as the time it takes to read a block divided by the marginal per-block read time when reading multiple blocks, as follows:

= (seek time + latency + single block transfer)/(single block transfer)
= (seek time + latency)/(single block transfer) + (single block transfer)/(single block transfer)
= (seek time + latency)/(single block transfer) + 1

The sort_read_fac - 1 = (seek time + latency)/(single block transfer) is the I/O cost without the actual block transfer cost. Setting SORT_READ_FAC requires that you can determine the seek time, latency, and block transfer rate from the disk manufacturers literature.

Sort direct writes is the mechanism where an Oracle process performing a disk sort bypasses the buffer cache and writes sort runs directly to sort segments. This will become clearer at the end of the following description of the conventional disk sort mechanism when creating an index. Figure 1 shows how data blocks move through process address space for a conventional disk sort.

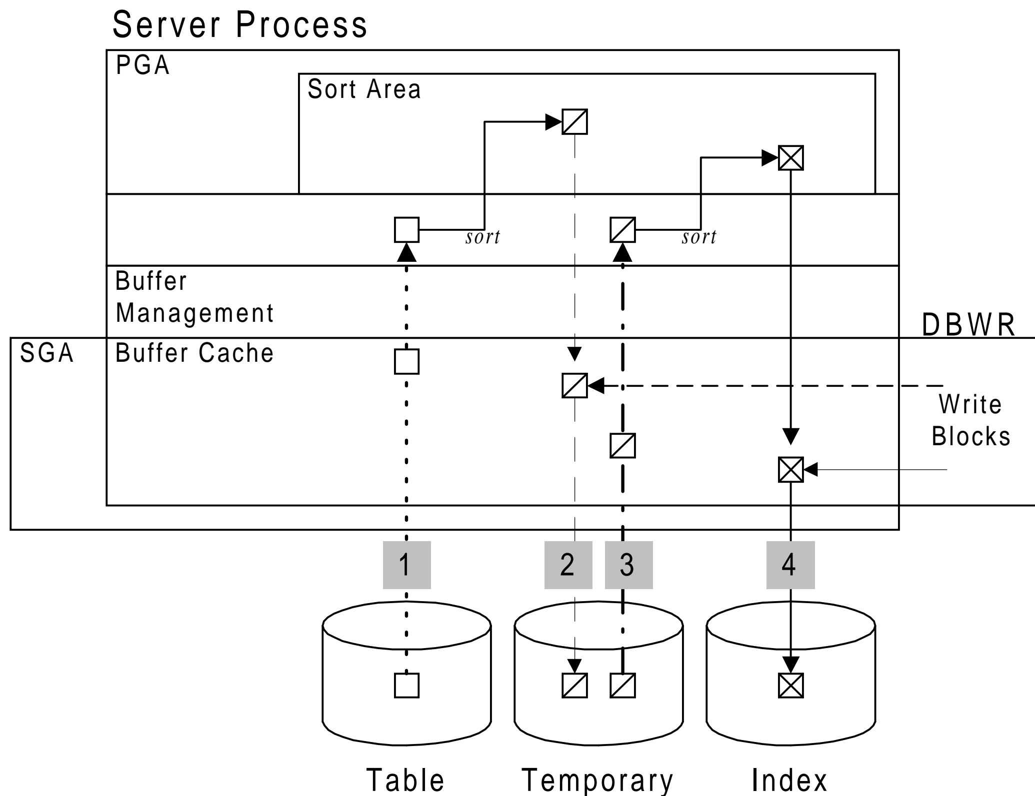


Figure 1: Data Block Movement Through Sort Process Address Space

Sort direct writes works slightly differently than shown in Figure 1. Rather than writing through the buffer cache [2] [4], the sort server process writes the sort data blocks directly to temporary sort segments. The sort direct write feature must be enabled by setting the SORT_DIRECT_WRITES initialization parameter to TRUE.

Introduced in release 7.1.5, the direct read feature is invoked by Oracle only for parallel full table scans. Thus, because parallel create index uses a full table scan, it also invoke direct reads. That is, in our parallel tests, table block reads [1] also bypassed the buffer cache.

Commentary on I/O Activity of Disk Sorts During Create Index

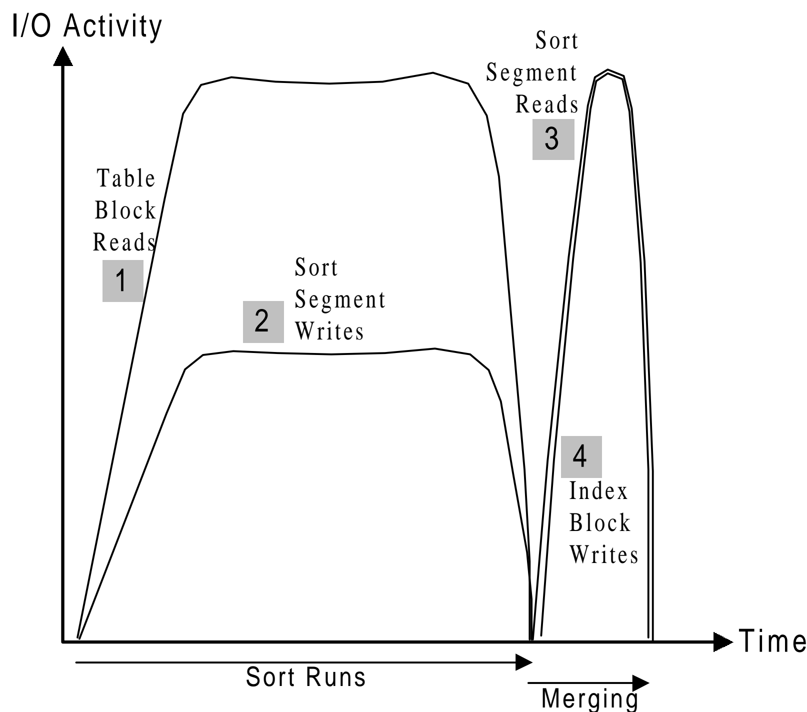


Figure 2: Generic Profile of I/O Activity of External Sort During Index Creation for Large Table

Figure 2 shows a typical profile of I/O activity during a serial index creation that performs a disk sort. Data blocks are read from the table to be indexed [1] and sort runs begin. Sort blocks are created in the sort area and written out to sort segments [2] to free memory for further sort runs. Table block reads [1] finish before the merge phase begins. Sort segment blocks are read [3] back into the sort area and the index blocks are written [4] to the index. Sort segment reads [3] and index block writes [4] first peak and then taper off until the index is completely created.

Difference in Table Reads and Sort Segment Writes

There are more table block reads [1] than sort segment writes [2] because data blocks read contain complete rows whereas the sort blocks written contain just the column data being indexed.

I/O Concurrency

Sort segment writes [2] occur concurrently with table block reads [1] and index block writes [4] occur concurrently with sort segment reads [3]. Obviously, if the temporary space (holding sort segments) and table are on the same disk, then I/O contention will occur. Similarly, I/O contention will occur if the temporary tablespace and index are on the same disk.

Temporary segments, table, and index should be placed in different tablespaces (and preferably in different datafiles) on different disks and each disk on separate disk controllers to optimize the performance of the creation of large indexes.

Note that there is only a small overlap in concurrent I/O on the table reads and index writes. Thus, it is possible to create an index on the same disk with minimal impact on index creation time. This may not be advisable in a processing environment where both table and index will be heavily accessed. However, when there is a shortage of disk space, a free disk could be used for the temporary space and the index could be created in a datafile on the same disk as the table. When index creation is completed, the temporary tablespace could be dropped and the index datafile copied and renamed onto the free disk. For very large indexes, the time to create the index plus the time to copy the index datafile is likely to be less than if the temporary tablespace was on the same disk as the table.

Parallelism

The use of parallel index creation will improve performance of the sort runs phase because scan servers concurrently read the table [1] in parallel and sort servers concurrently write sort segments [2] in parallel.

Table data distribution -- a table is logically partitioned into ROWID ranges at the beginning of a parallel full table scan. Each scan server will begin reading these partitions and pass the column data (to be indexed) to the sort servers. Scan servers will contend with each other on I/O if the table is on a single disk. To optimize the scan phase, I/O interference should be eliminated. This can be achieved by using striping or manually partitioning the table data. Manual partitioning is described below.

Preferably, hardware and/or operating system striping should be used. The degree of parallelism can then be chosen by balancing the I/O throughput and the amount of available memory for the sort servers.

In the case of a non-striped environment, a table should be split across separate datafiles placed on different disks. Assuming sufficient memory is available, then the degree of parallelism for a full table scan should be minimally set to the number of disks. If you want to increase the degree of parallelism, you should increase by a factor of two. However, the intention is to reduce I/O contention. As you double the degree of parallelism, you double the I/O on each disk and double the potential for I/O contention.

Temporary space distribution -- temporary space distribution will cause an I/O contention bottleneck during the sort runs phase if I/O on sort segments is not well distributed across disks. However, temporary space distribution is problematic because sort segments are allocated as requested and may be allocated within the same datafile. This problem is best overcome using hardware and/or operating system striping. That is, a temporary tablespace should be allocated and used specifically for disk sorts and the tablespace datafile(s) should be striped across as many disks as possible.

In the case of a non-striped environment, a temporary tablespace should be allocated and used specifically for disk sort operations. The tablespace should be manually partitioned as described in the next section. Once the tablespace is created, the degree of parallelism when creating the index should be set to m or some multiple of $m > n$.

Manually Partitioning a Non-Striped Tablespace -- a tablespace should be composed of n datafiles to be placed on m disks, where $m < n$ and n is some multiple of m. The first datafile is placed on the first disk, the next datafile on the second disk, and so on until m datafiles are on m disks. Datafile $m+1$ is then placed on the first disk, datafile $m + 2$ is placed on the second disk, and so on. That is, place the datafiles on the disks in a round robin fashion. Each datafile should contain one large extent equal to the initial extent allocated in the tablespace. PCTINCREASE is set to zero and NEXT is set to the size of INITIAL. The intention is to force the allocation of data segments within individual datafiles/disks in a round robin fashion. The total size of the datafiles/tablespace must exceed the maximum expected size of the table or index to be created in the tablespace.

Commentary on Direct Writes and Direct Reads

Whenever possible, our deferred writes design offsets writing data blocks out to disk until a convenient time when the database is not busy reading data blocks (or until a time when the blocks must be written out). Prior to 7.2, all data blocks to be written to disk were first placed in the buffer cache to be handled by DBWR. If a large operation must create and temporarily store data blocks and can avoid using the buffer cache by writing directly to a data file then overall database performance improves because buffer cache contention and DBWR load is reduced. (Blocks remain longer in the buffer cache because the temporary blocks are not placed in the buffer cache, which would force some blocks to be aged out.)

Test Categories

The research for this report focused on tuning disk sorts, as follows:

- SORT_WRITE_BUFFERS, SORT_WRITE_BUFFER_SIZE

  Three initialization parameters were introduced in 7.2 for sort direct writes. Setting `SORT_DIRECT_WRITES` to `TRUE` enables the new feature.

  What values should be chosen for `SORT_WRITE_BUFFERS` and `SORT_WRITE_BUFFER_SIZE` to improve performance of disk sorts?

- serial sorts -- sort direct writes, asynchronous I/O

  What performance improvements are possible for large serial sort operations when using sort direct writes and/or asynchronous I/O?

  How does sort area size influence this performance?

- parallel sorts -- sort direct writes, asynchronous I/O

  What performance improvements are possible for large parallel sort operations when using sort direct writes and/or asynchronous I/O?

  How does sort area size influence this performance?

General Conclusions and Tuning Guidelines

In this section, we summarize our test results with the intention of providing rough guidelines on tuning disk sorts invoked when creating an index. However, the following factors may have a large impact on actual sort performance: 1) Multiple database users contending for database or system resources; 2) System or disk configurations that are substantially different than our test environment.

The generalizations in this section are also loosely applicable to disk sorts invoked by SQL operations other than a create index; for example, a disk sort invoked by a query with an ORDER BY clause.

Contrary to conventional wisdom, an increase in sort area size does *not* always produce performance improvements. Larger sort areas provide performance improvements, *if* the sort can be completed entirely in memory. However, if a sort cannot be completed in memory, then the resulting disk sort is *not* greatly affected by sort area size.

Optimum performance for disk sorts is achieved using a sort area size that does not introduce intermediate sort runs and the following features: parallelism, sort direct writes and (if supported) asynchronous I/O. Enabling any of the the features will always improve the performance of disk sorts regardless of sort area size.

As expected, parallelism had the most noticeable effect on reducing index creation time. Without parallelism, we observed that index creation took ten times longer to complete.

If neither sort direct writes nor asynchronous I/O are enabled, then enabling just one of these features will substantially improve parallel and serial disk sort performance. Our tests showed index creation time being reduced by more than two thirds.

If asynchronous I/O is already enabled, then enabling sort direct writes provides respectable performance improvements for parallel and serial disk sorts. However, be cautious when using the 7.2 implementation of this feature because it is possible to overload your system. The release 7.3 implementation provides some relief from this problem. (See “Development Commentary on Sort Direct Writes” in the following section.)

If you enable sort direct writes, then increasing the number and/or size of the sort direct write buffers (by increasing the values of SORT_WRITE_BUFFERS and/or SORT_WRITE_BUFFER_SIZE) will only provide minor additional gains in performance of disk sorts. This is true for parallel and serial disk sorts regardless of asynchronous I/O being enabled or disabled.

Development Commentary on Sort Direct Writes

Careful consideration is required regarding the use of the sort direct write feature. This is especially true in environments with a large number of users or when the parallel query option is used. If the sort direct writes feature is used, then a large number of processes could quickly allocate all memory in a system.

The minimum sort direct write buffer configuration is the product of the minimum values for SORT_WRITE_BUFFERS times SORT_WRITE_BUFFER_SIZE, or 2*32, giving 64K. The maximum buffer configuration is 8*64K, giving 512K.

***Release 7.2* --** Using sort direct writes causes each Oracle process initiating a disk sort to allocate SORT_WRITE_BUFFERS * SORT_WRITE_BUFFER_SIZE bytes of memory *in addition* to memory already allocated for the sort area. Be sure that your operating system has enough free memory available to accommodate the extra memory to be allocated. One way to avoid increasing memory usage is to decrease the sort area by the amount of memory allocated for sort direct writes. As a general guideline, the total memory allocated for direct write buffers should be less than one tenth of the memory allocated for the sort area. If the minimum configuration is less than one tenth of the sort area, then you should not trade sort area for direct write buffers.

Release 7.3 -- For release 7.3, SORT_DIRECT_WRITES=TRUE operates in the same fashion as in release 7.2. Introduced in release 7.3, SORT_DIRECT_WRITES=AUTO determines whether sort direct writes will be used or not for a disk sort based on the one tenth rule (described above for Release 7.2). Sort direct writes will *not* be used if the minimum sort direct write buffer configuration is greater than one tenth of the sort area size. That is, sort direct writes will not be used if the sort area is smaller than 640K. The major benefit in using the auto mode feature is that memory for the sort direct writes feature is allocated from the sort area. That is, the memory is not allocated in addition to the sort area such that overall sort performance is not severely impacted.

Future Tests

The results presented in this paper may be used as a baseline for testing the effects of:

- buffer cache contention and system workload -- sort direct writes bypass the buffer cache, thus reducing buffer cache contention with other user processes. How does this effect overall system and database performance when there is a mix of users? That is, when some users are using direct sort writes and some users are accessing the database through the buffer cache.
- sort area size -- how does one estimate the best sort area size for parallel and serial queries?
- disk affinity -- introduced in 7.3, the disk affinity mechanism assigns query server processes to read from specific disks. How does disk affinity influence disk sort performance?

The Testbed

Methodology

Tests were conducted in a close to ideal environment designed to minimize contention or interference from housekeeping activities by Oracle or the operating system. For example, database files were striped across disk to minimize I/O contention, direct reads were used to avoid buffer cache management overhead, and the unrecoverable option was used when creating the index to avoid redo creation overhead.

Although only one type of sort operation was used in all the tests, the results are comparable with any sort operation in the same environment requiring disk sorts. In each test, the same large sort operation was invoked by creating the same non-unique index for the same large table. For the first two test suites, a degree of parallelism of 16 was used. For the last test suite, no parallelism was used.

Tests either used or did not use sort direct writes and/or asynchronous writes. Note that some platforms do not support asynchronous I/O. The `SORT_DIRECT_WRITES` and `ASYNC_IO` initialization parameters must be set to `TRUE` to enable these features.

utlbstat.sql and *utlestat.sql* were used to monitor the database during tests. The unix *ps* command was also used to gather statistics. Analysis of the gathered statistics showed that there were no anomalies in CPU usage. Thus, in this report, only the statistics for elapsed time (from *sar -p*) and physical blocks read and written for each tablespace are of interest.

The TEST.FRAME environment provided control over the collection of statistics. TEST.FRAME is an integrated set of UNIX scripts that automates the control of child processes tracking the status of a test suite, recording environment settings, collecting resource usage statistics from the operating system, and collecting database statistics. Statistics were collected every 30 seconds with the assumption that the overhead in gathering statistics is negligible.

Hardware

All tests were performed on a Sequent Symmetry 5000 SE60. There are three nodes in the cluster, but our tests were run in exclusive mode on one node. The node ran Dynix/PTX 2.1.6 and UNIX SVR3.2. Each node of the cluster has 16 CPUs and 1.6 gigabytes of RAM. As described in the following section, tablespace database files were striped across disk controllers, ports, and channels for optimum performance. Stripe width was 32K.

Oracle Configuration

Oracle Release

Release 7.3.1 was installed in the test environment. Although 7.3.1 was used in the tests, the results in this paper should be the same as if 7.2 was installed in our test environment. (SORT_DIRECT_WRITES=AUTO was not used. SORT_DIRECT_WRITES=TRUE was used, which invokes 7.2 behavior.)

Data Description

The table used in our tests was populated with data characteristic of DSS applications. A description of the generated data is given in Appendix B on page 35. Table creation was initiated using storage parameters: `INITIAL 3500K`, `NEXT 3500K`, and `PCTINCREASE 0`. After creation, the test table had the following characteristics:

Table Size:	450 MB (approximately)
Rows:	10 million (approximately)
Average Row Length:	42 bytes

Index Description

The index created on the test table in each test resulted in an index of size 240 MB. Each test initiated the following statement:

```
CREATE INDEX store_prod13 ON sales_desc13(store_id, prod_id)
    TABLESPACE indexes13
    STORAGE (INITIAL 3500K NEXT 3500K PCTINCREASE 0)
    UNRECOVERABLE
    PARALLEL (DEGREE n)
```

where *n* was set to 1 or 16. A value of 1 specifies that parallelism is not used. Parallelism was used in the first two test suites presented in this report, but not the last suite of tests.

Tablespaces

The tablespace containing the test table was created on one striped virtual raw disk file. The index tablespace consisted of 3 raw files of 500 MB totalling 1.5 GB. The files for index and table tablespaces were striped across 2 controllers, 4 physical disks and 4 channels.

The tablespace used to hold the created index consisted of 6 raw files of 500 MB totalling 3 GB. The virtual disks were striped across 2 controllers, 4 physical disks, and 4 channels.

The temporary tablespace used during the index creation tests was created on 6 raw files of 500 MB totalling 3 GB. The virtual disks were striped across 2 controllers, 4 physical disks and 4 channels. This tablespace was created using the TEMPORARY clause in the CREATE TABLESPACE command.

Initialization Parameters

`DB_BLOCK_SIZE` was set to 4K for database creation. The initialization file is listed in Appendix A on page 34.

Note that the `_DB_BLOCK_WRITE_BATCH` initialization parameter was used. Underscore parameters are not documented and should not be used. However, it was known at the time of testing that the behavior of this parameter would not change and that it would be externalized for general availability to customers in the subsequent release.

Oracle Files

Other database files, such as redo log files and control files, were placed on pre-allocated raw volumes. Scripts and test programs were kept separate on the user file system (UFS).

Understanding the Parallel and Index Creation Results

This section contains background "explanations" that are referred to the I/O profiles descriptions for the parallel and serial index creation test results.

Explanation 1: Suppose there are 4 direct sort write buffers, each of 64 K size. The server process can return to build the next sort run with 4 X 64 K = 256K of pending IO provided asynchronous I/O is used. If the sort area size is 256K, the process can return immediately to build the next sort run (assuming the actual memory to memory copy operation will take negliglible time). If sort area size is 512K, then the first 256K chunk is copied into the the direct sort write buffers. For the next 256K chunk, the process has to check that the first chunk has been written out before copying the second chunk from sort area to direct sort buffers. This involves some extra code path and possible blocking. The larger the sort area size, the more prevalent the second phase; that is, we go through something similar *n* times rather than once. Thus in general we should expect larger sort buffers to perform relatively poorly with the total size of the sort direct write buffer held constant. Conversely, the larger the size of the sort buffer is with respect to the sort area, the more effective is the buffering.

Explanation 2: With a decrease in sort area size, the size of sort runs written to temp tablespace decrease in size, but there will be a larger number of them (provided we are sorting on the same data). It is possible that there are more sort runs than can be merged in a single pass. In this case, an initial merge pass reads the sort runs and writes out larger merged runs. These are merged again to complete the merge phase. These intermediate merges result in extra run(s) of reads and writes on the temp tablespace. The value of sort area size below which intermediate merges are induced is a critical value.The resulting change in sort time with small changes in sort area size values makes the use of the word critical appropriate.

Explanation 3: Consider a sort area of size *S* and the amount of data to be sorted of size *D*. Then:

Number of sort runs = ceil[D/S]

Time for each sort run = K_1 S log(S) where K_1 is a suitable constant

If we consider the total time to generate all sort runs (without considering time for I/O), then:

T = ceil[D/S] K_1 S log(S)

= K_1 D log(S) (approximately)

This means that the **time to generate sort runs** increases logarithmically with the size of the sort area size *S*. (Assuming *D* is kept constant.) As a result, we should see lower throughput in temp writes and table reads with increased sort area size.

However, the merge phase is more efficient with a larger sort area. this is because a larger sort area size results in fewer sort runs to be merged. Furthermore, the larger the sort area, the smaller the chance that intermediate merge runs will be needed, if at all. Thus, reducing the time to generate sort runs by reducing sort area size might marginally offset the advantage of an efficient merge phase with a larger sort area size. These two opposing trends result in the lack of sensitivity of disk sorts to sort area size.

Explanation 4: In the case when sort direct writes is set to false, the write to disk is scheduled through the buffer cache and handled by DBWR. Here the two factors that determine sort write performance are: (1) size of the buffer cache (2) DBWR efficiency. Thus, DBWR may bottleneck and eventually the sort process waits on finding free buffers in the cache to effect the write to disk. In our experience, we find that DBWR is able to keep up with very rapid dirtying of the cache when asynchronous I/O is enabled.

Explanation 5: Whenever the parallel query option is used several additional factors come into play which could mask or override weak dependency on the feature under study. These include (1) skew or a large disparity in the work loads of the slave processes (2) scheduling issues are independent for each slave process (3) the bandwidth of the I/O subsystem may be relevant if the various parallel processes do I/O in a coherent manner (4) contention for latches and enqueues more likely (5) there is a large IPC overhead and state overhead between the communicating parallel processes. The various combinations of these factors will vary from run to run in a random manner and make interpretation of trends more difficult.

SORT_WRITE_BUFFERS, SORT_WRITE_BUFFER_SIZE

Test Purpose: What values should be chosen for `SORT_WRITE_BUFFERS` and `SORT_WRITE_BUFFER_SIZE` to improve performance of disk sorts?

Conclusions: As shown in the next section, enabling sort direct writes produced a substantial performance improvement (when asynchronous I/O *was not* enabled) or a moderate performance improvement (when asynchronous I/O *was* enabled). We consider a 65.6 percent reduction in index creation time a substantial improvement and a 9.2 percent reduction a moderate improvement. This is consistent with Explanation 4 on page 11. Without sort direct writes and asynchronous I/O, DBWR is the bottleneck.

Similar order of magnitude performance improvements did not occur by increasing the number or size of sort direct writes buffers. However, additional performance gains of a respectable magnitude are possible, but only at the cost of more memory. Note that in release 7.2 this extra memory *is in addition* to the memory allocated for the sort area. (See section "Future Tests" for a brief description on 7.3 rules on allocating memory for sort direct writes.)

Compared to the default number of buffers and buffer size, an additional reduction in index creation time, 6.9 percent, was observed by using 8 sort buffers of size 64 KB, but at an additional cost of 7 MB for all 16 parallel sort process during index creation [16 * ((8 * 64K)) - 64K]. However, a similar additional reduction of 6.2 percent was observed using significantly less sort area memory. That is, 4 sort buffers of size 32KB; an additional cost of only 1 MB [16 * ((4 * 32K)) - 64K)]. You must determine whether additional performance gains are worthwhile with respect to memory costs.

You should be cautious about using sort direct writes in release 7.2 because memory is allocated in *addition* to the sort area memory. As can be seen in Table 1, increasing the number of sort buffers and/or sort buffer size for sort direct writes has a significant impact on memory requirements. In particular, you must take this into consideration when a sort operation uses parallelism because the number of sort server processes will match the degree of parallelism.

SORT _WRITE _BUFFERS	SORT _WRITE _BUFFER _SIZE	Number of Sort Processes Session Memory Needed Per Sort Process				
		1	4	8	16	32
2	32	64K	256K	512K	1024K	2048K
2	64	128K	512K	1024K	2048K	4096K
4	32	128K	512K	1024K	2048K	4096K
4	64	256K	1024K	2048K	4096K	8192K
8	32	256K	1024K	2048K	4096K	8192K
8	64	512K	2048K	4096K	8192K	16384K

Table 1: Session Memory Needed Per Sort Process when Using Sort Direct Writes

Test Algorithm: The following algorithm was used in this test suite:

```
SORT_AREA_SIZE = 1M
foreach SORT_WRITE_BUFFERS in (2,4,8)
   foreach SORT_WRITE_BUFFER_SIZE in (32K, 64K)
         create non-unique index on 450M table
         with degree of parallelism set to 16
```

Test Results: All tests in this section used a 1 MB sort area size.

Table 2 shows the elapsed times in our test for various settings of `SORT_WRITE_BUFFERS` and `SORT_WRITE_BUFFER_SIZE`.

	SORT_WRITE_BUFFER_SIZE Index Creation Time	
SORT_WRITE_BUFFERS	32K	64K
2	4:51	4:36
4	4:33	4:32
8	4:33	4:31

Table 2: Elapsed time in Mins/Secs

In this section, we use a baseline for comparison. The baseline is the test run using the default settings of SORT_WRITE_BUFFERS=2 and SORT_WRITE_BUFFER_SIZE=32. As shown in the next section, enabling sort direct writes reduces sort time by as much as one third. Thus, when considering any performance improvement in this section, the improvement should be considered an *additional* reduction to the reduction caused by enabling the feature. This is an important consideration given the large amounts of session memory needed to provide the additional performance gain. For example, say that enabling sort direct writes reduces index creation time by 9.2 percent using a certain number of sort buffers of a certain size; if, in this section, we see that doubling the number and size of the buffers produces a 5 percent reduction, then the overall reduction is 8.74 percent [9.2 - (9.2 * 0.05)].

Figure 3 expands on the results in Table 2 on page 13. Each column shows the reduction in index creation time as a percentage when number and size of sort buffers are changed from the default settings. The boxes above the columns show what the reduction in index creation time in seconds. The plotted line shows the total session memory used by all sort servers used in the index creation. For example, the first column shows that the index creation with 2 buffers of size 64K needed an extra 1 MB to complete 15 seconds before the baseline giving a 5.2 percent additional gain in performance.

The baseline run took 4:51 minutes/seconds and the best run took 4:31 minutes/seconds. The difference of seconds 20 is a 6.9 percent gain. However, to achieve this additional gain costs 7 MB [16 * (512K - 64K)] more session memory than the baseline.

The first two columns show that for the same memory cost using 4 buffers of size 32K is better than using 2 buffers of size 64K. However, we cannot generalize that incurring the same memory cost by using more buffers of smaller size will provide better performance. For example, note how 8 buffers of size 32K performs marginally worse than 4 buffers of size 64K.

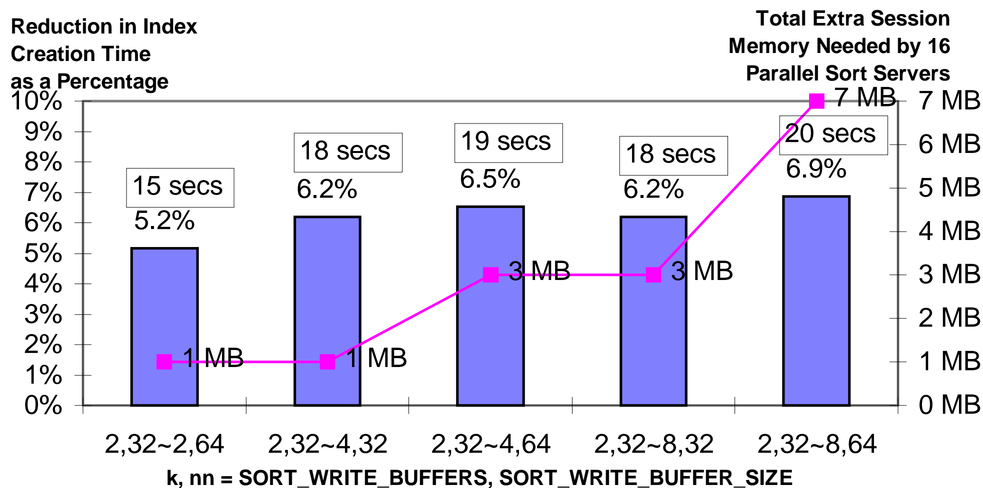


Figure 3: Total Memory Needed for Parallel Sort Servers

Figure 4 shows (more explicitly than Figure 3) how varying the number and size of buffers influences index creation time. Each column shows the elapsed time in minutes (using decimal notation) to create the index. The shadowed boxes between columns shows the effective percentage reduction in creation time due to varying the number of buffers or buffer size. For example, the first two columns show that a run with 4 buffers of size 32K will complete in 6.2 percent less time than a run with 8 buffers of size 32K.

The first eight columns show how increasing the number of sort direct write buffers influence performance. That is, doubling the number of buffers does increase performance, albeit very marginally.

The next six columns show how increasing buffer size influences performance. The first two of these columns shows a moderate improvement, but only marginal improvements occur when more than two buffers are used.

The four remaining columns confirm a conclusion noted for Figure 3. That is, it is not possible to generalize that more buffers of smaller size will provide better performance.

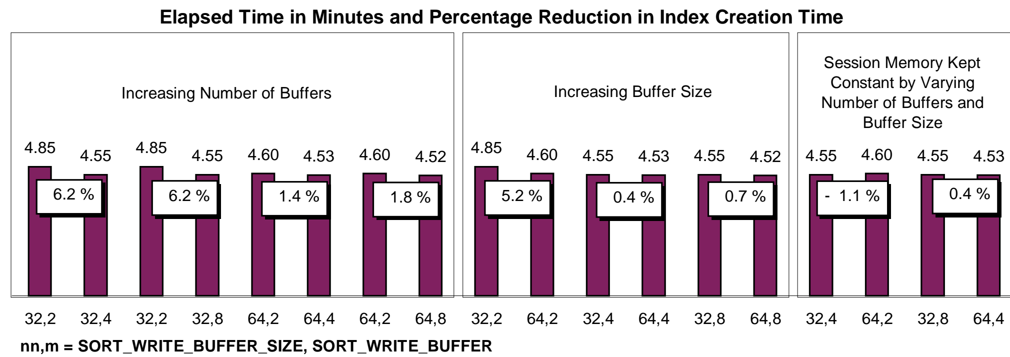


Figure 4: Reduction in Index Creation Times -- Varying SORT_WRITE_BUFFER_SIZE and SORT_WRITE_BUFFERS

Serial Index Creation Sort

Test Purpose: What performance improvements are possible for large serial sort operations when using sort direct writes and/or asynchronous I/O?

What effect does sort area size have on these results?

Conclusions: Without parallelism, we observed that the index creation time took atleast three times as long to complete. (See next section for parallel test timings.)

Performance gains are not nearly as substantial as when using parallel index creation.

The best performance gain, 11.4 percent, occurred when both asynchronous I/O and sort direct writes were enabled. Also notable, is that in the next section both asynchronous I/O and sort direct writes provided performance gains of a similar order of magnitude. This is not the case with serial sorts; sort direct writes performs about half as well as using asynchronous I/O.

In this test suite, for an environment *not* using asynchronous I/O, increasing sort area size caused minor improvements in sort performance. This is in contrast to the general case.

It is worth noting that in this serial case, DBWR is able to keep up with flushing the dirtied cache when sort direct writes and asynchronous I/O are both disabled.

Cautionary Note: Parallel index creation uses multiple sort areas, one per parallel process. This is not true for serial index creation, which uses only one sort area. This should be considered when comparing the test results for serial and parallel index creation because different amounts of memory is used for sorting.

Test Algorithm: The algorithm used in this test suite was the same as in the previous test suite.

Test Results: Table 3 shows the results of this test suite. They are also presented graphically in Fig.5 & Fig.6.

		SORT_DIRECT_WRITES	
ASYNC_IO	SORT_AREA_SIZE	FALSE	TRUE
FALSE	1M	45:50	43:51
	2M	44:20	43:34
	4M	42:24	43:20
TRUE	1M	42:17	40:37
	2M	42:10	40:46
	4M	41:20	41:07

Table 3: Elapsed Time in Mins/Secs for Large Serial Sort Operation

We will discuss the trends in index creation times for the different cases in the following section.

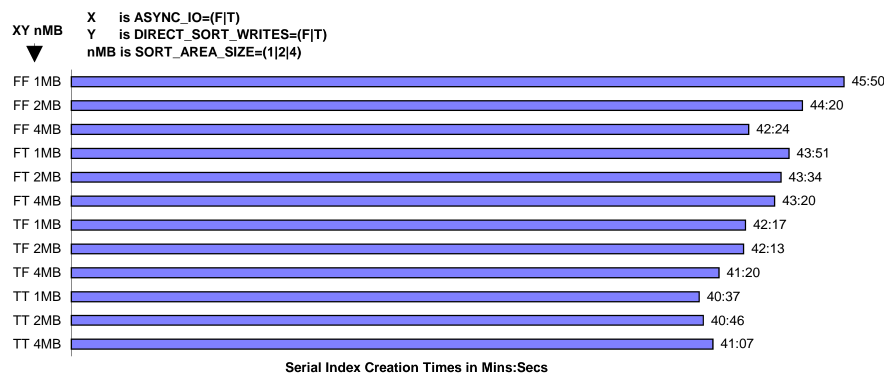


Figure 5: Serial Index Creation Times

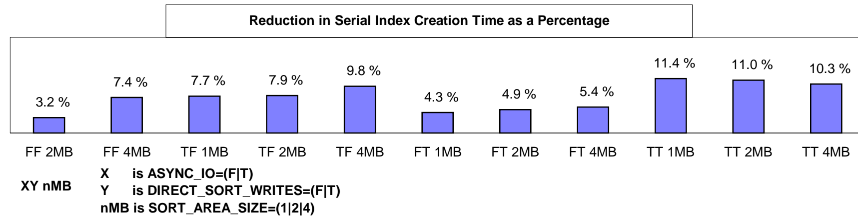


Figure 6: Reduction in Serial Index Creation Time

Serial I/O Profile for Async IO Disabled and Sort Direct Writes Disabled

Figure 7 shows the I/O throughput profiles for three serial index creation tests when asynchronous I/O was disabled and sort direct writes was disabled. Graph 1 is for a test run using a 1 MB sort area, Graph 2 for 2 MB, and Graph 3 for 4 MB.

(The results in this section are used as a baseline for comparison in the next three sections.)

Graph 1 shows intermediate merge run activity after the table reads because the small sort area was unable to merge all sort runs in a single pass in the merge phase. (See Explanation 2 on page 11.) With synchronous I/O the sort process completes writing a sort run before generating the next sort run. The buffer cache is large enough to cache the entire sort run. DBWR then performs a burst of I/O to write the sort segments to disk. Thus the DBWR workload is generated in chunks of sort area size and accounts for the "spiky" write rate to the temporary tablespace. Because DB_BLOCK_WRITE_BATCH was set to the high value of 512, DBWR was able to keep up with the dirtying of buffer cache.

Graph 2 shows fewer intermediate merge runs because of a larger sort area (see Explanation 2 on page 11). The "spiky" temp writes throughput is more exaggerated than in Graph 1 because of the larger sort area. The larger sort area causes a lower throughput for table reads and temp writes than seen in Graph 1; that is, it takes longer to complete the same amount of sort processing and intermediate I/O (see Explanation 3.) The buffer cache & DBWR keep up with the rate of dirtying of buffers by the sort process. The reduction in time needed for the intermediate merge runs offsets the lower sort run generation rate and results in a marginal overall improvement over the results shown in Graph 1.

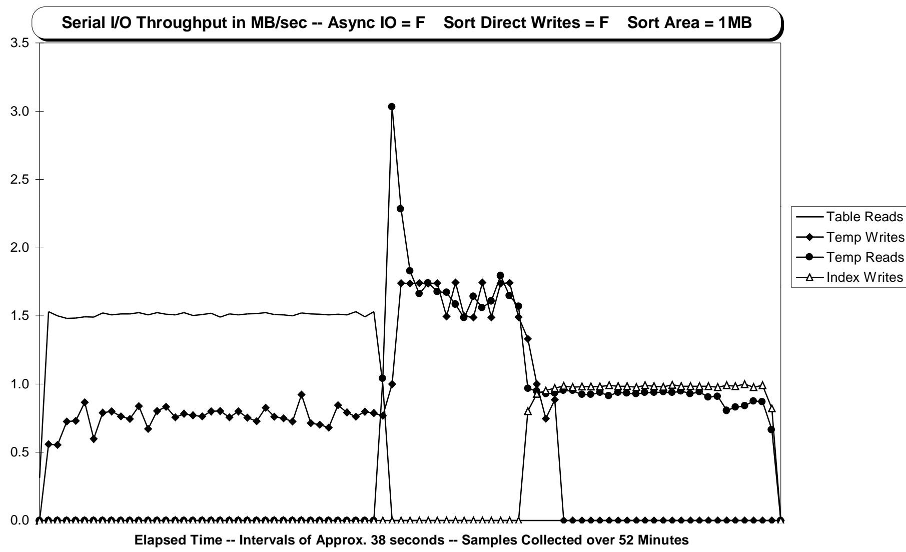
Graph 3 shows no intermediate merge runs because the 4 MB sort area generates fewer sort runs which can subsequently be merged in a single pass.

There is a marginal reduction in elapsed run time with larger sort areas because the merge phase can be completed in one pass, that is, the expense of intermediate merge runs is avoided. This offsets the lower throughput in table reads and temp writes (Explanation 3).

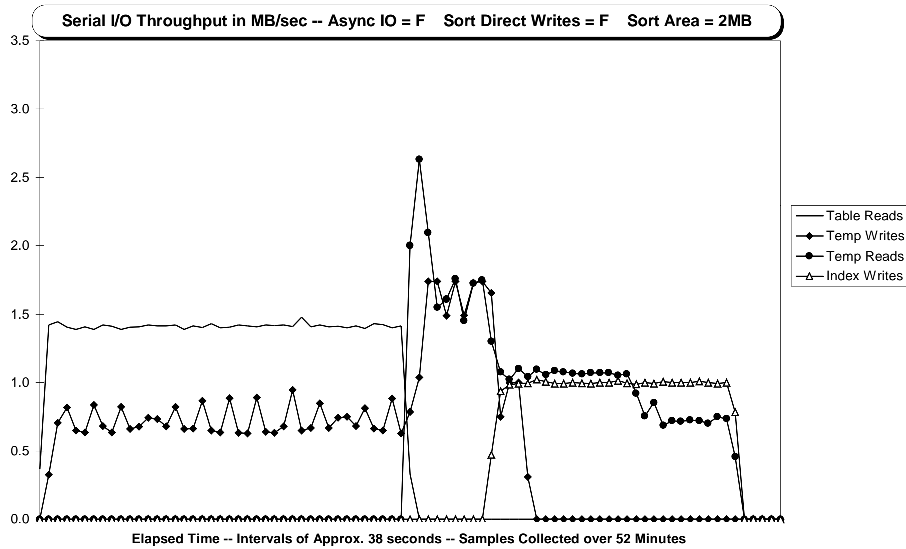
Graph 2 shows a reduction in run time as compared to Graph 1 because fewer intermediate merge runs were needed because of the larger sort area.

Thus, larger sort areas improve run time because intermediate merge runs are avoided. The time saved in avoiding intermediate merge runs is greater than the increase in time due to reduced table reads and temp write throughput.

Graph 1



Graph 2



Graph 3

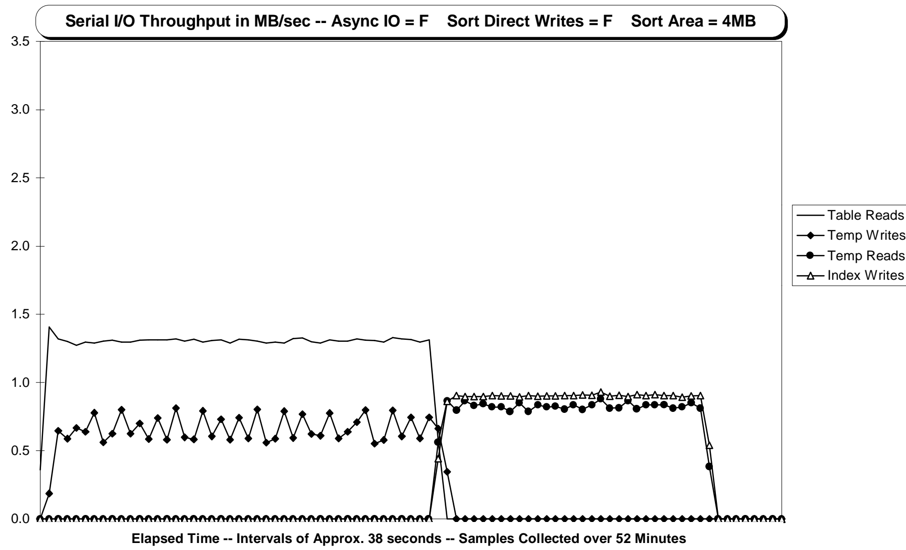


Figure 7: Serial I/O Throughput in MB/sec -- Async IO=FALSE -- Sort Direct Writes=FALSE

Serial I/O Profile for Async IO Disabled and Sort Direct Writes Enabled

Figure 8 shows the I/O throughput profiles for three serial index creation tests when asynchronous I/O was disabled and sort direct writes was enabled. Graph 1 is for a test run using a 1 MB sort area, Graph 2 for 2 MB, and Graph 3 for 4 MB.

(Results in this section will be compared with the baseline results described on page 16.)

The graphs show that when the buffer cache is avoided by using the sort direct write feature, the table reads and temp writes throughput is almost the same as for the baseline. (The server process still has to wait for synchronous I/O to complete before it can begin new sort runs.)

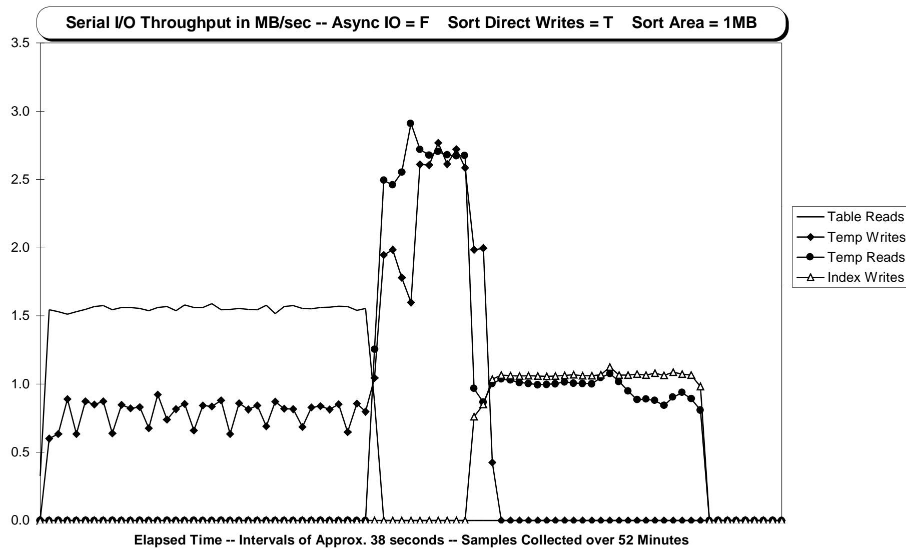
There is a marginal reduction in run time as sort area size increases avoids the time cost of intermediate merge runs. The saved time is greater than the loss in time due to a lower table reads and temp write throughput because of the larger sort area. (Explanations 2 & 3)

If we do a pairwise comparison between the baseline case and this one we notice the following:

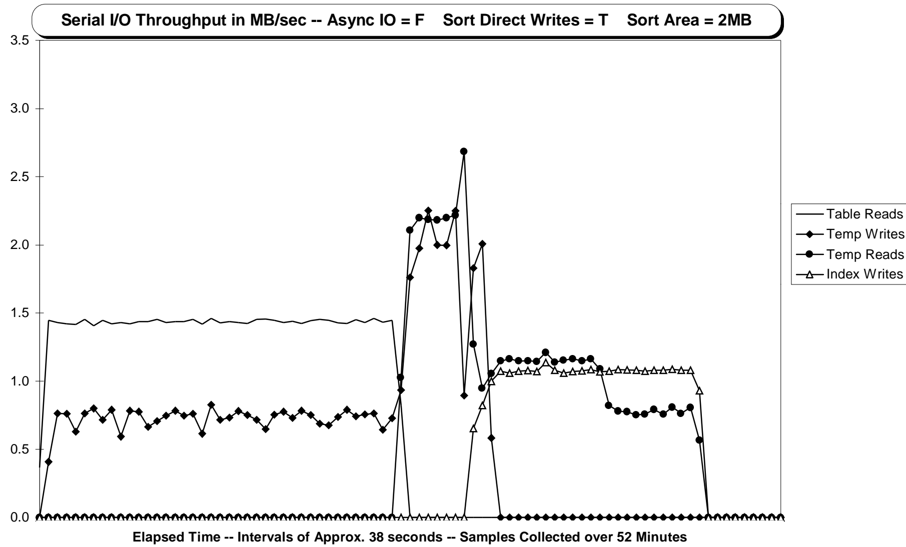
(i) For sort area size of 1M and 2M total elapsed time is less using direct sort writes. This reflects the extra code path and time to acquire latches when using a shared data structure such as the buffer cache, in the baseline test.

(ii) The improvements in total elapsed time is more pronounced for the baseline compared to this case. This is because of two factors (a) Instead of the sort process blocking on I/O it has only to copy into the buffer cache and let DBWR complete the I/O. (b) With larger sort area the buffering by the buffer cache is much more effective (larger cache) than the direct sort buffers.

Graph 1



Graph 2



Graph 3

Serial I/O Throughput in MB/sec -- Async IO = F Sort Direct Writes = T Sort Area = 4MB

3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0

Table Reads
Temp Writes
Temp Reads
Index Writes

Elapsed Time -- Intervals of Approx. 38 seconds -- Samples Collected over 52 Minutes

Figure 8: Serial I/O Throughput in MB/sec -- Async IO=FALSE -- Sort Direct Writes=TRUE

Serial I/O Profile for Async IO Enabled and Sort Direct Writes Disabled

Figure 9 shows the I/O throughput profiles for three serial index creation tests when asynchronous I/O was enabled and sort direct writes was disabled. Graph 1 is for a test run using a 1 MB sort area, Graph 2 for 2 MB, and Graph 3 for 4 MB.

(Results in this section will be compared with the baseline results described on page 16.)

The buffer cache is large enough to buffer the temp segments and DBWR can cope with flushing dirtied blocks because of asynchronous I/O.

In comparing Fig. 9 with Fig. 7 (the baseline case) the important points are:

(a) The asynchronous I/O causes the DBWR write rate to be quite smooth compared to the corresponding case without asynchronous I/O.

(b) The write rate with asynchronous I/O is close to the average write rate of DBWR without asynchronous I/O i.e. midway between the peaks and valleys.

(c) This case has the Oracle 7.3 Asynchronous Read Ahead feature automatically enabled when asynchronous I/O is enabled at the OS level. Experiments on this new feature have indicated its ability to move a process execution closer to a CPU bounded regime. We believe that in the final index creation phase the read rate of the sort segment is higher because of this (compare Temp Reads with Index Writes in the graphs across tests with and without Asynchronous I/O.)

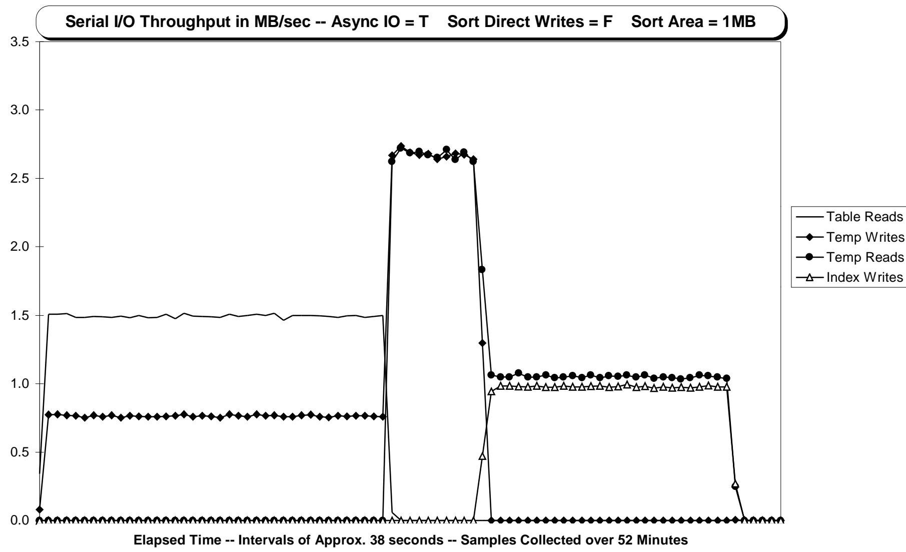
(d) The biggest source of the overall time improvement is due to a marked and dramatic improvement in Temp tablespace I/O throughputs for the intermediate merge phase for the smaller sort area sizes of 1M and 2M (which are below critical.) This phase see's a sustained I/O throughput rates of 2.7MB/sec and 2.5MB/sec. Compare this with a uneven or spiky output average rates of 1.7MB/sec and 1.7MB/sec. The unevenness has been noted and explained before.

(e) We are not certain why the I/O is smoothed when asynchronous I/O is enabled. The producer process produces at the same spiky rate and some smaller amount of unevenness does show on the temp writes as the sort area size increases.

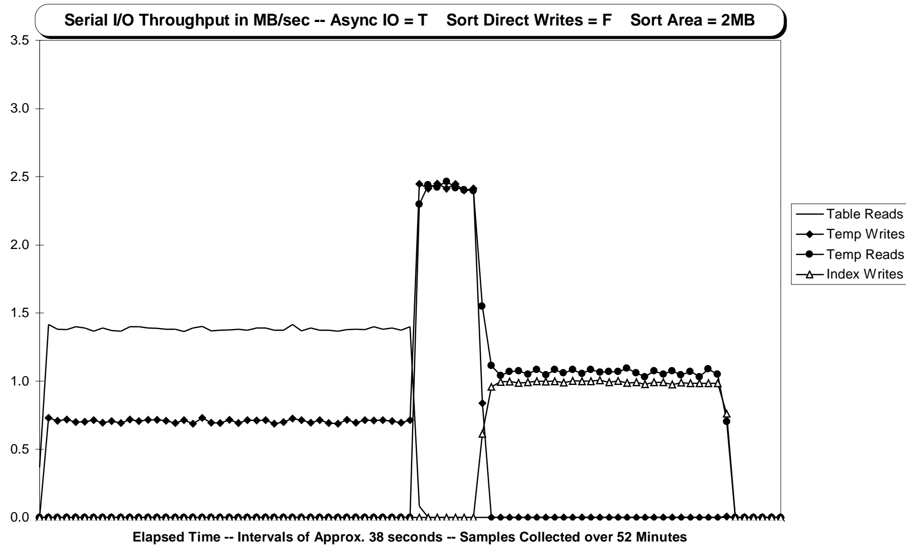
(f) We are also puzzled as to why across all cases the reading of datablocks does not reflect a sort area size chunkiness.

In summary the index creation time for sort area size of 1M and 2M show marked improvements in completion timings due to the significantly higher throughput with asynchronous I/O in the intermediate merge phase. The times are relatively insensitive to sort area size because of explanation 3. The difference between the 1M and 2M cases and the 4M case is due to explanation 2 i.e. 1M and 2M are below the critical sort area size.

Graph 1



Graph 2



Graph 3

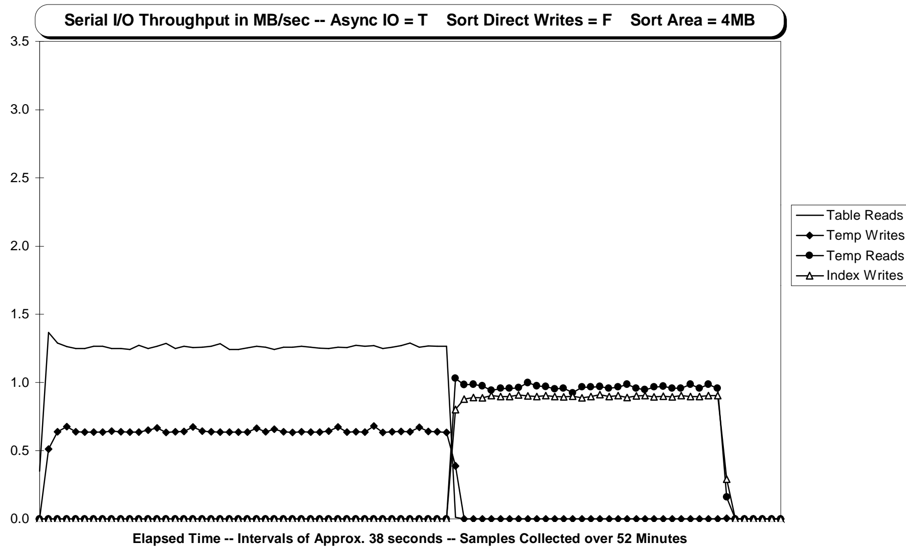


Figure 9: Serial I/O Throughput in MB/sec -- Async IO=TRUE -- Sort Direct Writes=FALSE

Serial I/O Profile for Async IO Enabled and Sort Direct Writes Enabled

Figure 10 shows the I/O throughput profiles for three serial index creation tests when asynchronous I/O was enabled and sort direct writes was enabled. Graph 1 is for a test run using a 1 MB sort area, Graph 2 for 2 MB, and Graph 3 for 4 MB.

(Results in this section will be compared with the previous results described on page 20, Fig. 9. This allows us to see the effect of Direct Sort writes versus Buffer Cache writes with asynchronous I/O.)

In Graph 1, note how temp reads and writes peak after table reads and sustain that peak for several minutes. Thereafter, temp writes drop to zero and temp reads and index writes maintain a similar sustained throughput. The following conclusions can be drawn for Graph 1, as follows:

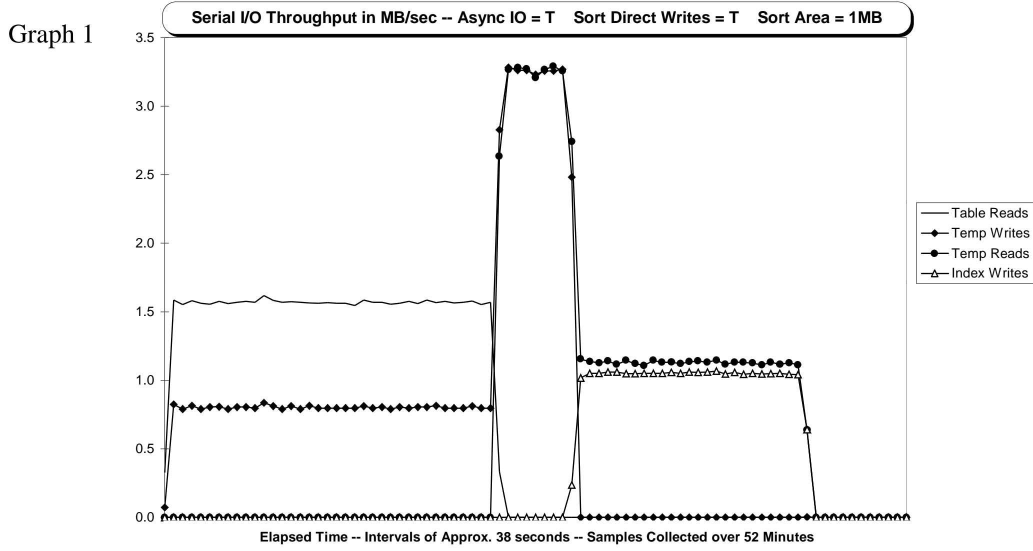
- the sudden spike in temp reads and writes (after table reads) correspond to intermediate merge runs required because the sort area (1 MB in this test), was too small to allow a single merge pass
- index writes begin only when the merge phase begins and only after intermediate merge runs complete
- the sustained spike in temp reads and temp writes on the temporary tablespace (after the table reads) occurs because the reads and writes are occurring simultaneously. Thus, the temp tablespace should be well striped over multiple devices to improve performance of the simultaneous reads and writes during intermediate merge runs.
- however, a larger sort area (4 MB in Graph 3), no intermediate merge runs are required

Graph 2 is very similar to Graph 1. However, the spiky temp writes is somewhat more pronounced than in Graph 1. The intermediate merge runs take less time (than in Graph 1) and temp writes throughput is lower because of the larger sort area. (Explanation 3.) Furthermore, the direct sort write buffering is less effective for larger sort areas. (see Explanation 1.)

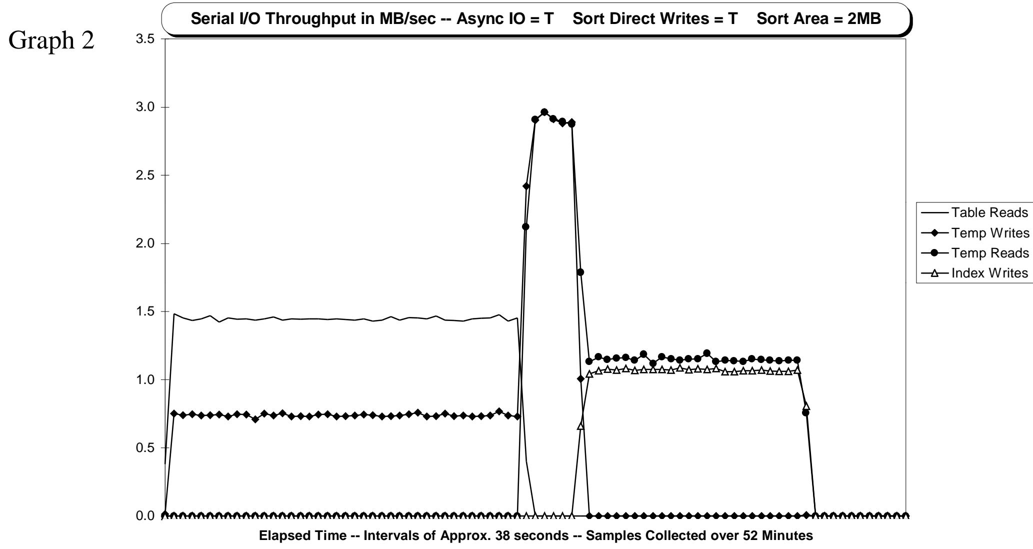
Graph 3 shows that the sort area is large enough to service the merge phase without incurring intermediate merge runs. Temp writes are more spiky than in the 1 MB and 2 MB sort area tests because of the larger sort area. In Graph 3, we see that index writes start at the beginning of the merge phase. Index writes throughput is close to temp read throughput. Note again the effect of Asynchronous Read Ahead by comparing with the Temp Read rate for the merge with Fig. 8 (case DSW=T & Asynchronous I/O= F). The throughput of the temp reads in the merge phase is dependent on the number of sort runs. (Explanation 3).

This is the first case for the serial set where we see that the decreasing effectiveness of larger sort area sizes. The slower generation of sort runs (Explanation 3) and the lower effectiveness of the buffering (Explanation 1) dominate the increased merge efficiency.

Graph 1



Graph 2



Graph 3

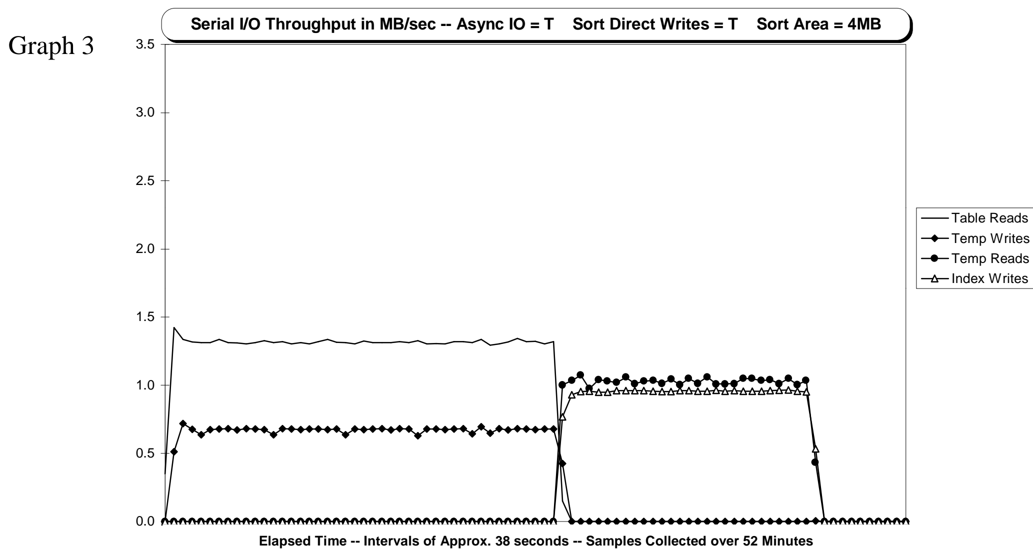


Figure 10: Serial I/O Throughput in MB/sec -- Async IO=TRUE -- Sort Direct Writes=TRUE

Parallel Index Creation Sort

Test Purpose: What performance improvements are possible for large parallel sort operations when using sort direct writes and/or asynchronous I/O?

What effect does sort area size have on these results?

Conclusion: Enabling sort direct writes and/or asynchronous I/O (if supported) will always improve sort performance.

Relatively better sort performances are achieved with the smaller sort areas for our tests.

For environments using neither sort direct writes nor asynchronous I/O, enabling one or both of these features will substantially improve disk sort performance.

For environments using only sort direct writes or only asynchronous I/O, enabling the other feature will moderately improve disk sort performance.

When neither feature is enabled, increasing sort area size improves sort performance marginally in our tests.

Note, however, that each of the 16 parallel sort slave processes in a test allocates a sort area. Thus, for each test in this section, the combined total memory used for sort areas was 16 MB, 32 MB, or 64 MB. In the parallel index creation case, the total amount of sort area allocated depends both on the degree of parallelism and on the sort area size. This ensured that no test in this section needed to perform intermediate merge runs. Alternatively the data size to be sorted per process is the total data set size divided by the degree of parallelism.

Cautionary Note: When comparing the test results for serial and parallel index creation keep in mind that different amounts of memory is used for sorting.

Test Algorithm: The following algorithm was used in this test suite:

```
from optimal values determined in previous test section, set
   SORT_WRITE_BUFFERS=4, SORT_WRITE_BUFFER_SIZE=64
foreach ASYNC_IO in (FALSE,TRUE)
   foreach SORT_DIRECT_WRITES in (FALSE,TRUE)
         foreach SORT_AREA_SIZE in (1M, 2M, 4M)
```

Test Results: Table 4 shows the results of this test suite. SORT_BUFFER_SIZE=64 and SORT_BUFFERS=4 were chosen for this test suite because these setting produced optimum results in the last test suite. That is, producing the best time for memory usage.

		SORT_DIRECT_WRITES	
ASYNC_IO	SORT_AREA_SIZE	FALSE	TRUE
FALSE	1 MB	14:22	04:57
	2 MB	14:00	05:05
	4 MB	13:35	05:14
TRUE	1 MB	05:01	04:33
	2 MB	05:11	04:42
	4 MB	05:30	04:55

Table 4: Elapsed Time for Large Parallel Sort Operation

Several conclusions can be drawn from the above results and are more easily understood when the difference in timings are plotted as shown and described in the next two figures.

Figure 11 shows parallel index creation time. In this section we will consider the slowest run our baseline for comparison. That is, the test run using the smallest sort area and neither asynchronous I/O nor sort direct writes are enabled. Each horizontal column shows index creation time. The y-axis shows whether asynchronous I/O and sort direct writes are enabled, and also the sort area size. For example, the column closest to the x-axis shows the elapsed time for the baseline timing for this test section; index creation time took 14:22 minutes/seconds. As another example, the column at the top of the graph shows that for a 4 MB sort area and neither asynchronous I/O or sort direct writes enabled, the index creation time was 4:55 minutes/seconds.

The “TT 1 MB” column shows the best performance at 4:33 minutes/seconds. That is, when sort area size is small and both asynchronous I/O and sort direct writes were enabled. This is to be expected because asynchronously writing out a 1 MB sort area at four different times is better than asynchronously writing out a 4 MB sort area once.

The “FT 1 MB” column shows that in an environment not using asynchronous I/O (possibly not supported), enabling sort direct writes will reduce index creation by one third (compared to the baseline case.)

The “TF 1 MB” column shows that in an environment not using sort direct writes, then enabling asynchronous I/O (if possible) will reduce index creation by almost one third.

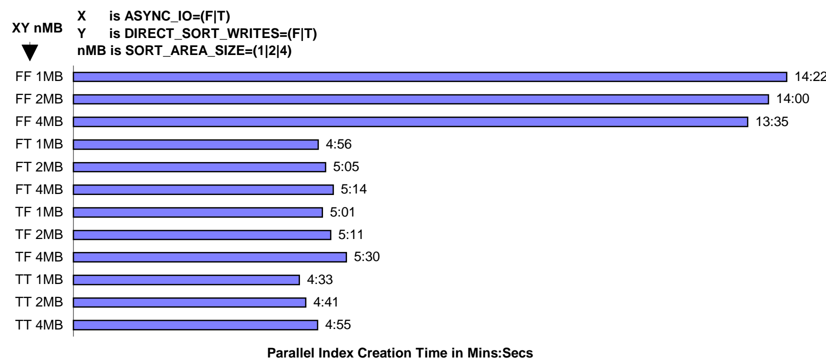


Figure 11: Parallel Index Creation Times -- ASYNC_IO, SORT_DIRECT_WRITES, SORT_AREA_SIZE

In Figure 12, using the baseline test (1 MB sort area and neither asynchronous I/O or direct sort writes enabled) for comparison, we plot the reduction in index creation time for different sort area size increases with asynchronous I/O and direct sort writes variously disabled and enabled. For example, the “FF 2 MB” column shows that when neither asynchronous I/O nor direct sorts are enabled, then increasing sort area size from 1 MB to 2 MB will improve disk sort performance by 2.6 percent.

The conclusions described columns “TT 1 MB,” “TF 1 MB,” and “FT 1 MB” shown in Figure 11 also apply in Figure 12.

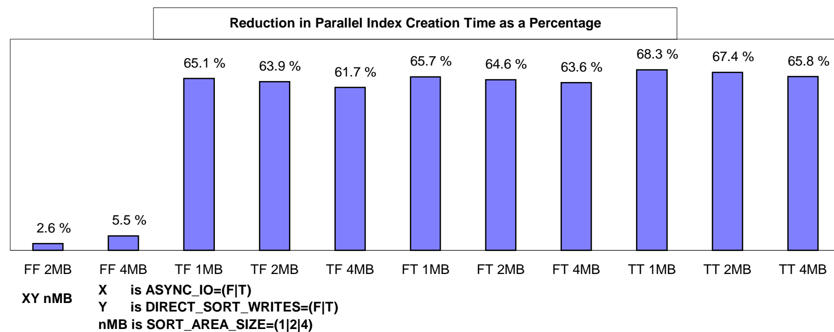


Figure 12: Reduction in Parallel Index Creation Time

Parallel I/O Profile for Async IO Disabled and Sort Direct Writes Disabled

Figure 13 shows the I/O throughput profiles for three parallel index creation tests when asynchronous I/O was disabled and sort direct writes was disabled. Graph 1 is for a test run using a 1 MB sort area, Graph 2 for 2 MB, and Graph 3 for 4 MB.

(The results in this section are used as a baseline for comparison in the next three sections.)

There is an initial peak in table reads in all graphs in Figure 13. Note how the peak is higher in the graphs for larger sort areas because more data can be read into the larger sort areas. This peak represents the initial read by the scan slaves of the table, the partition of index keys via key range through the table queue mechanism and finally into the sort area allocated by the sort slaves. This first read is somewhat synchronous and the superposition of these coherently results in the large spike. The actual completion of the read of the rows and the start of the sort run generation varies from slave to slave depending on key distribution in the extents. The variability of the start of sort run generation is further amplified by scheduling issues etc. of the sort slaves. As a result any form of coherence between the sort slaves regarding when they complete generating a sort run and start reading keys for generating the next run is completely lost. This lack of coherency ensures that we see averaged reads of data blocks and averaged writes of temporary sort segments to the Temp tablespace across the slaves.

After the initial peak in table reads, table reads and temp writes continue at a roughly constant throughput. This throughput is limited by buffer cache contention because all 16 sort processes are pushing sort segment blocks into the buffer cache and by DBWR ability to write the dirtied buffers generated by the parallel sort processes.

It is also important to keep in mind that when a full table scan is done in parallel data blocks are read directly into the private address space of the scan slaves bypassing the buffer cache. The read of the sort runs to merge and produce the final sorted output also uses the direct path mechanism and bypasses the buffer cache. Finally the sorted row source is consumed by the index build. As the indexes (more accurately sub indexes) are built the index segment blocks are again buffered and written out directly by the index build processes without using the buffer cache or the services of DBWR. We thus see that only the temporary sort segments are written through the cache and this proves to be the bottleneck as without Asynchronous I/O, DBWR simply does not write out the blocks fast enough, and there are many ‘free buffer wait’ events generated for the sorting processes.

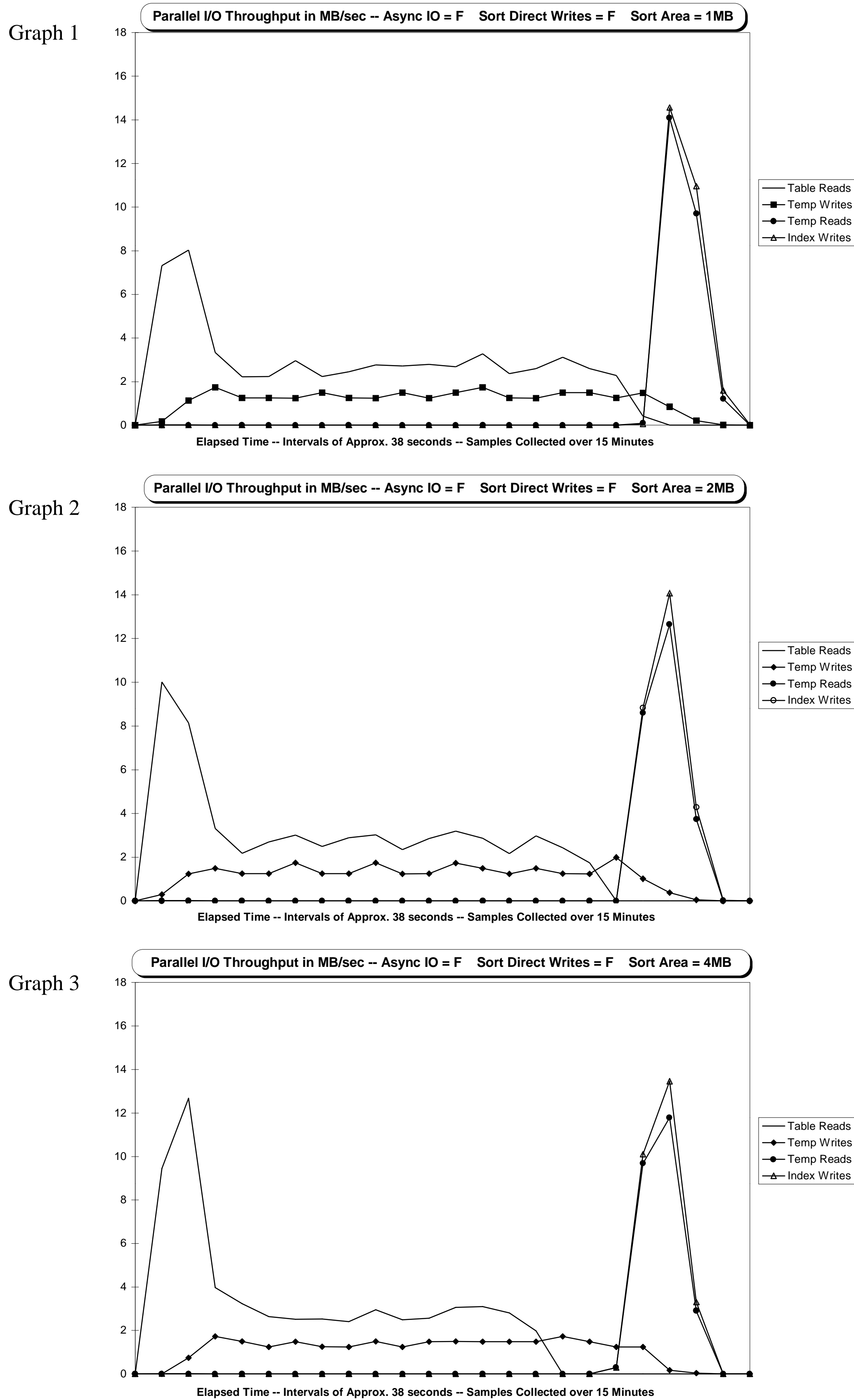


Figure 13: Parallel I/O Throughput in MB/sec -- Async IO=FALSE -- Sort Direct Writes=FALSE

Parallel I/O Profile for Async IO Disabled and Sort Direct Writes Enabled

Figure 14 shows the I/O throughput profiles for three parallel index creation tests when asynchronous I/O was disabled and sort direct writes was enabled. Graph 1 is for a test run using a 1 MB sort area, Graph 2 for 2 MB, and Graph 3 for 4 MB.

(Results in this section will be compared with the baseline results described on page 26.)

Compared to the baseline, invoking the sort direct writes feature improved performance dramatically for the tests in this section with a reduction in index creation time of 63 to 66 percent. The tests take far less time because the I/O operations take far less time. Thus, we see the architectural design of sort direct writes in action. I/O throughput is much higher than the baseline because the temp writes are not going through the buffer cache. (All graphs in Figure 14 show temp writes throughput far exceeding the "limit" of 2.44 MB/sec in the baseline.) It is worth keeping in mind that the area under a phase of the graph is constant. This is because this area represents the total number of blocks read or written, which is constant. One can immediately see which phase of the graph completes much more quickly due to better I/O throughput.

In the graphs, there appears to be an overlap in the sort runs phase and the merge phase. This is slightly deceptive. Each parallel sort operation completes their sort runs prior to beginning their merge phase. The overlap in the graph shows that some of the parallel sort processes are beginning their merge phase before other parallel sort processes have completed their sort runs.

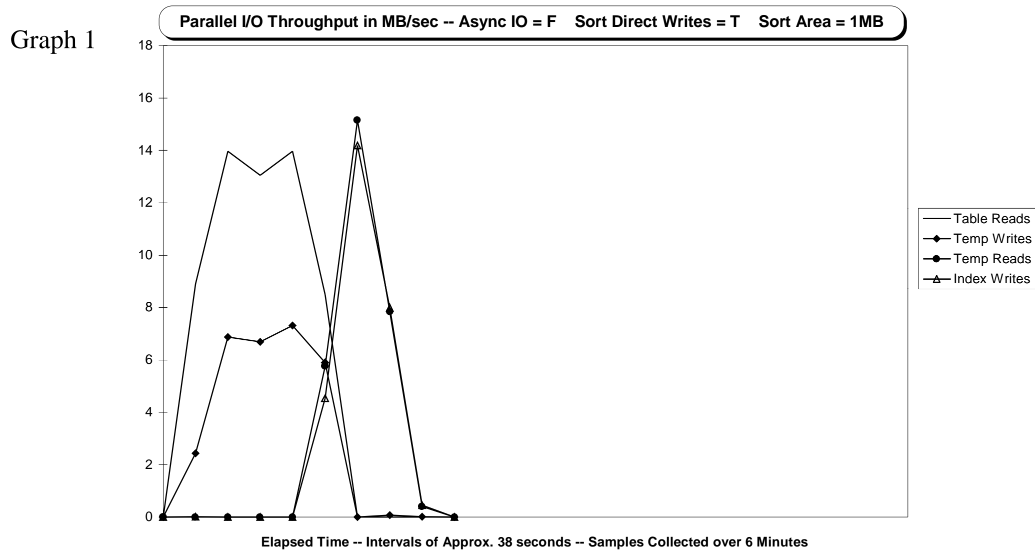
Note how the sustained throughput for table reads and temp writes decreases marginally with increasing sort area size, which causes a corresponding minor increase in run time. This is more obvious in Figure 11 on page 25. (Explanation 3.)

The marginal improvement (a 2 to 4 percent reduction) in index creation time can be attributed to Explanation 3. That is, table reads and temp writes throughput decreases as sort area size increases. What we see is an averaged reflection of this trend across all processes doing the sort. Furthermore, smaller sort areas translate into smaller bursts of I/O than for larger sort areas. For example, the cumulative affect of more frequently writing out 1 MB sort areas by 16 parallel sort processes provides sustained I/O that is more efficient than less frequently writing out 4 MB sort areas.

Conclusion: It is probably most effective to use a sort area size somewhat above the critical sort area size. This choice provides all of the performance while conserving overall system resources. Further, the performance per unit of sort memory allocated is highest for the smallest sort area size.

Finally we note that the performance improvement with smaller sort area size is within the margin of error that may affect and alter the results in the parallel case. The way to answer this would be to do several parallel runs and average across them to detect trends.

Graph 1



Graph 2

Parallel I/O Throughput in MB/sec -- Async IO = F Sort Direct Writes = T Sort Area = 2MB

Table Reads
Temp Writes
Temp Reads
Index Writes

Elapsed Time -- Intervals of Approx. 38 seconds -- Samples Collected over 6 Minutes

Graph 3

Parallel I/O Throughput in MB/sec -- Async IO = F Sort Direct Writes = T Sort Area = 4MB

Table Reads
Temp Writes
Temp Reads
Index Writes

Elapsed Time -- Intervals of Approx. 38 seconds -- Samples Collected over 6 Minutes

Figure 14: Parallel I/O Throughput in MB/sec -- Async IO=FALSE -- Sort Direct Writes=TRUE

Parallel I/O Profile for Async IO Enabled and Sort Direct Writes Disabled

Figure 15 shows the I/O throughput profiles for three parallel index creation tests when asynchronous I/O was enabled and sort direct writes was disabled. Graph 1 is for a test run using a 1 MB sort area, Graph 2 for 2 MB, and Graph 3 for 4 MB.

(Results in this section will be compared with the baseline results described on page 26.)

Compared to the baseline, invoking asynchronous I/O improved performance dramatically for the tests in this section with a reduction in index creation time of 61 to 65 percent. This shows that invoking asynchronous I/O allows DBWR to catch up with parallel sort processes dirtying the buffer cache. Once again we see how the enabling of Asynchronous I/O dramatically alters the time taken to complete the writes of sort segments to the temporary tablespace.

The marginal improvement (a 2 to 4 percent reduction) in index creation time can be attributed to Explanation 3 on page 11. That is, table reads and temp writes throughput decreases as sort area size increases. Furthermore, smaller sort areas translate into smaller bursts of I/O than for larger sort areas. For example, the cumulative effect of more frequently writing out 1 MB sort areas by 16 parallel sort processes provides sustained I/O that is more efficient than less frequently writing out 4 MB sort areas. (see Explanation 1.)

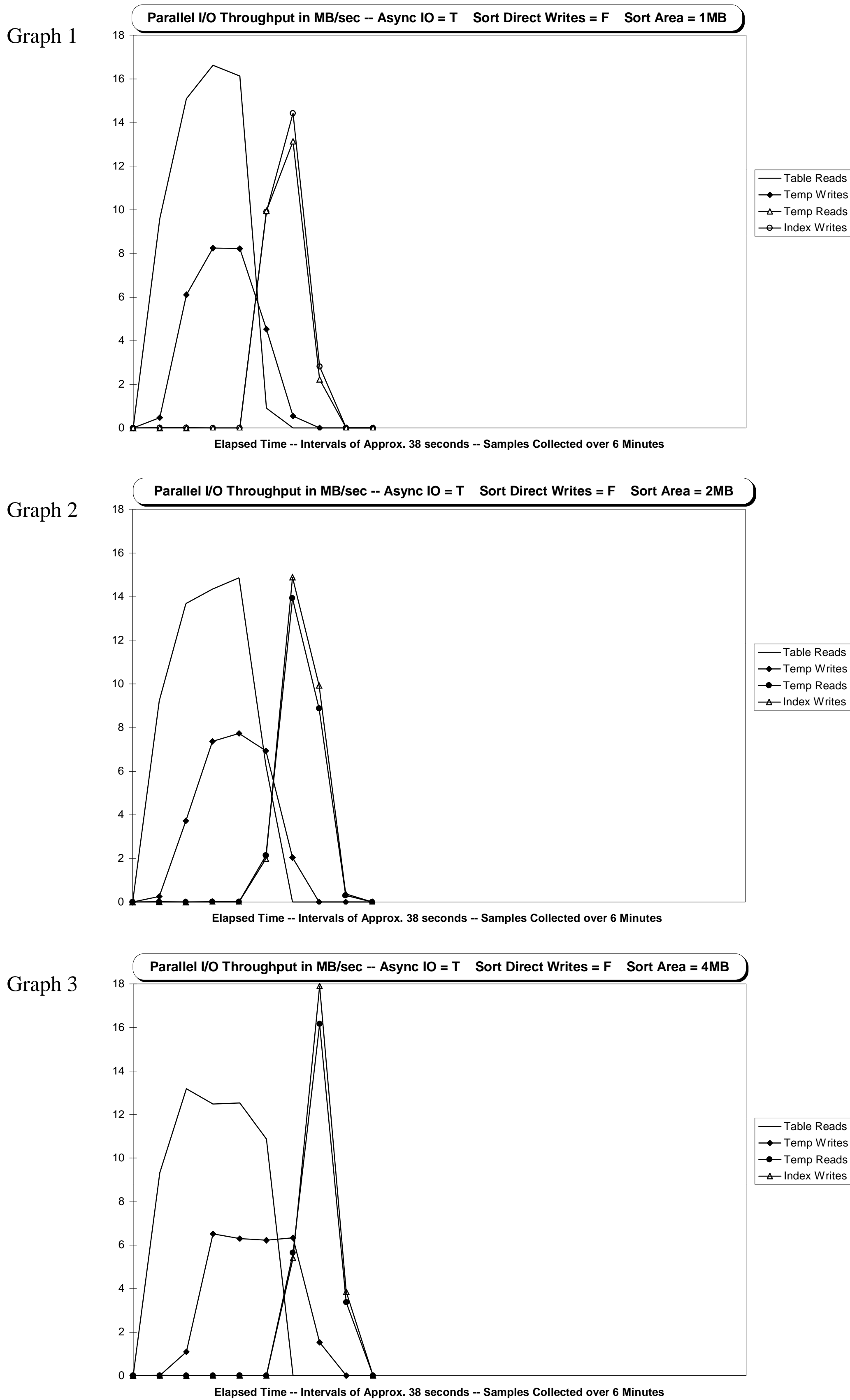


Figure 15: Parallel I/O Throughput in MB/sec -- Async IO=TRUE -- Sort Direct Writes=FALSE

Parallel I/O Profile for Async I/O Enabled and Sort Direct Writes Enabled

Figure 16 shows the I/O throughput profiles for three parallel index creation tests when asynchronous I/O was enabled and sort direct writes was enabled. Graph 1 is for a test run using a 1 MB sort area, Graph 2 for 2 MB, and Graph 3 for 4 MB.

(Results in this section will be compared with the baseline results described on page 26.)

Compared to the baseline, invoking both asynchronous I/O and the sort direct write feature improved performance dramatically for the tests in this section with a reduction in index creation time of 65 to 69 percent. This shows that enabling both asynchronous I/O and sort direct writes only performs marginally better than just enabling one of the features.

The marginal improvement (a 2 to 4 percent reduction) in index creation time can be attributed to Explanation 2 on page 11. That is, table reads and temp writes throughput decreases as sort area size increases. This is more critical in the parallel case because total memory used for all sort areas increases with the degree of parallelism, which was 16 for the tests in this section. Thus, the 1 MB sort area test used a total of 16 MB, the 2 MB test used 32 MB, and the 4 MB test used 64 MB.

Furthermore, smaller sort areas translate into smaller bursts of I/O than for larger sort areas. For example, the cumulative affect of more frequently writing out 1 MB sort areas by 16 parallel sort processes provides sustained I/O that is more efficient than less frequently writing out 4 MB sort areas.

In some sense we have not stressed the system enough as we would expect that if several producer processes try to have their I/O all handled by a single writer process then this writer process will eventually become a bottleneck. However we are not able to reach this regime with a single DBWR and asynchronous I/O. To some extent this is not totally surprising as the use of asynchronous I/O dispatches kernel threads to complete the I/O. But eventually for high enough rates of production of dirty buffers the latching used to protect shared objects will eventually become a bottleneck. When the producer process is able to do it's own buffering and I/O we will not expect this bottleneck to arise.

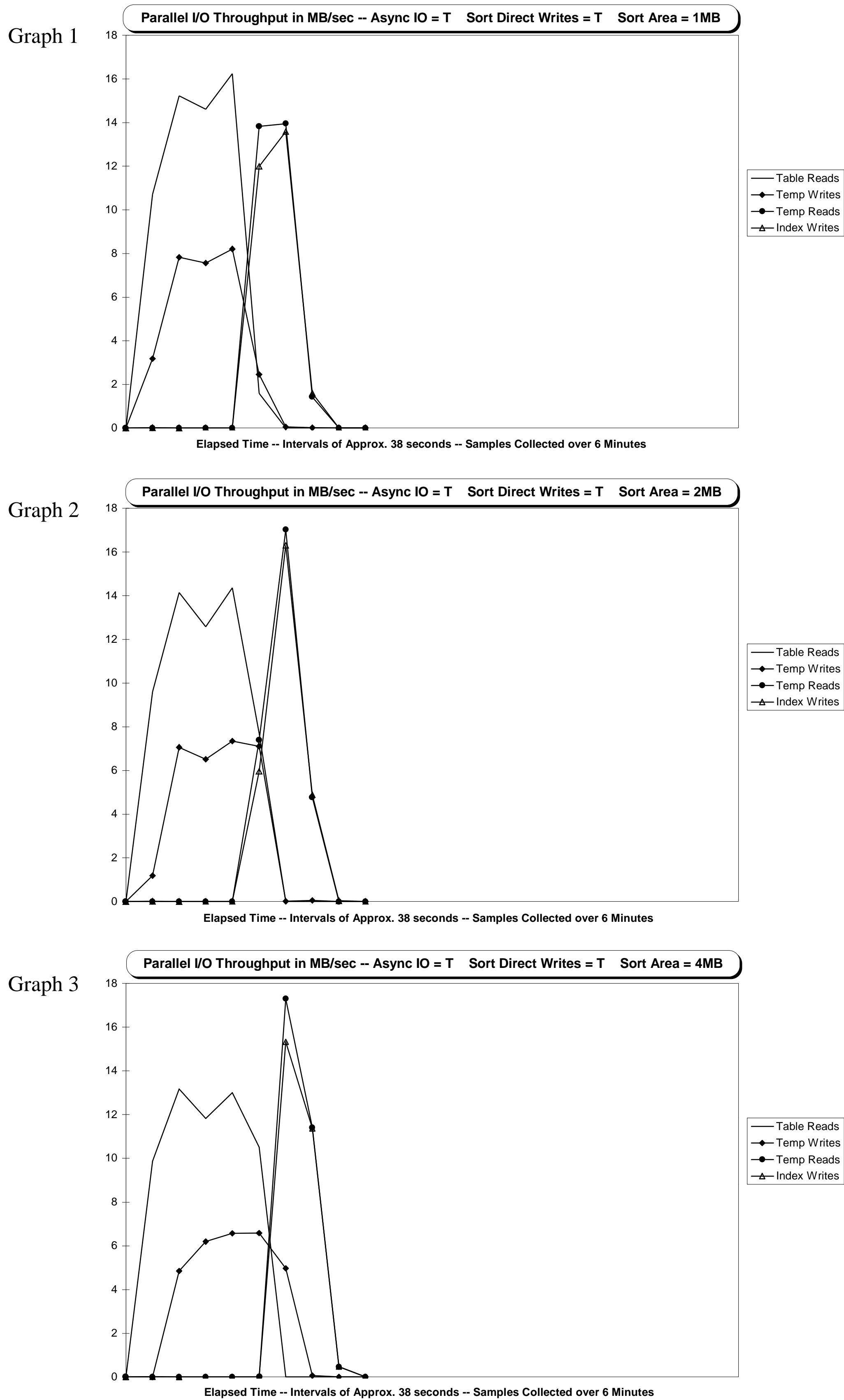


Figure 16: Parallel I/O Throughput in MB/sec -- Async IO=TRUE -- Sort Direct Writes=TRUE

APPENDICES

Appendix A -- Initialization Parameters

```
#control file, dump file locations, etc
ifile= /proj/p95141/a/dbs/configSSD.ora
compatible = 7.3.0.0.0
global_names  = FALSE
db_domain = regress.rdbms.dev.us.oracle.com
dblink_encrypt_login  = true
rollback_segments=(r_1_01, r_1_02, r_1_03, r_1_04, r_1_05, r_1_06, r_1_07, r_1_08, r_1_09,
r_1_10, r_1_11, r_1_12, r_1_13, r_1_14, r_1_15, r_1_16, r_1_17, r_1_18, r_1_19, r_1_20,
r_1_21, r_1_22, r_1_23, r_1_24, r_1_25, r_1_26, r_1_27, r_1_28)
db_file_multiblock_read_count = 16
_db_block_write_batch = 512

db_block_buffers = 5000
shared_pool_size = 50000000
log_checkpoint_interval = 50000
processes = 200
dml_locks = 500
log_buffer = 163840
sequence_cache_entries = 100
sequence_cache_hash_buckets = 89
audit_trail = false
max_dump_file_size = 5000000
checkpoint_process=true
db_files=512
db_file_simultaneous_writes=8
log_buffer=40960
log_checkpoint_interval=50000000
timed_statistics=false
parallel_max_servers=64
parallel_min_servers=2
parallel_default_scansize = 100

sort_area_size = [ 1 MB | 2 MB | 4 MB ] depending on test
thread=1
optimizer_mode=choose
gc_rollback_locks=100
gc_rollback_Segments=100
sessions=100
_trace_files_public=true

sort_direct_writes = [ true | false ] depending on test
sort_read_fac = 42
sort_write_buffers = [ 2 | 4 | 8 ] depending on test
sort_write_buffer_size = [ 32K |64K ] depending on test
async_io = [ true | false ] depending on test
```

Appendix B -- Description of Test Table

This appendix describes how the test table was built. Results presented in this paper were achieved by testing a large sort operation; namely, the creation of a non-unique index for a large table. Information in this appendix may be useful if you are interested in how the key data was distributed in the test table.

Column Position	Column Name	Datatype and Precision
1	SALES_DATE	DATE
2	STORE_ID	NUMBER
3	PROD_ID	NUMBER
4	REGULAR_QTY	NUMBER
5	REGULAR_AMT	NUMBER (38,2)
6	DISCOUNT_QTY	NUMBER
7	DISCOUNT_AMT	NUMBER (38,2)
8	PROMO_QTY	NUMBER
9	PROMO_AMT	NUMBER (38,2)

Table 5: Test table -- sales_desc13

The table sales_desc13 contains daily sales data for a given month. To ensure that generated data closely depicts realistic business sales patterns, the data generation program used weights to distribute the sales over time periods (days), stores and products. The following assumptions have been made relating to each dimension.

Time Holiday (long weekends) account for more sales than normal weekends

Non-holiday weekends account for more sales than week days

Store Some stores sell more than other stores. To achieve this, the generator groups stores into 10 classes and assigns weights to each store. There are a total of 2000 stores.

Product Some products sell more than other products. Products are classified into 10 classes and weights are assigned to these classes. There are 16700 products in all.

There are three different types of sales for any product viz. Regular, Discounted and Promotional. The proportion of quantities sold in these three types is : 50% of Regular , 25% Discounted and 25% Promotional. The sales prices for the three use the following weights : 1.0 for Regular , 4/5 for Discounted and 3/5 for Promotional.

The table, once loaded with data, does not change i.e. there are no updates/deletes on the data. This assumption was made to characterize large datasets in a typical DSS/Data Warehouse environments. Also, the data generated is 1% of the possible sample space from a combination of sales_date, prod_id, and store_id. This assumption was made so that the data generated is representative of the data in real-world applications.