Improving Query I/O Performance by Permuting and Refining Block Request Sequences

Xiaoyu Wang Mitch Cherniack

Department of Computer Science
Brandeis University
Waltham, MA 02454-9110
{wangxy, mfc}@cs.brandeis.edu

ABSTRACT

The I/O performance of query processing can be improved using two complementary approaches. One can try to improve on the buffer and the file system management policies of the DB buffer manager and the OS file system manager (e.g. page replacement). Alternatively, one can assume the above policies as fixed and instead improve the sequence of requests that are submitted to a file system manager and that lead to actual I/O's (*block request sequences*). This paper takes the latter approach. Exploiting common file system practices as found in Linux, we propose four techniques for *permuting and refining* block request sequences: *Block-Level I/O Grouping*, *File-Level I/O Grouping*, *I/O Ordering*, and *Block Recycling*. To manifest these techniques, we create two new plan operations, `MMS` and `SHJ`, each of which adopts some of the block request refinement techniques above. We implement the new plan operations on top of Postgres running on Linux, and show experimental results that demonstrate up to a factor of 4 performance benefit from the use of these techniques.

1. INTRODUCTION

A Database Management System (DBMS) depends upon two components to manage how data is transferred to and from disk: the *Database (DB) Buffer Manager*, and the underlying Operating System's *File System Manager*. The File System Manager manages the I/O requests concerning reads and writes of *files*. It controls the OS I/O buffer that is shared by all processes, and provides an interface for processes to manage persistent data in files. The DB Buffer Manager manages the dedicated DB buffer, whose content is determined by reads (queries) and writes (updates) of database objects (i.e., relations). Management here is exclusively of a buffer that is independent of the OS I/O buffer, and used only by the query processing engine of the DBMS.

Acting in concert though typically independently, the DB Buffer Manager and File System Manager together comprise a policy for mapping a query-generated sequence of block read and write *requests* (a "*block request sequence*") to a sequence of actual I/O's that result in data transfer to and from disk (an "I/O Sequence"). (Note that the I/O sequence may not be exactly the same as the block request sequence because some block requests in the sequence may be satisfied by the (DB or File System) buffer and therefore, never result in a disk I/O.) This combined policy can be characterized as a function, f, that accepts a block request sequence, s, as input and produces an I/O sequence, $f(s)$ as output.

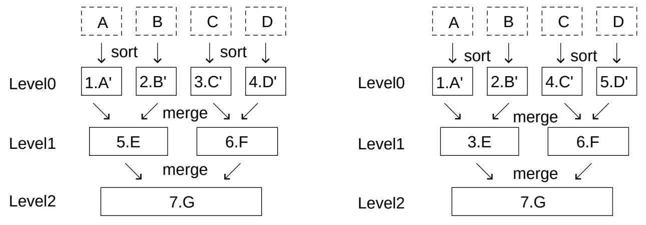
Proceedings of the 32nd VLDB Conference, Seoul, Korea, 2006

The effectiveness of f for a given block request sequence, f, can be measured in numerous ways, including:

1. *I/O Count:* the number of blocks transferred to and from disk in $f(s)$,
2. *Disk Seek Count:* the number of *disk seeks* required to process the I/O's of $f(s)$ (note that depending on f, consecutive I/O's of adjacent blocks can often be processed with a single disk seek), and
3. *I/O Time:* the amount of time required to perform the I/O operations in $f(s)$.

These measures are listed in increasing order of the number of I/O performance factors they consider. *I/O Count* measures the effectiveness buffer management. *Disk Seek Count* measures both effective buffer management as well as clustering of consecutively requested blocks in the block request sequence. *I/O Time* more generally measures the effectiveness of physical disk layout by accounting for consecutively requested sets of blocks that are a short disk seek apart).

A traditional approach to improving I/O performance is to alter f, either at the *DB Buffer* or *File System* levels. A survey of buffer management strategies is presented in [3]. A novel, complementary approach proposed in this paper assumes that f is fixed, and instead *permutes* and *refines* the block request sequences that are input to f, to better exploit buffer and file system management policies to achieve better I/O performance during query processing. For the purposes of this paper, we assume a fixed f which is the combined buffer and disk management policies of the Postgres DB Buffer Manager and the Linux File System Manager. While the results we present are specific to Linux and Postgres, we believe that the approach of manipulating block sequence requests to better exploit underlying buffer management policies has general applicability to any DBMS



(a) (b)

Figure 1: Sorting Steps of EMS(a) and MMS(b)

R_A,$\mathsf{W}_{A'}$, R_B, $\mathsf{W}_{B'}$, R_C, $\mathsf{W}_{C'}$, R_D, $\mathsf{W}_{D'}$,
$\mathsf{R}_{A'}$,$\mathsf{R}_{B'}$,W_{E_1},W_{E_2},$\mathsf{R}_{C'}$,$\mathsf{R}_{D'}$,W_{F_1},W_{F_2},...
(a)
R_A,$\mathsf{W}_{A'}$,R_B,$\mathsf{W}_{B'}$,$\mathsf{R}_{A'}$,$\mathsf{W}_{B'}$,W_{E_1},W_{E_2},
R_C,$\mathsf{W}_{C'}$,R_D,$\mathsf{W}_{D'}$,$\mathsf{R}_{C'}$,$\mathsf{R}_{D'}$,W_{F_1},W_{F_2},...
(b)

Figure 2: Block Request Sequences for `EMS`(a), `MMS`(b)

Block Requests	I/O Buffer	I/O Sequence
R_A	A	I_A
$\mathsf{W}_{A'}$	$A\ A'$	
R_B	$A\ A'\ B$	I_B
$\mathsf{W}_{B'}$	$B'\ A'\ B$	
R_C	$B'\ C\ B$	$O_{A'} I_C$
$\mathsf{W}_{C'}$	$B'\ C\ C'$	
R_D	$D\ C\ C'$	$O_{B'} I_D$
$\mathsf{W}_{D'}$	$D\ D'\ C'$	
$\mathsf{R}_{A'}$	$D\ D'\ A'$	$O_{C'} I_{A'}$
$\mathsf{R}_{B'}$	$B'\ D'\ A'$	$I_{B'}$
W_{E_1}	$B'\ E_1\ A'$	$O_{D'}$
W_{E_2}	$B'\ E_1\ E_2$	
$\mathsf{R}_{C'}$	$C'\ E_1\ E_2$	$I_{C'}$
$\mathsf{R}_{D'}$	$C'\ D'\ E_2$	$O_{E_1} I_{D'}$
W_{F_1}	$C'\ D'\ F_1$	O_{E_2}
W_{F_2}	$F_2\ D'\ F_1$	
...		

(a) `EMS`

Block Requests	I/O Buffer	I/O Sequence
R_A	A	I_A
$\mathsf{W}_{A'}$	$A\ A'$	
R_B	$A\ A'\ B$	I_B
$\mathsf{W}_{B'}$	$B'\ A'\ B$	
$\mathsf{R}_{A'}$	$B'\ A'\ B$	
$\mathsf{R}_{B'}$	$B'\ A'\ B$	
W_{E_1}	$B'\ A'\ E_1$	
W_{E_2}	$B'\ E_2\ E_1$	
R_C	$C\ E_2\ E_1$	I_C
$\mathsf{W}_{C'}$	$C\ E_2\ C'$	O_{E_1}
R_D	$C\ D\ C'$	$O_{E_2} I_D$
$\mathsf{W}_{D'}$	$D'\ D\ C'$	
$\mathsf{R}_{C'}$	$D'\ D\ C'$	
$\mathsf{R}_{D'}$	$D'\ D\ C'$	
W_{F_1}	$D'\ F_1\ C'$	
W_{F_2}	$D'\ F_1\ F_2$	
...		

(b) `MMS`

Table 1: Process Block Request Sequences (LRU)

and File System combination.

1.1 A Simple Example

To demonstrate our approach, we show two example block request sequences that both result in sorting a small file, R, consisting of four initial runs that are merged two at a time[1]. The first is that produced by a standard external merge sort. The second is a refinement of the first produced using our techniques.

Figure 1a illustrates the steps involved in sorting R using a standard *external merge sort* (`EMS`) as described in most introductory database texts [12]. The tree structured diagram presents the levels of the algorithm: level0 involves sorting individual runs, A, B, C and D into sorted runs A', B', C' and D' respectively; level1 involves merging runs A' and B' into a new run E, and C' and D' into a new run, F; and level2 involves merging E and F into the result run, G. Each run in the diagram is annotated with an integer (1-7) reflecting the order in which runs are constructed. Because `EMS` is a *lazy* merge algorithm [5], level1 runs E and F are only constructed after all level0 runs (A', B', C', and D') have been constructed.

Figure 1b shows how R is sorted using an alternative merge sort introduced in this paper (*Multilevel Merge Sort*, or `MMS`) that permutes the block request sequence produced by `EMS`. One of the major differences between `MMS` and `EMS` is that `MMS` performs an *eager* rather than *lazy* merge sort. As evidenced by the differing numbering scheme, this means that the level1 run, E, is constructed *immediately* after the level0 runs, A' and B', are constructed. Similarly, F is constructed immediately after C' and D'.

Figure 2 contrasts the block request sequences generated by `EMS` and `MMS`. A block request sequence shows data requests made by a process in the order these requests are issued. (R_X refers to a request to read block X.) For example, the block request sequence for `EMS` shows that every level0 run (e.g., A) is read, and its sorted version (A') is written before any level1 runs are constructed. On the other hand, the block request sequence for `MMS` shows that the writes of A' and B' are immediately followed by the reads of the same blocks for the purpose of generating the two pages (E_1 and E_2) of the level1 run, E. These two block request sequences result in different I/O sequences when processed by the Linux file system. Assume the file system practices applied by Linux are:

[1]To keep the example as simple as possible, we assume that each of the initial runs occupies a single page, and that the internal memory used for sorting runs (1 page) is distinct from the internal memory used for merging (3 pages) as in Postgres [14].

- *LRU-Based Page Replacement,*
- *Delayed Buffer Flushing:* Linux permits configuration of buffer flushing policies. For example, it can be configured to flush dirty pages only when these pages are evicted from the buffer (as we assumed in the example shown in Table 1 and r̃eftbl:emsmmsopt discussed below).

Table 1a and 1b compare the I/O buffer contents and the I/O sequences of the two block request sequences in Figure 2 assuming a 3-page I/O buffer. The first column of each table ("Block Requests") shows the block request sequence for each sort algorithm. The shaded block requests are the ones used for constructing level1 runs. The next 3 columns of each table ("I/O Buffer") show the contents of the buffer after each block request is satisfied, and the last column of each table ("I/O Sequence") shows what (if any) disk input (I_X) or output (O_X) operations must be executed for some block X to satisfy the corresponding block request.

It can be seen from Tables 1a and 1b that the block request sequence of `MMS` results in only 6 actual I/O's versus the 14 I/O's resulting from that of `EMS`. The better I/O behavior exhibited by `MMS` attributes to how it permutes its block request sequence to better exploit the following file system practices:

LRU-Based Page Replacement: Note that the 4th-8th requests of the `MMS` block request sequence involve reading runs A' ($\mathsf{R}_{A'}$) and B' ($\mathsf{R}_{B'}$) prior to merging them to construct run E (W_{E_1} and W_{E_2}). The reads of A' and B' occur im-

Block Requests	I/O Buffer	I/O Sequence
R_A	A	I_A
$\mathsf{W}_{A'}$	$A\ A'$	
R_B	$A\ A'\ B$	I_B
$\mathsf{W}_{B'}$	$B'\ A'\ B$	
R_C	$B'\ A'\ C$	I_C
$\mathsf{W}_{C'}$	$B'\ A'\ C'$	
R_D	$B'\ A'\ D$	$O_{C'} I_D$
$\mathsf{W}_{D'}$	$B'\ A'\ D'$	
$\mathsf{R}_{A'}$	$B'\ A'\ D'$	
$\mathsf{R}_{B'}$	$B'\ A'\ D'$	
W_{E_1}	$B'\ E_1\ D'$	
W_{E_2}	$E_2\ E_1\ D'$	
$\mathsf{R}_{C'}$	$C'\ E_1\ D'$	$I_{C'}$
$\mathsf{R}_{D'}$	$C'\ E_1\ D'$	
W_{F_1}	$F_1\ E_1\ D'$	
W_{F_2}	$F_1\ E_1\ F_2$	
...		

(a) `EMS`

Block Requests	I/O Buffer	I/O Sequence
R_A	A	I_A
$\mathsf{W}_{A'}$	$A\ A'$	
R_B	$A\ A'\ B$	I_B
$\mathsf{W}_{B'}$	$B'\ A'\ B$	
$\mathsf{R}_{A'}$	$B'\ A'\ B$	
$\mathsf{R}_{B'}$	$B'\ A'\ B$	
W_{E_1}	$B'\ A'\ E_1$	
W_{E_2}	$B'\ E_2\ E_1$	
R_C	$C\ E_2\ E_1$	$O_{E_2} I_C$
$\mathsf{W}_{C'}$	$C'\ E_2\ E_1$	
R_D	$C'\ D\ E_1$	I_D
$\mathsf{W}_{D'}$	$C'\ D'\ E_1$	
$\mathsf{R}_{C'}$	$C'\ D'\ E_1$	
$\mathsf{R}_{D'}$	$C'\ D'\ E_1$	
W_{F_1}	$F_1\ D'\ E_1$	
W_{F_2}	$F_1\ F_2\ E_1$	
...		

(b) `MMS`

Table 2: Process Block Request Sequences (OPT)

mediately after these runs have been constructed, meaning that they are still contained in the I/O buffer when the read requests are satisfied. Thus, `MMS` does not need to execute actual I/O's to satisfy these block requests. In contrast, `EMS` constructs run E with the 9th-12th requests of its block request sequence, meaning that A' and B' are required long after they have been constructed, and therefore after they were evicted from the buffer. As a result, `EMS` must perform actual I/O's ($I_{A'}$ and $I_{B'}$) to bring A' and B' back into the I/O buffer. The same phenomenon is exhibited in the construction of run F using C' and D'. As a result, `MMS` has no actual I/O's corresponding to actual I/O's, $I_{A'}$, $I_{B'}$, $I_{C'}$ and $I_{D'}$, executed by `EMS`. This demonstrates the caching benefits of eager merging resulting from better locality in the block request sequence.

Delayed Buffer Flushing: Not only must `EMS` read runs A', B', C' and D' into the buffer to construct runs E and F, `EMS` must previously have output these runs to disk when they were originally evicted from the I/O buffer. On the other hand, `MMS` no longer has a need for A' and B' once run E has been constructed, and thus need not ever flush A' or B' to disk! Of course, there is no way for a file system to know that a dirty page is no longer needed by an application. So `MMS` exploits *delayed buffer flushing* to trick the file system into never writing the dirty page to disk. The technique involves *block recycling*: fooling the file system into thinking it is updating a page of one file when it is in fact, overwriting this page with a page from a different file. This is what `MMS` does in response to the block request, W_{E_2}: page E_2 overwrites page A' without the file system's knowledge, thereby circumventing the page replacement operation that would have flushed A' to disk. Delayed buffer flushing enables this technique because A' is not forced to disk when it is constructed but only when it is evicted from the I/O buffer. Because the construction of E is interpreted by the file system as a write to A', eviction of A' never takes place. This optimization accounts for the remaining 4 actual I/O's ($O_{A'}$, $O_{B'}$, $O_{C'}$, and $O_{D'}$) that are performed by `EMS` but not `MMS`.

It should be stressed that the block request sequence manipulation illustrated here is complementary to any algorithmic improvements to file system management. Even given an optimal buffer management policy, f, the effectiveness of $f(s)$ can still be improved by permuting s. Table 2a and 2b compare the I/O buffer contents and the I/O sequences of `EMS` and `MMS` assuming that f uses optimal page replacement (OPT) [9]. Still, the block request sequence of `MMS` results in fewer actual I/O's than that of `EMS`. Table 2 shows that the block request sequence of `MMS` results in 5 actual I/O's, while that of `EMS` results in 6.

1.2 Our Contributions

This paper demonstrates the viability of permuting and refining block request sequences as a technique for improving the I/O performance of query processing. This technique has two benefits:

- It offers a complementary research direction to the work on developing new file system and DB buffer management strategies (e.g., new page replacement policies).
- It also provides a relatively uninvasive way to improve the performance of existing DB systems. Instead of altering the DB buffer manager or the file system manager of the underlying DB or file system, it assumes these are fixed, and exploits them at the operator level.

We will demonstrate these benefits in the context of Postgres running on Linux, and show how the block request sequences for temporary files constructed by Postgres' block operations (sort, hash join) can be permuted and refined to better exploit the management strategies used by the file system manager[2]. Thus, we assume a single fixed f, based on the file system management policies of Linux. We effectively permute and refine the block request sequence produced by Postgres sort and hash join operations by introducing alternative versions of these operations that produce alternative block request sequences. The results shown in this paper lay the groundwork for more general study of how permuting and refining block request sequences can improve the I/O performance for the management policies used in any underlying DBMS and file system.

The remainder of this paper is organized as follows. In Section 2, we discuss four common file system practices and present techniques for permuting and refining block request sequences in file systems to exploit these practices. In Section 3, we introduce two new sort and join operations, `MMS` and `SHJ`, that use the above techniques. In Section 4, we show experimental results and illustrate the I/O benefits of `MMS` and `SHJ` with file system tracing tools, and before discussing related work in Section 5 and concluding with future research directions in Section 6.

2. MANIPULATING BLOCK REQUEST SEQUENCES

In this section, we describe four common file system techniques that are used by Linux (Section 2.1) and four ways

[2]Postgres temporary file construction bypasses the DB Buffer manager and instead directly interacts with Linux file system manager to construct files.

$\mathsf{R}_A, \mathsf{R}_B, \mathsf{W}_C, \mathsf{W}_A, \mathsf{W}_B, \mathsf{R}_C, \mathsf{R}_A, \mathsf{W}_D, \mathsf{R}_D, \mathsf{W}_B, \mathsf{R}_D, \mathsf{W}_C$

(a) A Block Request Sequence

$\mathsf{R}_A, \mathsf{W}_A, \mathsf{R}_B, \mathsf{R}_A, \mathsf{W}_B, \mathsf{W}_B, \mathsf{W}_C, \mathsf{W}_D, \mathsf{R}_C, \mathsf{W}_C\ \mathsf{R}_D, \mathsf{R}_D$

(b) The Result of Block-Level I/O Grouping

$\mathsf{R}_A, \mathsf{W}_A, \mathsf{R}_B, \mathsf{R}_A, \mathsf{W}_D, \mathsf{W}_B, \mathsf{W}_B, \mathsf{R}_D, \mathsf{R}_D, \mathsf{W}_C, \mathsf{R}_C, \mathsf{W}_C$

(c) The Result of File-Level I/O Grouping

$\mathsf{R}_A, \mathsf{W}_A, \mathsf{W}_A, \mathsf{W}_D, \mathsf{R}_D, \mathsf{R}_D, \mathsf{R}_B, \mathsf{W}_B, \mathsf{W}_B, \mathsf{W}_C, \mathsf{R}_C, \mathsf{W}_C$

(d) The Result of I/O Ordering

Figure 3: A Block Request Sequence and its Permutations

to manipulate block request sequences to exploit them to achieve better I/O performance (Section 2.2).

2.1 File System Management in Linux

Linux employs the following file system techniques which can be exploited during query processing:

LRU-Based Page Replacement: LRU ([L]east [R]ecently [U]sed) is the most commonly used page replacement policy for managing the contents of a File System's I/O Buffer. Besides Linux, LRU-Based page replacements are also used in FreeBSD, Solaris, and Windows NT. Linux chooses pages to replace based on their page reference counters. Because it decreases page counters exponentially, its page replacement behavior approximates LRU.

I/O Merging: I/O bound applications issue continuous streams of I/O requests that are collected in an *I/O request buffer*. Disk scheduling policies (e.g., LOOK or SCAN) choose the order in which these requests are satisfied. But independently of the disk scheduling policy, Linux *merges* pending requests for I/O's over adjacent blocks into a single multi-block request. This has the effect of reducing the number of actual I/O's (and hence, disk seeks) required to respond to all I/O requests in the I/O request buffer.

Delayed Buffer Flushing: When a page in the I/O buffer is changed ("dirtied"), an Operating System can flush its updated contents immediately to disk (*write-through*) or delay this operation until some condition is met or the page is evicted (*write-back*). Write-through reduces the risk of failure and the cost of recovery, while write-back (by allowing multiple writes to the same page to result in one rather than multiple writes to disk) can improve I/O performance. Most Operating Systems, including Linux, permit the configuration of the buffer manager to enact either of these flushing policies, though because of its performance advantages, delayed buffer flushing (write-back) is more common.

Block Grouping: This file system practice involves pre-allocating a contiguous group of disk blocks to a file when a new block is needed. For example, Ext2FS pre-allocates a group consisting of up to 8 adjacent blocks by default when allocating a new block [2]. The goal of block grouping is to make a file as contiguous as possible so as to speed up sequential access.

2.2 Block Request Sequence Refinement

We introduce four techniques for permuting and refining block request sequences in ways that exploit the file system practices discussed in Section 2.1. To illustrate these techniques, we will refer to the example block request sequence shown in Figure 3a involving four disk blocks: A, B, C, and D.

2.2.1 Technique #1: Block-Level I/O Grouping

Block-level I/O grouping involves permuting a block request sequence so that block requests involving the same block are in close proximity within the sequence.[3] We saw one example application of this technique in Section 1, where `MMS` reordered block requests so that requests to read blocks of level0 runs to construct level1 runs were made immediately after the level0 runs were constructed. As another example, given the block request sequence of Figure 3a, block-level I/O grouping might result in the block request sequence shown in Figure 3b where all block requests involving pages A and B occur before all block requests involving pages C and D.

Block-level I/O grouping exploits two file system practices to achieve better I/O performance for the same set of block requests:

LRU-Based Page Replacement: Block-level I/O grouping effectively reduces the block-reference working set given a fixed window size. Because LRU attempts to maintain the current working set in the file system buffer, this means that the resulting block request sequence is likely to result in more buffer hits and fewer actual I/O's to satisfy a given set of block requests (as was the case in the example presented in Section 1).

Delayed Buffer Flushing: If a page is only flushed to disk when it is evicted from the file system buffer, block-level I/O grouping improves the chance that a dirty page is flushed to disk only once, even if it is written multiple times in a given block request sequence.

2.2.2 Technique #2: File-Level I/O Grouping

File-level I/O grouping involves permuting a block request sequence so that block requests involving blocks from the same *file* are in close proximity within the sequence. For example, if blocks A, B and D from the block request sequence of Figure 3a were from one file, R, and block C was from another file, S, then file-level I/O grouping of this sequence might result in the block request sequence shown in Figure 3c where all block requests involving file R are grouped together.

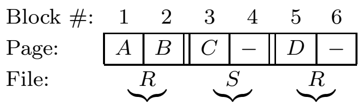
This technique exploits *block grouping* to achieve better I/O performance for the same set of block requests. To illustrate, suppose that an application generates the block request sequence,

$$\mathsf{W}_A, \mathsf{W}_B, \mathsf{W}_C, \mathsf{W}_D,$$

such that A, B and D belong to a new file, R and C belongs to a new file, S. Suppose further that block groups pre-allocated to a file can hold 2 pages. Then, the block request sequence above would result in a block group being allocated

[3]Obviously, not all permutations of a block request sequence are possible. For example, a read request and a write request of the same block cannot be reordered without changing the semantics of the requesting application. However, the plan operations that we propose are constrained to permute block request sequences in a semantics-preserving way.

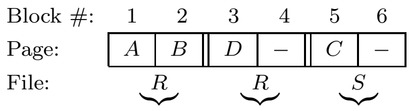
to file R and assigned pages A and B, followed by a block group being allocated to file S and being assigned page C (with the second block in the group unassigned), followed by a second block group being allocated to file R and assigned page D (with the second block in the group unassigned). This is illustrated below:



Now suppose that file-level I/O grouping is applied to the above block request sequence, resulting in the block request sequence shown below:

$$\mathsf{W}_A, \mathsf{W}_B, \mathsf{W}_D, \mathsf{W}_C.$$

This block request sequence would produce the following block assignments:



Observe that given the original block allocation, a file scan of R requires the disk head to travel 5 blocks (from block 1 to block 5), whereas the same file scan with the blocks allocated as above requires the disk head to travel only 3 blocks (from block 1 to block 3). This simple example illustrates that file-level grouping can help reduce the cost of performing a sequential scan of any file whose blocks are allocated from disk as a result of executing a given block request sequence.

An alternative strategy to make files more contiguous is to increase the sizes of block groups. However, this may result in increased internal fragmentation (by increasing the number of unused blocks at the end of a file) and therefore may increase the disk head distance between files, making algorithms that require alternating I/O's between files (as in a `MERGE` operation) more expensive.

As with block-level I/O grouping, arbitrary reordering of a block request sequence is not possible without affecting the semantics of the requesting applications. However, our `SHJ` operator (Section 3) reorders independent page writes that construct partition files, thereby preserving the semantics of the join.

2.2.3 *Technique #3: I/O Ordering*

Suppose that blocks A, B, C and D from the block request sequence of Figure 3a lie on tracks 100, 150, 151 and 101 respectively. Then, it may pay to order the block request sequence so that block requests for blocks that lie on adjacent tracks are themselves adjacent, as demonstrated in the block request sequence of Figure 3d.

This reordering exploits I/O merging. Specifically, if adjacent I/O requests for adjacent blocks both result in active I/O's, then the adjacent requests will be merged into a single multi-block request. Take the block request sequence of Figure 3d for example. If the 3^{rd} and 4^{th} requests in this sequence (writing A and D) both go to disk, they will do so with a single I/O. Similarly for the 9^{th} and 10^{th} requests (writing B and C). As with block-level and file-level I/O grouping, our operator that performs I/O ordering (`SHJ`) does so in a semantics-preserving way.

2.2.4 *Technique #4: Block Recycling*

Suppose, given the block request sequence of Figure 3a, that A belongs to a file that is temporary and no longer needed after it is last referenced (i.e., after it is read as a result of executing the 8^{th} request of the sequence). Because it belongs to a temporary file that is no longer needed, even though it is dirty it is unnecessary to write this page to disk when it is evicted from the I/O buffer. There is no way for a file system to know that a dirty page belongs to a file that is no longer needed by an application. However, a query operator can "trick" the file system into throwing out the dirty pages of a file that is no longer needed by recycling those pages on disk, reassigning them to some other file of the same size. For example, if A and D are both single page files, A is temporary and no longer needed, and D is a new file, then the write to D that follows the last read of A does not require allocation of a new block but can instead overwrite the block allocated for A. As a result, the dirty contents of A never get written to disk. This benefit was seen in the example application of `MMS` presented in Section 1, where pages A' and B' (C' and D'), used to construct run E (F) were overwritten by the pages of E (F), and therefore never written to disk.

A second benefit to disk block recycling is to allow an operator to circumvent LRU in choosing a page to replace in the buffer. For example, a page, A that belongs to a temporary file that is no longer needed may have been the most recently referenced page in the buffer at the time when eviction is necessary. LRU will evict the oldest page in the buffer, predicting that it will be the page that won't be referenced for the longest time. But an application can use block recycling to instead overwrite A, knowing that A will never be referenced again.

3. MODIFYING BLOCK REQUEST SEQUENCES

In this section, we present an alternative external sorting algorithm (Multi-Level Merge Sort, or `MMS`), and an alternative hash join algorithm (Sort Hash Join, or `SHJ`) that differ from traditional sorting and hash join algorithms in the block request sequences they produce.

`MMS` and `SHJ` both adopt some of the block request sequence refinement techniques described in in Section 2.2. First, we explain intuitively how they incorporate these techniques. Then, we present the algorithms. Finally, we demonstrate how these algorithms produce alternative block request sequences with examples.

3.1 Multilevel Merge Sort (MMS)

`MMS` is a variation of eager merge sort [6] that recycles blocks from runs that are no longer needed for use in new runs. An eager merge sort differs from the lazy merge sort of `EMS` by eagerly merging runs as soon as enough runs have been constructed, and prior to the construction of all runs of the same merge level. `MMS` performs an eager merge sort, but also records the blocks of runs that have been merged (which can be identified with file names and offsets) and uses them to store blocks of newly constructed runs. By merging runs "eagerly", `MMS` reads runs shortly after writing them and achieves *Block-Level I/O Grouping*. By replacing the blocks of old runs with those of new runs, `MMS` achieves *Block Recycling*. The effectiveness of `MMS`' block request sequence refinement was demonstrated in Section 1. In Table 1b, 4 actual disk reads, $I_{A'}$, ..., $I_{D'}$ are avoided because

MMS achieves block-level I/O grouping, 4 actual disk writes, $O_{A'}, \ldots, O_{D'}$ are also avoided because MMS achieves block recycling.

Algorithm MMS (Table S)
```
Initialize run counter and run pointers,
  set $C_0$, ..., $C_M$ to 0,
  set $R_0^0$, ..., $R_0^{N-1}$, ..., $R_M^0$, ..., $R_M^{N-1}$ to NULL

REPEAT
  CreateInitialRuns:
    REPEAT
      $R_0^{C_0}$ := {load tuples from input and sort them}
      $C_0$ := $C_0$ +1
    UNTIL $C_0$ = $N$ or EOF ($S$)

  MultiLevelMerge:
    $L := 0$
    REPEAT
      $R_{L+1}^{C_{L+1}}$ := MergeWithRecycle($R_L^0$ ...$R_L^{C_{L-1}}$)
      $C_L$ := 0
      $C_{L+1}$ := $C_{L+1}$ +1
      $L := L + 1$
    UNTIL ($C_L$ < $N$)
UNTIL EOF ($S$)
{merge runs level by level until all runs are merged}
```

Figure 4: Pseudocode of MMS

Figure 4 shows the pseudocode of MMS, and assumes that the merge fan-out is N and the maximal merge level is M. The counter, C_i, counts the number of runs at level i, and the pointer, R_i^j, refers to the j^{th} run at level i. MMS consists of the following two phases:

- *CreateInitialRuns:* This phase reads tuples from the input relation, S, until the memory is full, or until there is no tuple left in S (EOF (S)). It then sorts tuples in memory and stores them as an initial run, and repeats this process until there are N initial runs ($C_0 = N$), or until there is no tuple left in S. $R_0^0, \ldots, R_0^{C_0-1}$ point to the newly created runs.
- *MultiLevelMerge:* This phase merges all runs of level i into new runs at level $i+1$, until it reaches a level, L, containing fewer than N runs ($C_L < N$). Function MergeWithRecycle records the blocks of runs that are no longer needed, and recycles them for blocks of new runs.

MMS alternates between these two phases until the input is exhausted. Then, starting from level 0, MMS merges runs level by level until the final run is produced.

In short, by merging runs *eagerly* and writing new runs to no longer needed runs, MMS achieves *Block-Level I/O Grouping* and *Block Recycling,* and exploits *LRU-Based Page Replacement* and *Block Recycling.*

3.2 Sort-Hash Join (SHJ)

SHJ is a variation of Hybrid Hash Join (HHJ) that performs one pass replacement selection sort over tuples on their partition numbers prior to their placement in partitions. HHJ uses a separate output buffer to collect tuples that are stored in partition files. Whenever an output buffer gets full, it sends a block request to the file system to write the buffer to the correspondent partition file. Thus, it is the order in which the output buffers get full that determines the block request sequence of HHJ. SHJ reorders tuples before writing them to the output buffers, which changes the order in which the output buffers fill, and as a consequence, changes the order of the block requests. By sorting by partition number prior to partitioning, SHJ achieves both *File-Level I/O Grouping* and *I/O Ordering.*

Algorithm SHJ (Table R, Table S)
```
Create a buffer for each partition $B_0$ ...$B_{N-1}$
Open partition files for $R$ and $S$, $R_0$ ...$R_{N-1}$, $S_0$ ...$S_{N-1}$

Build:
 {Read tuples from R, insert them to heap H, until the
   memory is full, or until all tuples in R are read}

 REPEAT
   $t_o$ := HeapRemove ($H$)
   IF ($t_o \neq$ NULL) THEN
      $p_o$ := Partition ($t_o$)
      $B_{p_o}$ := AddToBuffer ($B_{p_o}$, $t_o$)
      IF ($B_{p_o}$ is full) THEN
         WriteBuffer ($B_{p_o}$, $R_{p_o}$)

   $t_i$ := NextTuple ($R$)
   IF ($t_i \neq$ NULL) THEN
      $p_i$ := Partition ($t_i$)
      $H$ := HeapInsert ($H$, $t_i$, $p_i$)
 UNTIL ($t_o$ = NULL)
 {Write $B_0$, ..., $B_{N-1}$ to $R_0$, ..., $R_{N-1}$}

 {Repeat the above procedure, sort and partition S
    into $S_0$ ...$S_{N-1}$ }

Probe:
 FOR $i := 0$ TO $N - 1$
   CreateHashTable ($R_i$)
   REPEAT
      $t$ := NextTuple ($S_i$)
      {Probe the hash-table and output join results}
   UNTIL EOF ($S_i$)
```

Figure 5: Pseudocode of SHJ

SHJ determines the number of partitions it creates in the same way that HHJ does. Let R be the build relation (the smaller of the join relations), M be the available memory, $|R|$ and $|M|$ denote the size of R and M measured in memory pages. The number of partitions SHJ creates is

$$N = \left\lceil \frac{|R| \times F - |M|}{|M| - 1} \right\rceil + 1 \quad (1)$$

(F is a fudge factor, which in this paper, we assume to be 1.) N is also the number of partition files SHJ creates for each join relation. SHJ uses N memory pages to create output buffers and the remaining $|M| - N$ memory pages to reorder tuples.

Figure 5 shows the pseudocode of SHJ. R and S are the build and probe relations of the join. N is the partition number calculated with Equation 1. We use a heap H to reorder tuples based on replacement selection. SHJ consists of the following two phases:

- *Build:* This phase reads tuples from the input, and inserts them into a priority heap ordered by their partition numbers. It then removes tuples from the top of the heap and stores them into the output buffers of their corresponding partition files. Function HeapInsert (H, t_i, p_i) inserts tuple t_i into a position in heap H based on its partition number p_i. HeapRemove (H) returns the tuple at the top of H and removes it from H.

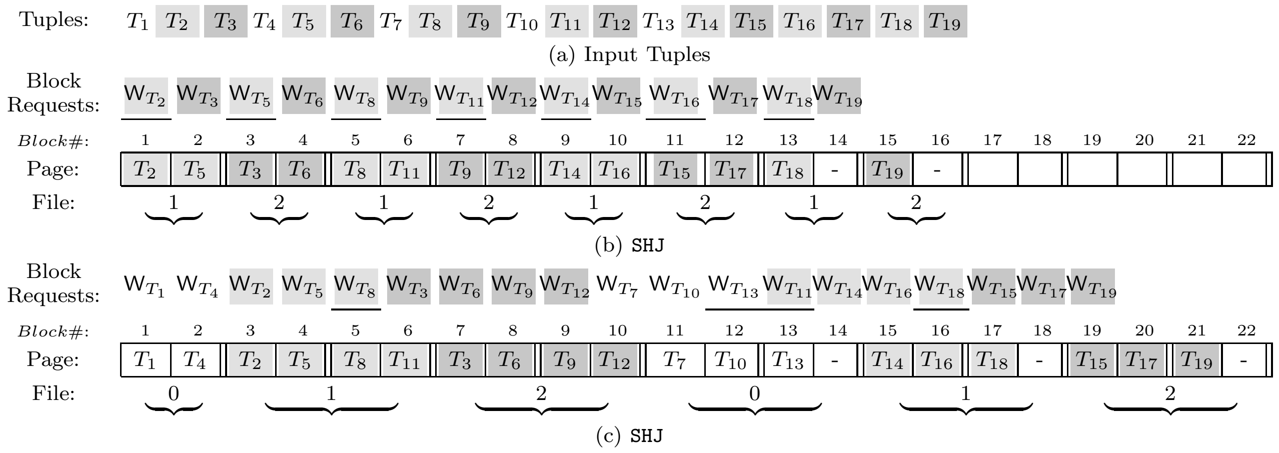

Tuples: T_1 T_2 T_3 T_4 T_5 T_6 T_7 T_8 T_9 T_{10} T_{11} T_{12} T_{13} T_{14} T_{15} T_{16} T_{17} T_{18} T_{19}
(a) Input Tuples
Block Requests: W_{T_2} W_{T_3} W_{T_5} W_{T_6} W_{T_8} W_{T_9} $W_{T_{11}}$ $W_{T_{12}}$ $W_{T_{14}}$ $W_{T_{15}}$ $W_{T_{16}}$ $W_{T_{17}}$ $W_{T_{18}}$ $W_{T_{19}}$
Block#: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
Page: T_2 T_5 T_3 T_6 T_8 T_{11} T_9 T_{12} T_{14} T_{16} T_{15} T_{17} T_{18} - T_{19} -
File: 1 2 1 2 1 2 1 2
(b) SHJ
Block Requests: W_{T_1} W_{T_4} W_{T_2} W_{T_5} W_{T_8} W_{T_3} W_{T_6} W_{T_9} $W_{T_{12}}$ W_{T_7} $W_{T_{10}}$ $W_{T_{13}}$ $W_{T_{11}}$ $W_{T_{14}}$ $W_{T_{16}}$ $W_{T_{18}}$ $W_{T_{15}}$ $W_{T_{17}}$ $W_{T_{19}}$
Block#: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
Page: T_1 T_4 T_2 T_5 T_8 T_{11} T_3 T_6 T_9 T_{12} T_7 T_{10} T_{13} - T_{14} T_{16} T_{18} - T_{15} T_{17} T_{19} -
File: 0 1 2 0 1 2
(c) SHJ

Table 3: Partitioning Input Tuples (a) with HHJ (b) and SHJ (c)

- *Probe:* Same as in HHJ, this phase creates an in-memory hash-table, and populates the hash-table with all tuples from a build partition file R_i. It then probes the hash-table using tuples read from the corresponding probe partition file S_i to produce join results. Function CreateHashtable (R_i) creates a hash-table and inserts all tuples into R_i to it.

We illustrate how SHJ achieves *File-Level I/O Grouping* and *I/O Ordering* with a simple example. Assume a relation of 19 pages that is partitioned with 7 memory pages ($|R| = 19, |M| = 7$). According to Equation 1, HHJ creates 2 partition files ($N = 3$). We also assume for simplicity that:

- the size of a disk block and a memory page are equal,
- the file system pre-allocates a group of 2 adjacent disk blocks to a file when a new block is needed, and
- each tuple occupies one memory page (or disk block).

Table 3a shows all the tuples to be partitioned. (Tuples, block requests and disk pages are in white if they belong to partition 0, light gray if to partition 1, and dark gray if to partition 2.) HHJ keeps the tuples of partition 0 (T_1, T_4, ...) in memory and writes the remaining 14 tuples to partition files. The first row in Table 3b ("Block Requests") shows its block request sequence. The file system allocates block groups to partitions 1 and 2 according to the block requests. The result shows that both partition files 1 and 2 are stored in 4 separate block groups (row "Page" and "File"). Thus, accessing each of them sequentially requires 4 disk seeks. In total, it requires 8 disk seeks to access the partition files of HHJ.

To partition the same relation using the same amount of memory, SHJ creates 3 partition files. Thus, SHJ writes one more partition file to disk then HHJ does (partition 0). However, though more data are written and read to and from disk by SHJ, the costs of the reads and writes are offset by the placement of blocks from SHJ resulting in fewer disk seeks. Of the 7 memory pages, SHJ uses 3 as output buffers, and the remaining 4 to sort tuples. Tuples are written to their partition files with the block request sequence shown in Table 3c. Unlike HHJ, SHJ writes the tuples of partition 0 to a file as well. Table 3c also shows how the disk blocks are allocated to each partition file. Partition 0 is stored in 3 block groups, two of which are adjacent (the block groups starting from block 11 and 13). Accessing it sequentially requires 2 disk seeks. Partition 1 and 2 are both stored in 4 block groups. Accessing each of them sequentially requires 2 disk seeks. In total, it requires 6 disk seeks to access the partition files of SHJ. This compares favorable to the 8 disk seeks required to access the partition files of HHJ. This demonstrates the benefit brought by *File-Level I/O Grouping*.

Table 3 also shows that SHJ achieves *I/O Ordering*. As discussed in Section 2, consecutive block requests that require consecutive disk blocks are likely to be merged into one actual I/O by the I/O scheduler. The block request sequence of HHJ contains 7 block requests that are followed immediately by requests demanding disk blocks that are *not* adjacent to the blocks they demand (These are underlined in Table 3a: W_{T_2}, W_{T_5}, ...). Meanwhile, the block request sequence of SHJ contains only 4 such block requests. Again, although the request sequence of SHJ contains more requests, it can be processed with fewer actual I/O's and therefore faster because the I/O scheduler is able to merge more requests in it into one[4]. Thus, by sorting, SHJ also achieves *I/O Ordering*. This reduces the number of seeks required to write the partition files.

It should be pointed out that SHJ is beneficial versus HHJ only in certain cases. Specifically, SHJ is beneficial if it produces block request sequences in which the number of consecutive block requests of the same file are greater than the block group pre-allocation size of the file system. It can be seen from Table 3c that it is the large number of consecutive block requests of the same file that forces the file system to allocate several adjacent block groups to the file. For example, block requests, W_{T_3}, W_{T_6}, W_{T_9}, and $W_{T_{12}}$, force the file system to allocate two adjacent block groups (block groups starting from block 7 and 9) to partition 2. Therefore, the average number of consecutive block requests belonging to the same file indicates whether SHJ will be benificial, which can be estimated from the partition number and the available memory using Equation 2.

$$G = \left\lceil \frac{2 \times (|M| - N)}{N} \right\rceil \qquad (2)$$

[4]Note that a block request sequence reflects the order in which blocks are written to the I/O buffer. If two blocks are written to the I/O buffer closely, they are likely to be flushed out together. To make this example simple, we leaved out the part how the I/O buffer flushes dirty pages.

When G is greater than the block group pre-allocation size of the file system, SHJ is benefitial.

To summarize, SHJ's strategy of partially sorting its input during the build phase of the hash join achieves *File-Level I/O Grouping* and *I/O Ordering*, and exploits *I/O Merging* and *Block Grouping.*

4. IMPLEMENTATION AND RESULTS

To determine the benefits of performing block request reordering in query processing, we implemented MMS and SHJ on top of Postgres 8.0 running on Linux. In Section 4.1, we compare the execution time of MMS against the Postgres sort, Polyphase Merge Sort (PPMS) [7] and several other external sorting algorithms. In Section 4.2, we compare the performance of SHJ with that of Postgres' built-in Hybrid Hash Join (HHJ). For each of these experiments, we also test our theories about how their benefits derive from better exploitation of file system practices by tracing file system behavior during query processing.

We conducted all of our experiments on a system with a 2 Ghz Pentium 4 processor, (the machine has two processors, but only one of them is in use in our experiments), 1 GB RAM, and two IDE hard drives over Linux-2.4.31 compiled with modules that enable tracing of disk I/O and file system activity. One 120 GB hard drive is used to store the Linux kernel, executables, and all user files. The other 300 GB hard drive is dedicated to database files and all files generated by Postgres during query processing. The 300 GB hard drive is formatted to the ext2 file system with a file system block size of 4KB, and a block group pre-allocation size of 8 blocks. The database files use 11% of the hard drive and are all stored contiguously.

The tracing code is executed only when certain tracing modules are loaded into the system. All experiments are run with and without loading these tracing modules. We use the default I/O buffer flushing configuration of Linux; pages are flushed to disk under the following conditions:

- dirty pages are flushed *asynchronously* when over 30% of the whole memory contains dirty pages.
- dirty pages are flushed *synchronously* when over 60% of the whole memory contains dirty pages, until the dirty pages ratio drops below 20%.
- dirty pages which are older than 3 seconds are flushed *asynchronously.*

The plan execution times were generated using Postgres' EXPLAIN ANALYZE tool. Our testing data is generated with DBGEN for TPC-H [15] with various scale factors. We categorize tables by their sizes: tables less than 512 MB are small, between 512 MB and 1.5 GB are medium, and greater than 1.5 GB are large. We choose two tables from each category in our experiments.

To make sure that the timing result of the algorithms were not influenced by external processes (i.e., I/O workloads or buffer page references from other processes), we reboot the system immediately before a query is processed. Only kernel processes and Postgres are running in the system during query processing.

4.1 Multilevel Merge Sort

In this Section, we describe experimental results that show:

	Eager or Lazy	Block Recycling
PPMS	Lazy	Yes
PPMS-NR	Lazy	No
MMS	Eager	Yes
MMS-NR	Eager	No

Table 4: Sort Algorithm Features

	Table	Scale Factor	Size (MB)
Small-1	Partsupp	3.0	438.7
Small-2	Orders	2.0	492.3
Medium-1	Partsupp	5.0	731.3
Medium-2	Lineitem	1.0	1224.8
Large-1	Partsupp	13.0	1902
Large-2	Lineitem	2.0	2448.5

Table 5: Table Sizes (MMS experiments)

- the relative performance of MMS vs. Postgres' built-in sort (PPMS) over tables of varying sizes,
- the impact of Block-Level I/O Grouping on the performance of MMS, and
- the impact of Block Recycling on the performance of MMS.

We start by comparing the performance of four sort algorithms:

- PPMS: Postgres's Polyphase Merge Sort [7] (which is lazy and performs block recycling)
- PPMS-NR: PPMS modified to do no block recycling
- MMS: as described in Section 3, and
- MMS-NR: MMS modified to do no block recycling.

These four algorithms cover the space of possibilities given the choices between eager (MMS) vs. lazy merging (PPMS), and block recycling or no block recycling (-NR), as shown in Table 4. All sort algorithms were configured to use the same amount of *sort memory* that is allotted by default to Postgres' PPMS (i.e., 1 MB)[5].

In Section 4.1.1, we compare the performance of the sort operators above on TPC-H tables ranging in size from .4 GB - 2.4 GB. In Section 4.1.2, we show how eager sort algorithms get better I/O buffer behavior than lazy sorting algorithms (thus proving the effectiveness of block-level I/O grouping). Then in Section 4.1.3, we show how block recycling algorithms result in fewer I/O's (though sometimes poorer performance) than non-block recycling algorithms.

4.1.1 Performance Results

Figure 6 compares the execution time of the four sort algorithms described previously, as well as two additional variations of MMS: MMS (1) and MMS (2). These sort algorithms were added to the experiments, because results showed that for medium to large files, algorithms that do block recycling (PPMS and MMS) performed worse than corresponding algorithms that do not (PPMS-NR and MMS-NR). We speculated that this had to do with the fact that block recycling results in discontiguous allocations of blocks to runs, and that for

[5]The *sort memory* specifies the amount of private memory each sort (or hash) operation in Postgres can have. If there are n simultaneous sorts (or hashes), the total sort memory will be n times the default. Therefore, common wisdom dictating that the number should be kept fairly small to avoid swapping. A detailed discussion about this issue can be found in [8].

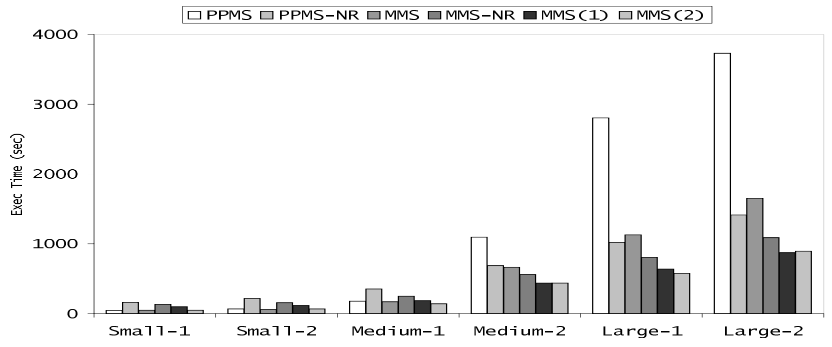


Figure 6: A Comparison of Execution Times for Six Sorting Algorithms

very large runs, this results in poor scan performance due to excessive seek movement. We tested this theory by looking at hybrid algorithms that perform block recycling only for small runs (i.e, only during the level0 and level1 merges (`MMS (1)`) or during the level0, level1 and level2 merges (`MMS (2)`)[6]. For all six tables, the `MMS`-based algorithms consistently performed better than the `PPMS`-based algorithms. For example, for very large tables (Large-2), `MMS (1)` is over 400% faster than `PPMS`.

`MMS-NR` consistently performs better than `PPMS-NR` over all size tables. The reason is that `MMS-NR` exploits the LRU-based page replacement policy of Linux, and thereby gets a higher I/O buffer hit ratio. For the same reason, `MMS` performs better than `PPMS`.

When the table being sorted is smaller than the I/O buffer (Small-1, Small-2 and Medium-1), `PPMS`, `MMS` and `MMS (2)` all perform much better than `PPMS-NR` and `MMS-NR`. This is because these algorithms perform block recycling.[7] `MMS-NR` and `PPMS-NR` do not recycle blocks from old runs and instead write new runs to new disk blocks. Every time they write a new disk block, an I/O buffer page needs to be allocated to buffer the disk block. Although the table size is less than the I/O buffer, the same amount of data is added to the I/O buffer at each merge level. Once the number of I/O buffer pages used to cache the output exceeds the buffer flushing threshold, the I/O buffer flushes pages to disk to obtain more clean buffer pages. By flushing pages they no longer need to disk, these algorithms perform more writes than needed. Also, after the I/O buffer is flushed, run blocks which will be needed again soon are replaced by newly created run blocks and need to be read back from disk, further increasing the number of disk I/O's required.

As tables get large, the cost of `PPMS` increases dramatically. This is because block recycling results in intermediate sort runs that are highly discontiguous. For large files, this results in poor I/O performance, as writing and reading those runs require excessive disk head (seek) movement. For the same reason, `MMS` does not do well for very large files. `MMS-NR` and `PPMS-NR` always write runs to new blocks, thereby storing intermediate runs contiguously. Therefore they take less time to finish, even though they read and write more data. We will compare the actual I/O's executed by these algorithms in the following section.

As we speculated, `MMS (1)` and `MMS (2)` performed best when sorting large tables. Compared to `PPMS` and `MMS`, these algorithms access data sequentially after the first few merge levels and thereby avoid excessive random disk I/O. Compared to `MMS-NR`, these algorithms recycle blocks in the first few merge levels and reduce the number of unnecessary writes.

4.1.2 Exploiting LRU-Based Page Replacement

In this section, we compare the read requests and actual data read by each of the sorting algorithms to test the hypothesis that eager sort algorithms such as `MMS` benefit from exploiting LRU-based page replacement, and therefore needs to perform fewer actual I/O's to read data from disk. We tested this measure by estimating the I/O buffer hit ratio for every run described in Section 4.1.1.

For these experiments, we used the Linux tools STRACE and IOSTAT to log all system calls and disk I/O's. The read requests comes from STRACE. We use STRACE to trace all read system calls of the process and sum up their request sizes. The actual I/O calculations come from IOSTAT. We use it to report the blocks read from disk as the query is processed. The actual I/O percentage is the actual I/O divided by the read requests. We compare the I/O requests and actual I/O percentages of the sorting algorithms in Table 6. Note that because these tools measure the *amount* of data requested, and actually transferred to/from disk, the "actual I/O percentage" results serve as an estimate of the I/O buffer miss ratio rather than an actual measure of such.

For small tables (Small-1, Small-2 and Medium-1), the actual I/O percentage for `PPMS-NR`, `MMS-NR` and `MMS (1)` is higher than that for the other sort algorithms. This is because these algorithms do not perform disk block recycling (`PPMS-NR` and `MMS-NR`) or do not perform enough disk block recycling to make a difference to performance (`MMS (1)`). Merging adds new blocks to the I/O buffer, which causes blocks which will be used soon to be evicted from the I/O buffer. These run blocks are read back later in the merging phase, which results in more actual I/O's and a higher actual I/O percentage. `PPMS` reads less data because it performs disk block recycling. By doing so, `PPMS` is able to use a limited set of buffer pages to cache all the runs it needs when table is small. The number of buffer pages it requires to cache runs is the table size.

For larger tables (Medium-2, Large-1, and Large-2), `MMS-NR` and `MMS` read much less data than `PPMS` and `PPMS-NR`. This is because they all exploit the LRU-based page replacement

[6]Due to time constraints, we were not able to complete experiments for `MMS (3)` or `MMS (4)`. (For the largest tables we sorted, `MMS (5)` = `MMS`.)

[7]The block recycling benefits of `MMS (1)` are negligible for small tables, given that block recycling is performed only for a single merge level.

Sorted Table	Read Requests (MB)				Actual I/O Pct			
	PPMS	PPMS-NR	MMS-NR	MMS	PPMS	PPMS-NR	MMS-NR	MMS
Small-1	2641	2641	2671	2673	16.7%	62.7%	41.3%	16.5%
Small-2	3070	3070	3004	3010	18.5%	72.5%	41.3%	16.4%
Medium-1	4641	4641	4485	5222	27%	88.4%	46.9%	15.2%
Medium-2	8225	8225	8030	8745	63%	92.2%	57.7%	36.4%
Large-1	13548	13548	13872	13871	100%	100%	56.9%	40.2%
Large-2	17743	17743	17775	17827	100%	100%	59.3%	44.1%

Table 6: Percentage of the Actual Data Read and Read Requests

Sorted Table	Write Requests (MB)						Actual I/O Pct					
	PPMS	PPMS-NR	MMS-NR	MMS (1)	MMS (2)	MMS	PPMS	PPMS-NR	MMS-NR	MMS (1)	MMS (2)	MMS
Small-1	2202	2202	2201	2201	2201	2201	4.1%	90%	92.2%	52.3%	22.4%	24%
Small-2	2578	2578	2476	2476	2476	2478	47.2%	66%	91%	51.2%	25.8%	24.5%
Medium-1	3910	3910	3669	3669	3669	4401	77.6%	100%	100%	56.6%	33.4%	27.1%
Medium-2	7001	7001	7338	7338	7338	7341	100%	100%	100%	66.8%	50.2%	46.9%
Large-1	11646	11646	11442	11443	11443	11443	100%	100%	100%	66.5%	50.2%	46.1%
Large-2	15294	15240	14673	14680	14673	14680	100%	99%	100%	66.8%	50.9%	50.3%

Table 7: Percentage of the Actual Data Written and Write Requests

policy of Linux, and access run blocks from the I/O buffer before they are evicted. When the table size is larger than the I/O buffer, PPMS is not able to cache all of its runs in the I/O buffer.

With large tables, PPMS run blocks have to be written to disk before they are used again. MMS reads the smallest amount of data in all six cases because it maximizes the buffer hit ratio by both grouping block requests and recycling blocks. MMS-NR, MMS, MMS (1) and MMS (2) differ only by their block recycling level and their relative performance confirms that more block recycling results in smaller actual I/O percentages.

Note that when tables get large, the actual I/O percentage does not correspond to the execution time. For example, the actual I/O percentage of PPMS-NR is about the same as PPMS but its execution time is much smaller. This is the effect of the random disk I/O's introduced by recycling blocks, which was discussed in Section 3.

4.1.3 Exploiting Delayed Buffer Flushing

MMS exploits delayed buffer flushing and prevents dirty pages which are no longer required from being written to disk by performing disk block recycling. This results in less data being actually written to disk. Table 7 compares the actual I/O percentage of the sorting algorithms. Again, we use STRACE and IOSTAT to get the write requests and the actual I/O, tracing the write system calls and actual block written to disk.

Compared to PPMS, PPMS-NR has a higher actual I/O percentage for small tables. This is because PPMS is able to cache all of the runs it needs in the I/O buffer and perform disk block recycling effectively. When the table gets large, PPMS-NR writes the same amount of data as PPMS does. For MMS-NR, MMS (1), MMS (2), and MMS, their actual I/O percentages reflect how much block recycling they do (more block recycling leads to lower actual I/O percentages).

4.2 Sort-Hash Join

In this section, we compare the execution times of HHJ and SHJ measured over joins of tables from TPC-H with various scale factors. In Sections 4.2.2 and 4.2.3, we show how SHJ benefits from performing I/O ordering and file-level block grouping to exploit the file system practices of I/O merging and block grouping.

	Scale	Table Size (MB)	
	Factor	Supplier	Partsupp
Small-1	2.0	4	292
Small-2	3.0	6	439
Medium-1	6.0	12	877
Medium-2	9.0	18	1316
Large-1	13.0	26	1902
Large-2	14.0	28	2048

Table 8: Table Sizes (SHJ experiments)

4.2.1 Performance Results

Table 8 lists the sizes of tables joined in our experiments. We chose the Supplier and Partsupp tables from TPC-H, varying their sizes by changing the DBGEN scale factors. Those two tables are chosen because the Supplier table, which is the build relation and therefore it determines the number of partitions, is not large in comparison to the Postgres sort memory. We have done experiments on larger tables with larger number of partitions, and for these, SHJ still outperforms HHJ but requires more memory.

Figure 7 compares the performance of SHJ with HHJ. For small tables, SHJ does slightly worse than HHJ. This is because when the tables are small, the I/O buffer is sufficiently large that neither of the two algorithms actually read partition files from disk. Therefore, SHJ is not able to get any benefit from I/O merging or block grouping, but incurs CPU overhead because it sorts tuples, thereby writing more data to and reading more data from the I/O buffer than HHJ. As tables get larger, partition files are actually written to disk and read back, and the savings SHJ gets from faster disk I/O outweighs the CPU overhead it incurs. Therefore, for large tables SHJ performs better than HHJ. But when the build relation gets too large (Large-2), more partition files must be created and in this case, SHJ is not able to benefit from block grouping and starts to perform worse than HHJ.

4.2.2 Exploiting I/O Merging

I/O merging produces fewer but larger I/O requests. We tested to see if SHJ benefits from I/O merging by comparing the average size of I/O requests for SHJ and HHJ. If the aver-

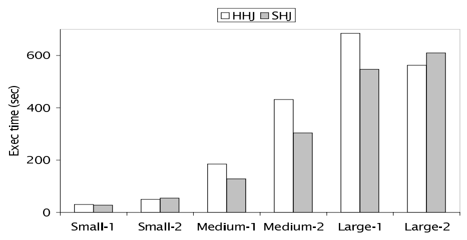


Figure 7: Execution Time of Joining Tables Using `HHJ` and `SHJ` (memory = 1MB)

age size of an I/O request differs for the two algorithms over the same input, this implies that I/O merging was more successful for the algorithm with the higher average I/O request size.

For this experiment, we used a disk request tracing tool, DCM, to trace all I/O requests of a particular partition sent to the IDE disk controller. DCM reports the type (read or write), location and size of each I/O request. We average the size of all requests reported during query processing.

Table 9 compares the average size of I/O requests for `SHJ` and `HHJ`. We used one table from each category (Small-1, Medium-1, Large-1) in our test set. The average I/O size of `SHJ` is greater than that of `HHJ` in all three cases.

It is worth mentioning that for table, Large-1, the average read size of `SHJ` and `HHJ` is about the same, but the average write size of `SHJ` is larger than that of `HHJ`. This benefit comes mostly from the ordered and better merged write requests.

Table	Actual I/O's		Average I/O Size (KB)	
	`HHJ`	`SHJ`	`HHJ`	`SHJ`
Small-1	7130	5061	57.7	60.3
Medium-1	44376	37654	42.3	49.9
Large-1	179683	172835	32.0	33.5

Table 9: Average Sizes of I/O's of `HHJ` and `SHJ`

4.2.3 Exploiting Block Grouping

In this section, we present experiments that determine how well `SHJ` performs file-level I/O grouping. In an extent-based file system, each file is stored on disk with one or more *contiguous regions*, such that a contiguous region is a set of one or more block groups from the same file that are adjacent on disk. The number of contiguous regions of a file reflects the minimal number of seeks required to access the file sequentially. The basic idea is that file-level I/O grouping results in block groups assigned to a file tending to themselves be contiguous, leading to a file that is broken up into a smaller number of larger contiguous regions, rather than a larger number of smaller contiguous regions.

We assume for this experiment that the distance on disk between contiguous regions is independent of the size of contiguous regions, and therefore, a file broken up into fewer, larger contiguous regions will be faster to scan than a file of the same size that is broken up into more, smaller contiguous regions. (The seek distance between 1st and last block of a file will be less in the former case.) Thus, our experiments test to see which of `SHJ` or `HHJ` creates hash partitions consisting of larger (on average) contiguous regions.

We trace the blocks allocated to each file with an ext2 inode tracing tool of our own. Once the tracing module

Table	`HHJ`	`SHJ`
Small-1	8	48
Medium-1	8	18
Large-1	8	8

Table 10: Contiguous Region Sizes of `HHJ` and `SHJ`

is loaded, every time the file system allocates a new block to an inode, the block number and the inode number are logged together. Block numbers are mapped directly to logical block addresses (logical block address = block number × 8). We then compute the average contiguous region size by dividing the total blocks used for all partition files by the total block group number in all partition files.

We chose one table from each category in our test set. The average block group size of `HHJ` and `SHJ` in our experiments are shown in Table 10. The average block group size of `HHJ` is 8, which is the block group size of the file system. For Small-1 and Medium-1, `SHJ` is able to perform file-level I/O grouping effectively and stores files in blocks groups six and twice the system default size respectively. As a consequence, the I/O requests of sequentially retrieving the files created with `SHJ` can be better merged. This has been verified in the previous section by the average request size.

4.3 Discussion

The results above demonstrate the effectiveness of permuting and refining (with block recycling) block request sequences in a manner that exploits the underlying Postgres and Linux buffer and file system management policies. This is the first step in showing that block request sequence refinement can be used to improve the I/O performance of query processing assuming the DB buffer and file system management policies are fixed.

The effectiveness of permuting block requests inside a plan operation is limited by both the amount of resources the plan operation possesses (e.g. private memory) and the semantics of the plan operation (e.g. hash join has to partition both relations before joining them). But the good news is from outside, we can force plan operations to produce refined block request sequences. Take hash join as an example, `SHJ` intends to refine its block request sequence by partially sorting tuples by their partition numbers so as to group block requests at file-level. However, if the tuples a hash join operation receives are "grouped" by their partition numbers already, it is able to produce file-level grouped block requests even without sorting. Therefore, refining the block request sequence of a hash join operation can also be done by supplying it tuples that are "grouped". There are many ways to produce grouped tuples, e.g. blocks of a relation can be retrieved based on their contents so that tuples appear to be grouped. Without the restrictions imposed by plan operations, block request sequence refinements can be done more effectively and will bring more substantial benefits.

5. RELATED WORK

Database management systems are implemented on top of operating systems. Stonebraker studied the operating system supports for database systems basing on *UNIX* and *Ingres* [13]. He argued that many operating system supports are slow or inappropriate for database systems and suggested a small efficient operating system be built for database systems. In contrast, we approach this issue from the opposite direction – changing the plan operations to exploit the practices used by operating systems.

Our work resembles in spirit, the *cache-aware query processing* work done independently by Ailamaki [1], Ross [16] and others. Like cache-aware query processing, our work is about formulating query processing techniques that exploit the characteristics of the underlying hardware or software system over which the DBMS operates.

Some DBMS' storage managers control their own disk I/O's. (e.g. *Larchesis* [11].) Block request sequence refinements could potentially exploit the policies used by such storage managers just as we have shown them to exploit the file system policies of Linux. Thus, we believe this work complements the research on DB-specific storage management.

DB2 sorts block reads in its list prefetching [4] to minimize the cost of disk seeks. Our I/O ordering and grouping techniques go further in two respects. First, we sequence writes of new files so that logically consecutive blocks (blocks belonging to the same file) are physically positioned in close proximity on disk (thus making future file scans cheaper). Secondly, we group writes of blocks across multiple files by disk location so as to minimize seek time during writing. Thus, DB2 reduces the cost of randomly accessing existing files, our I/O ordering and grouping techniques reduce the cost of writting new files as well as the cost of accessing them sequentially.

Graefe discussed "eager merge" and "lazy merge" for external merge sort in [6]. `MMS` chooses to perform "eager merge" for the purpose of grouping block requests instead of reducing the number of runs.

Patel et al. [10] noticed the interference between the reads of join relations and the writes of partition files in Hybrid Hash Join and built a more accurate model to estimate the cost of Hybrid Hash Join. But they did not consider the interference between the writes of different partition files. This kind of interference affects the partition file layouts and makes the cost of accessing partition files sequentially less predictable. `SHJ` reduces the interference between the writes of different files in a semantics-preserving way.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we propose to improve the I/O performance of query processing by permuting and refining the block request sequences of plan operations based on the underlying management policies of the DB buffer and the file system. We studied some common file system practices used in Linux: *LRU-Based Page Replacement*, *I/O Merging*, *Delayed Buffer Flushing*, and *Block Grouping*, and developed four block request refinement techniques: *Block-Level I/O Grouping*, *File-Level I/O Grouping, I/O Ordering,* and *Block Recycling.*

To manifest these techniques, we created two new plan operations, a variation of external merge sort, *Multilevel Merge Sort* (`MMS`), and a variation of hybrid hash join, *Sort Hash Join* (`SHJ`). Both of these operations adopt some of the block request refinement techniques above while correctly performing sorts and joins .

Experimental results show that the best sorting performance occurs with versions of `MMS` that perform block recycling during early merge levels, but not during later merge levels. This is shown to be due to the benefits of Block-Level I/O Grouping on I/O buffer performance, and the benefits of Block Recycling to exploit delayed buffer flushing when constructing short intermediate runs. We also show that `SHJ` outperforms `HHJ` in most cases, due to its use of I/O Ordering to exploit I/O Merging, and File-Level I/O Grouping to exploit Block Grouping. The only exceptions are for joining extremely small tables (where the I/O buffer is sufficiently large to make it unnecessary to flush hash table partitions to disk) and for extremely large tables (where too many partition files are created and block requests can not be grouped effectively).

Block request sequence refinement is a complementary approach to techniques that improve the management of the DB buffer and the underlying file system. We have demonstrated its viability as a technique for improving query I/O performance by showing the benefits of performing block request sequence refinement *inside* plan operations. In future work, we plan to refine block request sequence *outside* plan operations – forcing plan operations to produce refined block request sequences from outside. Without the limitation improsed by plan operations, more effective block request sequence refinement can be achieved. Therefore, as a way of improving query I/O performance, block request sequence refinement has the potential to bring even more substantial benefits.

7. REFERENCES

[1] A. Ailamaki. *Architecture-Conscious Database Systems: The PAX Storage Model.* PhD thesis, University of Wisconsin Madison, 2000.

[2] R. Card, T. Ts'o, and S. Tweedie. Design and implementation of the second extended filesystem. In *Proceedings of the First Dutch International Symposium on Linux*, 1999.

[3] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *VLDB*, pages 127–141, 1985.

[4] http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/index.jsp.

[5] G. Graefe. Parallel external sorting in volcano. Technical Report 459, Department of Computer Science, Univ. Of Colorado.

[6] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, 1993.

[7] D. E. Knuth. *The Art of Computer Programming, Vol 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.

[8] B. Monjian. Postgres performance tuning. *Linux Journal*, 88, 2001.

[9] G. Nutt. *Operating Systems.* Addison-Wesley, Reading, Mass., 2000.

[10] J. M. Patel, M. J. Carey, and M. K. Vernon. Accurate modeling of the hybrid hash join algorithm. In *Measurement and Modeling of Computer Systems*, pages 56–66, 1994.

[11] J. Schindler, A. Ailamaki, and G. R. Ganger. Lachesis: Robust database storage management based on device-specific performance characteristics. In *VLDB*, pages 706–717, 2003.

[12] A. Silberschatz, H. F. Korth, and S. Sudershan. *Database System Concepts, 4th edition.* Computer Science Series. McGraw-Hill, 2001.

[13] M. Stonebraker. Operating system support for database management. pages 168–174, 1994.

[14] M. Stonebraker, L. A. Rowe, and M. Hirohama. The implementation of postgres. *Transactions in Knowledge and Data Engineering*, 2(1):125–142, 1990.

[15] http://www.tpc.org.

[16] J. Zhou and K. A. Ross. Buffering databse operations for enhanced instruction cache performance. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 191–202, New York, NY, USA, 2004. ACM Press.