# Improved and Optimized Partitioning Techniques in Database Query Processing

Kjell Bratbergsengen and Kjetil Nørvåg

Department of Computer Science
Norwegian University of Science and Technology,
7034 Trondheim, Norway
{kjellb,noervaag}@idt.unit.no

**Abstract.** In this paper we present two improvements to the partitioning process: 1) A new dynamic buffer management strategy is employed to increase the average block size of I/O-transfers to temporary files, and 2) An optimal switching between three different variants of the partitioning methods that ensures minimal partitioning cost. The expected performance gain resulting from the new management strategy is about 30% for a reasonable resource configuration. The performance gain decreases with increasing available buffer space. The different partitioning strategies (partial partitioning or hybrid hashing, one pass partitioning, and multipass partitioning) are analyzed, and we present the optimal working range for these, as a function of operand volume and available memory.

Keywords: Relational algebra, partitioning methods, buffer management, query processing

## 1 Introduction

Relational algebra operations are time and resource consuming, especially when done on large operand volumes. In this paper, we present a new, dynamic, buffer management strategy for partitioning which can significantly reduce the execution time: *the circular track snow plow strategy*. Our approach is motivated from three observations:

1. When doing disk intensive operations, it is advantageous to process as large blocks as possible when doing disk accesses. Our strategy uses an optimal partitioning, which gives as large blocks as possible.
2. With the traditional fixed size block methods, only half the available memory is actually holding records. Our strategy, with variable size blocks, will with the same amount of available memory, double the average block size, and thus make much better use of available memory resources.
3. While it might be true that main memory on computers are large, and getting even larger, not all of this is available for one relational algebra operation. Often, several programs are running concurrently, several users are running queries concurrently, and queries can be quite complex, resulting in a large

tree of query operators. In this case, available memory for each operator can be rather small. As the comparison between the methods will show later in the paper, our method will be especially advantageous with small amounts of memory available, but it will always perform better than the traditional approach.

In the rest of the paper, we first present some related work in Sect. 2, and give an introduction to partitioning strategies in Sect. 3. The circular track snow plow strategy is introduced in Sect. 4. We present our cost model and assumptions in Sect. 5, and derive cost functions for the partitioning strategies. Finally, we compare these approaches in Sect. 6, and show a significant performance gain by using the snow plow strategy.

## 2 Related Work

Optimal splitting of source relations is discussed by Nakayama et. al. in [9]. Their conclusion is to partition the relation into as many partitions as possible. They say nothing about the block size, except that it is fixed. Their algorithms is beneficial to use when we have heavy data skew, but in other situations the small block size makes them expensive, as pointed out by Graefe [5]. Block size (clusters) has usually been determined from experiments, simulations, or just common sense, with Volcano [6] as an exception. Recently, the use of variable block size has also been recognized and studied by Davidson and Graefe [3], and Zeller and Gray [11], but these papers focuses on memory availability, rather than a thorough analysis by the use of cost functions.

## 3 Partitioning Strategies

With smaller operand volumes, nested loop methods are superior, requiring only one scan of the operands to create the resulting table. When the operand volume increases, nested loop methods are still employed in the final stage, but now after a hash partitioning stage, as described in [2, 1, 4].

Several partitioning strategies exists. They can be classified as *no splitting*, *partial, one pass, and multipass partitioning*. This first one, no splitting, is used when the smallest operand is less than or equal to available workspace. When this is the case, the whole operation can be done in main memory.

When the smallest operand is larger than available memory, one can split in one pass. All available workspace is used for splitting. We call this variant *one pass partitioning*. The number of partitions, $p$, is so large that each partition can be held in work space in the next stage.

When the smallest operand $V$ is larger than the memory $M$, but less than some limit $V_{ppu}$, partial splitting (or hybrid hash) can be employed. Part of the memory is used for partitioning, the other part is used for performing the relational algebra operation. When the operand gets larger, more work space

area is needed for splitting. The upper limit $V_{ppu}$ is found when all available memory is best used for splitting the operand.

As the smallest operand gets very large, a large number of partitions is necessary when one pass partitioning is employed. The result is a small block size, because the block size $b = M/p$ (or, as we will see later, with our method, $b = 2M/p$). As this block size gets smaller, there is a limit where splitting is best done in several passes, *multipass partitioning*.

It is useful to classify the (smallest) operand size, relative to available workspace $M$, in four classes: small, medium, large, and huge operands. As is shown in Table 3, partitioning strategy is determined from operand class. How to decide the bounds for each class will be shown later in the paper. As pointed out in the introduction, it is important to keep in mind that not all of the main memory is available as workspace for the partitioning process.

| Operand Class | Size of Smallest Operand | Strategy |
|---|---|---|
| Small | $V \leq M$ | No partitioning |
| Medium | $M < V \leq V_{ppu}$ | Partial partitioning |
| Large | $V_{ppu} < V \leq V_{opu}$ | One pass partitioning |
| Huge | $V_{opu} < V$ | Multipass partitioning |

**Table 1.** Operand classes and corresponding partitioning strategies.

## 4 The Circular Track Snow Plow Strategy

With the traditional splitting strategy, as described in [2, 1, 7, 4, 10], we have a fixed size of memory for each group. When a new tuple arrive, it is moved to its block buffer. Whenever a block buffer is full, it is written to disk.

If we look closer at the fixed block size splitting, we see that only about half the available memory is actually holding records. If we do not divide the memory into fixed sized block buffers, but rather let work space be one common memory pool, we do not have to write records to disks until all memory is taken. Then we write the first group to disk. We now have a new period where all groups (part of partitions in memory) are growing at approximately the same speed. This holds if the hash partitioning formula gives an even distribution. Again when there is no room left, the next group is written to disk. After all groups have been written once to disk, we start over again with the first group. After a transient start up phase, we can see that the average size of the groups written to disk is approximately $b = 2M/p$. This method is analogous to the replacement selection sort which is used for initial sorting in sort-merge programs. Why the average block size is $2M/p$, is well described by Knuth in [8]. The situation can be compared to a snow plow on a circular track of length $l$. The plow is always plowing full depth snow. Just behind the plow, the depth is zero, and the average snow depth on the track is one half the full depth $h$. The total amount of snow on the track is $hl/2$. In our case, the full snow depth $h$ corresponds to the block

size $b$, and the track length $l$ corresponds to the number of subfiles $p$. Then $bp/2 = M \Rightarrow b = 2M/p$. This is illustrated in Fig. 1, which shows the groups in memory during splitting. To the left we see the situation just before the first group (0) is written to disk. To the right we see a steady state situation just after group 3 has been written to disk. The next group to go is number 4, when the lower limit of available space is reached.

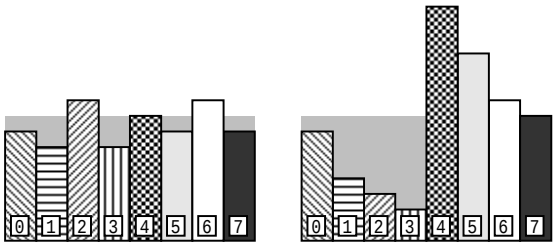


**Fig. 1.** Groups in memory during splitting.

### 4.1 Memory management

The memory is now used as a heap, storing records. The records are logically separated into $p$ groups. This can be done using linked lists, pointer arrays, linked lists with pointers, etc. The group is determined using a hash formula on the operand key. Records are read and stored in memory until the amount of free memory reaches a lower limit. In the stationary situation, the amount of memory used is the same as the amount released. The addresses of the free space slots could be stored on a small stack. Memory management is especially simple if all records have the same size, however, if there are variable length records, more elaborate schemes should be used. Memory management at this level is important, because it could take a lions share of CPU time.

### 4.2 Writing Blocks to Disk

To take full advantage of this method, each group should be written to the disk as one block. Because of the stochastic nature of the group size, groups do not in general have the same size. Also, in the initialization phase, the average group size is rapidly growing from $M/p$ to $2M/p$.

Despite the groups in memory are of different sizes, each group could be stored as a chain of fixed sized blocks on the disk. We should set aside two buffers to allow for double buffering. Each buffer should be $2M/p$ bytes. When a group is ready for output, its records are moved to the free output buffer. In the startup transient phase, we would not be able to fill the buffer before we

have to write it, but in the stationary phase most of the time it will be nearly full. Sometimes we have to leave some records of the group behind, they will have to go to the next block for that group. To better fill the output buffer, we could change the *round robin* sequence strategy for writing groups to disk, to a *largest group first strategy* (LGF). This strategy might even lead to a larger average block size than $2M/p$.

It would be even better if we could get user level functionality enabling us directly to write to, and read from, the SCSI port. This requires some redesign of the ASPI interface. The current ASPI interface provides only a traditional block transfer I/O command. The command specifies a block address, buffer address and block length. No constructive interaction is possible until the command has finished, i.e. transferred a complete block. We would need new functionality, like moving single words or smaller blocks between SCSI port and user memory. This is effectively a *gather write* function available at the user level. This could save a lot of unnecessary copying.

### 4.3 Disk Layout

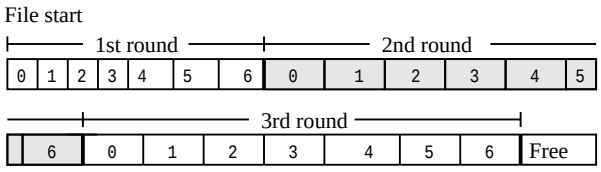


**Fig. 2.** The disk image after all groups have been emptied three times.

If we write the disk with fixed sized blocks, there is no problem finding all blocks of one group, they are chained together. The following is about disk layout when the disk is written with variable sized blocks (groups). When the groups are written to disk, we will get a pattern similar to that shown in Fig. 2, if neighboring groups are written contiguously onto disk. A *Round* in the figure is *one round around the circular track*, processing each group once. The numbers are group numbers. Writing groups contiguously gives a bonus when we flush all groups at end of input. This can be done in one operation. It is also necessary if we want to create *supergroups*, by joining neighboring groups into one supergroup. This is useful when the operand(s) has been over-partitioned (the operand has been partitioned into a larger number of partitions than necessary). This can happen if we are uncertain about the number of subfiles we need, and choose to overpartition to err on the right side. The overpartitioning causes more disk accesses than necessary during the splitting phase. The extra cost of overpartitioning should be removed from the reading phase, and we can

| Round | Address | g0 | g1 | g2 | g3 | g4 | g5 | g6 |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 10 | 12 | 14 | 15 | 17 | 18 | 21 |
| 1 | 107 | 19 | 23 | 22 | 20 | 21 | 22 | 22 |
| 2 | 256 | 21 | 21 | 18 | 20 | 21 | 20 | 21 |
| 3 | 398 | 22 | 19 | 23 | 17 | 15 | 12 | 9 |
| 4 | 515 | 7 | 3 | 1 | 0 | 0 | 0 | 0 |
| Size | 526 | 79 | 78 | 78 | 72 | 74 | 72 | 73 |

**Table 2.** Index table, where the last row holds accumulated values (the size) for each group in number of address units.

do so by joining neighboring groups into supergroups. This is easy as long as all groups are written contiguously onto the same disk.

Because of the variable block size, we need an index to efficiently retrieve blocks when we read back one or more groups. The index is a table with one column for each group (subfile), and one row for each round in the Round Robin output process. The corresponding table element would then contain the size of each block. The size of the index table depends on the number of groups and the number of rounds. To speed up address calculations, it would be useful to add one column to the index table, holding the address of the first block in each round. It is shown in Table 4.3 how an index table might look like.

The size of each block could be in bytes, sectors or some other unit. The total size of each subfile after partitioning is found by adding the numbers in each column. This information will tell how many subfiles can be read together, or how much space is needed to hold the largest subfile during the final algebra operation.

Even for large data volumes, the index table should be stored in memory without eating up to much space: Suppose that for every round, approximately $2M$ data is written to disk. The size of the index table, in number of variable sized blocks is $s = \frac{Vp}{2M}$. If $V = 1$ GB, $p = 100$ and $M = 50$ MB, the size $s$ is 1000 integers.

### 4.4 Estimation of the Free Space Limit

The free space limit should be as small as possible, however there are some constraints. If we have only one SCSI bus, we have no real I/O overlapping. An initiated I/O operation must be completed until another operation could be started. If we started a read operation and could not complete it because of lack of input buffer space, we would have a deadlock. Then we would not be able to free space by writing a group to disk. Before we start a read, we should make sure that we have enough free space to accommodate all data read.

### 4.5 The Minimum Block Size

Writing a group should be skipped if it is below a minimum size. This will happen only when the hashing formula works poorly, producing uneven sized groups, or

we try to stress the system, having uncoordinated parameters. The minimum block size should be in the vicinity of 4 KB, and this should be far below the normal group size when M has some reasonable value (above 100 KB, at least). If we use the LGF strategy there will be no minimum block size problem.

### 4.6 Multiuser Environments

It is also worth noting that the size of $M$ can change dynamically, this only affects the size of the groups. Partitioning strategy, however, is done at operation startup time. Thus, if available memory decreases, an extra pass might be needed.

## 5 Cost Functions

Our cost model is based on I/O transfer only. This is the most significant cost factor, and in reasonable implementations, the CPU processing should go in parallel with I/O transfer making the CPU cost "invisible". Our disk model is traditional. Disk transfers are done blockwise. One block is the amount of data transferred in one I/O command. The main cost comes from two contributors: *the start up cost* and *transfer cost*. Start up cost comes from command software and hardware overhead, disk positioning time, and disk rotational delay.

In our model, the average start up cost is fixed, and is set equivalent to $t_r$, the time it takes to do one disk revolution. The transfer cost is directly proportional to the block size, and is equivalent to reading disk tracks contiguously, e.g., transfer cost is equal to $\frac{b}{V_s}t_r$, where $V_s$ is the amount of data on one track and $b$ is the block size to be transferred. For very large blocks, it is likely that several tracks and also tracks in different cylinders are read contiguously. This implies positioning, but we assume that the times used for this is insignificant compared to transfer time. The time it takes to transfer one block is $t_b = t_r(1 + \frac{b}{V_s})$. The time to transfer a data volume $V$ using block size $b$ is:

$$T_T = \frac{V t_r}{b} + \frac{V t_r}{V_s} = V t_r \left( \frac{1}{b} + \frac{1}{V_s} \right) \tag{1}$$

Our emphasis is to find optimal parameters and the best working range for each method. To be able to handle the different equations mathematically we regard them as continuous functions rather than discontinuous functions resulting from applying ceiling and floor functions. This approximation gives a better overall view, but could cause small errors compared to the exact mathematical description.

To get smooth input and output streams, double buffering can be employed. However, to simplify the computations, the space for input buffers and extra output buffers is not counted within the memory $M$ in our cost functions. Thus, the memory $M$, can be thought of as the available memory *after reservation of memory for the extra input and output buffers*.

The splitting is based on a hash formula applied to the operation key. The subtables will vary in size due to statistical variations and the "goodness" of the

selected hash formula, in a real system it would be beneficial to let the average subtable be slightly smaller than the workspace. With good approximation we can ignore this and we are also ignoring space needed for structural data.

### 5.1 One Pass Partitioning

We will start with a simple partitioning strategy. If the input table is larger than available work space, we will split the table in subtables, such that each subtable fits into work space $M$. The subtables are are written to temporary files. The necessary split factor is: $p = \lceil \frac{V}{M} \rceil$, and the time used is:

$$T_{1p} = 2T_T = 2Vt_r\left(\frac{1}{b} + \frac{1}{V_s}\right) = 2Vt_r\left(\frac{V}{M^2} + \frac{1}{V_s}\right)$$

The actual block size with fixed block size, assuming we are free to select the block size, is $b = \lfloor \frac{M}{p} \rfloor \approx \frac{M^2}{V}$, which gives:

$$T_{1p}^F = 2Vt_r\left(\frac{V}{M^2} + \frac{1}{V_s}\right)$$

and with variable block size, we get on the average $b = \frac{2M}{p} \approx \frac{2M^2}{V}$, which gives:

$$T_{1p}^V = 2Vt_r\left(\frac{V}{2M^2} + \frac{1}{V_s}\right)$$

For operand volumes slightly larger than $M$, we see that it would probably be better to let some of the work space be used for holding records participating in the final algebra stage, and use only a portion of the work space for split buffer. This leads us to the partial partitioning or hybrid hash algorithm. At the other end, we see that when operand volumes are very large, the split factor increases and the block size goes down. Small I/O blocks are severely slowing down the I/O process. It may be better to split the data repeatedly, using larger blocks, rather than using smaller blocks and reaching the correct subfile size in one pass. This leads us to the multipass splitting method.

### 5.2 Partial Partition

Partial partitioning is especially advantageous for operand volumes larger than $M$, but so small that we do not need all available memory for efficient splitting. A part of the memory is used for holding a number of complete groups, to avoid the unnecessary I/O caused by the splitting. If the operand volume $V$ is known in advance, we can compute the optimum memory size $x$, not used for splitting. $x$ is thus also the amount of data which is not split. We ignore floor and ceiling functions to keep the equations mathematically tractable.

**Fixed Block Size.** The number of subfiles is $N = (V - x)/M$. The complete split is done in one pass, hence $N = p$, the split factor. The average block size using fixed block size splitting is:

$$b = \frac{(M - x)}{p} = \frac{M(M - x)}{V - x}$$

which gives the split time by substitution of $b$ into Eq. 1 is:

$$T_{pp}^F = 2(V - x)t_r\left(\frac{1}{b} + \frac{1}{V_s}\right) = 2(V - x)t_r\left(\frac{V - x}{M(M - x)} + \frac{1}{V_s}\right)$$

The time will vary with $x$, a large $x$ reduces transferred volume, however, the average block size is decreased, and the time spent may increase. To find a minimal value for $T_{pp}$, we have to find the value of $x$ that will give the minimum value of $T_{pp}$. This can be done by solving:

$$\frac{dT_{pp}^F(x)}{dx} = 2t_r\left(\frac{-2(V - x)M(M - x) + M(V - x)^2}{M^2(M - x)^2} - \frac{1}{V_s}\right) = 0$$

To solve this equation we substitute $\lambda = M/V_s$ and

$$z = \frac{V - x}{M - x} \tag{2}$$

Then we obtain the following equation in $z$ giving:

$$z^2 - 2z - \lambda = 0 \Rightarrow z = 1 \pm \sqrt{1 + \lambda}$$

Back substitution of $z$ into Eq. 2 gives:

$$x = \frac{zM - V}{z - 1}$$

$x$ should never be negative, which implies: $z > 0$ and $zM - V \geq 0$. This sets a restriction on $V$, which must be less than $zM$. It does not make sense to use partial partitioning when operand volumes are greater than $zM$. When $V \leq M$ , splitting is not at all employed. This limits the working range of partial partitioning to: $M < V \leq zM$.

**Variable Block Size.** Derivation in the case of variable block size is done the same way as for fixed block size. The only difference is that we are doubling the effective block size, which gives:

$$T_{pp}^V = 2(V - x)t_r\left(\frac{1}{b} + \frac{1}{V_s}\right) = 2(V - x)t_r\left(\frac{V - x}{2M(M - x)} + \frac{1}{V_s}\right)$$

### 5.3 Multipass Partitioning

Data are read from an input file and hashed into a number of subfiles. The actual block size with fixed size blocks is: $b = \lfloor \frac{M}{p} \rfloor \approx \frac{M}{p}$. The necessary number of subfiles at the end is $N = \lceil \frac{V}{M} \rceil \approx \frac{V}{M}$. Depending on the operand volume, it may be beneficial to partition the operand in several passes. This is equivalent to the merging used in sort-merge processing, and the approximate number of operand passes is $w = \log_p N$. When we substitute for $N$, we get $w = \log_p \frac{V}{M}$. The total amount of data read and written to disk to complete a partitioning and the final relational algebra step is $V_T = 2Vw$, which gives a total partitioning time of:

$$T_{mp} = V_T T_T = 2V t_r \log_p \frac{V}{M} \left( \frac{1}{b} + \frac{1}{V_s} \right)$$

Substituting for $\log_p \frac{V}{M} = \frac{\ln V/M}{\ln p}$:

$$T_{mp} = 2V \frac{\ln V/M}{\ln p} t_r \left( \frac{p}{M} + \frac{1}{V_s} \right) \tag{3}$$

An optimum block size exist because when we split into many subfiles in one pass, each subfile need an output buffer, hence they get smaller. We can minimize Eq. 3 by noting that the variable part of this expression is:

$$f(p) = \frac{1}{\ln p} \left( \frac{1}{b} + \frac{1}{V_s} \right)$$

To find a minimum value for $f(p)$, we derive $f(p)$ with respect to $p$, and the optimum value for p is when $f'(p) = 0$:

$$p(\ln p - 1) - \frac{M}{V_s} = 0$$

It is easily seen that $p$ is a function of the quotient $\lambda = \frac{M}{V_s}$:

$$p(lnp - 1) - \lambda = 0$$

This equation can be solved numerically, to find the value for $p$. Thus, the function for multipass partitioning with fixed block size is given by:

$$T_{mp}^F = 2V t_r \log_p \frac{V}{M} \left( \frac{p}{M} + \frac{1}{V_s} \right)$$

With the same method, we can find an optimum value for $p$ in the case of variable block size (where $b = \frac{2M}{p}$):

$$T_{mp}^V = 2V t_r \log_p \frac{V}{M} \left( \frac{p}{2M} + \frac{1}{V_s} \right)$$

As an example, Table 5.3 shows optimal split factors, block size, and number of passes for different sizes of memory, with operand volume held constant on $V = 1000$ MB and $V_s = 50$ KB.

| $\lambda = M/V_s$: ($M$ in MB:) | 1 (0.05) | 2 (0.1) | 5 (0.25) | 10 (0.5) | 20 (1.0) | 50 (2.5) | 100 (5) | 200 (10) | 500 (25) | 1000 (50) |
|---|---|---|---|---|---|---|---|---|---|---|
| Fixed block size: | | | | | | | | | | |
| Optimal $p$ | 3 | 4 | 6 | 8 | 12 | 23 | 37 | 63 | 129 | 226 |
| $b$ | 17 | 25 | 42 | 62 | 83 | 109 | 135 | 159 | 194 | 226 |
| $w$ | 9.01 | 6.64 | 4.63 | 3.66 | 2.78 | 1.91 | 1.47 | 1.11 | 1.00 | 1.00 |
| Var. block size: | | | | | | | | | | |
| Optimal $p$ | 4 | 5 | 8 | 12 | 20 | 37 | 63 | 108 | 226 | 400 |
| $b$ | 25 | 40 | 62 | 83 | 100 | 135 | 159 | 185 | 221 | 250 |
| $w$ | 7.14 | 5.72 | 3.99 | 2.31 | 1.66 | 1.28 | 1.00 | 1.00 | 1.00 | 1.00 |

**Table 3.** Optimal split factors, block size and number of passes for different sizes of memory.

| Strategy: | Partial | One Pass | Multipass |
|---|---|---|---|
| Range: | $M < V \leq zM$ (medium operands) | $zM < V \leq p_{mp}M$ (large operands) | $V > p_{mp}M$ (huge operands) |
| Fixed Size, $T^F$: $z = 1 + \sqrt{1+\lambda}$ $p(\ln p - 1) - \lambda = 0$ | $2(V-x)t_r\left(\frac{V-x}{M(M-x)} + \frac{1}{V_s}\right)$ | $2Vt_r\left(\frac{V}{M^2} + \frac{1}{V_s}\right)$ | $2Vt_r \log_p \frac{V}{M}\left(\frac{p}{M} + \frac{1}{V_s}\right)$ |
| Variable size $T^V$: $z = 1 + \sqrt{1+2\lambda}$ $p(\ln p - 1) - 2\lambda = 0$ | $2(V-x)t_r\left(\frac{V-x}{2M(M-x)} + \frac{1}{V_s}\right)$ | $2Vt_r\left(\frac{V}{2M^2} + \frac{1}{V_s}\right)$ | $2Vt_r \log_p \frac{V}{M}\left(\frac{p}{2M} + \frac{1}{V_s}\right)$ |
| Supporting values: $\lambda = M/V_s$ | $x = \frac{zM-V}{z-1}$ | | $w = max\left(1.0, \frac{\ln V/M}{\ln p}\right)$ |

**Table 4.** Cost functions for splitting.

### 5.4 Summary and Application of the Cost Functions

The cost functions and their working areas are summarized in Table 5.4. To demonstrate the relationship between the three variants, we have computed the partitioning times for different operand volumes. By using $V_s = 50$ KB, $t_r = 11$ ms, and $M = 1$ MB, we get the partitioning times as shown in Table 5.4.

From the table, we can see that for fixed size blocks, partial partitioning is best for volumes from 2 to 5 MB. Multipass is better when operands are larger than 12 MB. For variable blocks, partial partitioning is best for the volumes from 2 to 7 MB. Then one pass takes over, multipass partitioning is better when operands are larger than 20 MB. The different working areas for the variants of partitioning is illustrated in Fig. 3.

| | Fixed sized blocks | | | Variable sized blocks | | |
|---|---|---|---|---|---|---|
| | $\lambda = 20, z = 5.6, p_{mp} = 12$ | | | $\lambda = 20, z = 7.4, p_{mp} = 20$ | | |
| $V$ (in MB) | $T_{pp}$ | $T_{1p}$ | $T_{mp}$ | $T_{pp}$ | $T_{1p}$ | $T_{mp}$ |
| 2 | 0.69 | 0.98 | 1.39 | 0.61 | 0.93 | 1.33 |
| 3 | 1.39 | 1.53 | 2.19 | 1.22 | 1.43 | 2.00 |
| 4 | 2.08 | 2.13 | 2.92 | 1.83 | 1.96 | 2.37 |
| 5 | 2.77 | 2.78 | 3.65 | 2.44 | 2.50 | 3.33 |
| 6 | - | 3.47 | 4.38 | 3.04 | 3.07 | 4.00 |
| 7 | - | 4.20 | 5.11 | 3.65 | 3.66 | 3.67 |
| 8 | - | 4.98 | 5.84 | - | 4.27 | 5.34 |
| 16 | - | 12.8 | 12.7 | - | 9.96 | 10.7 |
| 32 | - | 37.0 | 31.7 | - | 25.6 | 24.7 |
| 64 | - | 119.5 | 76.1 | - | 74.0 | 59.2 |
| 128 | - | 421.0 | 177.6 | - | 238.9 | 138.2 |
| 256 | - | 1570.0 | 405.9 | - | 842.0 | 315.9 |

**Table 5.** Partitioning times for different operand volumes and different basic partitioning methods.

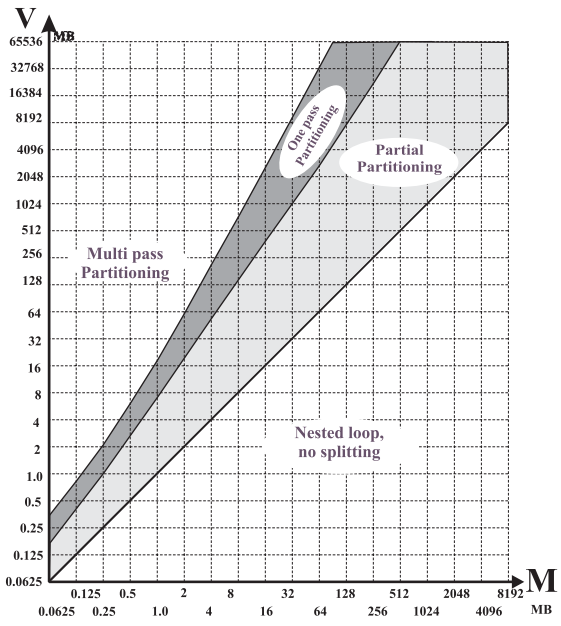


**Fig. 3.** Working areas for different algebra methods, with numbers from variable block size.

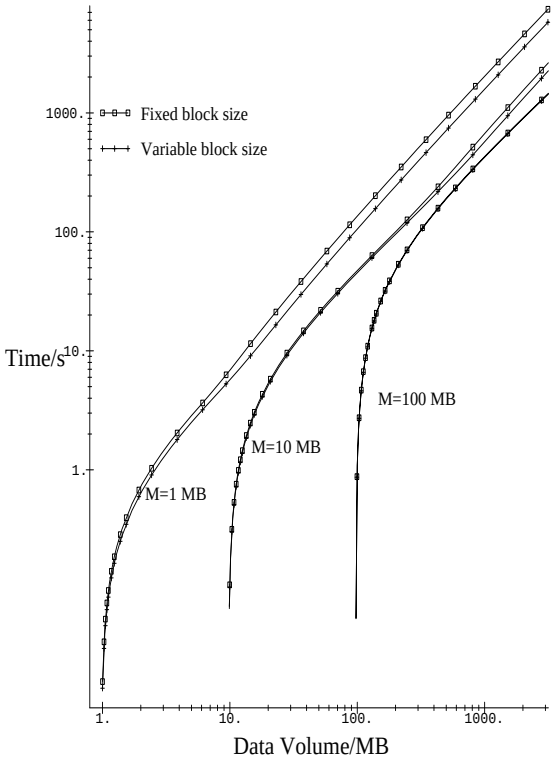


**Fig. 4.** Execution time with snow plow splitting.

## 6 Comparison

To compare the effect of increased average block size, we compare the traditional and new methods for three representative cases: $M = 1$ MB, $M = 10$ MB, and $M = 100$ MB. In Fig. 4 we have plotted the functions for variable and fixed block size. We have for each value of $V$ used the partitioning strategy (no/partial/one pass/multipass) that gives the best time. It is important to keep in mind that this is a log-log-plot, and therefore the difference in execution time between fixed and variable block size is not large in the figure. The improvement can be better illustrated by looking at the performance gain, which is computed in Table 6 and illustrated in Fig. 5. In the table, blank fields denotes no splitting.

As expected, the improvements are most noticeable for smaller work space areas. For larger work spaces, the number of blocks is relatively small, and the block access time is negligible compared to the transfer time.

| | $M$ = 1 MB, $\lambda$ = 20 | | | $M$ = 10 MB, $\lambda$ = 200 | | | $M$ = 100 MB, $\lambda$ = 2000 | | |
|---|---|---|---|---|---|---|---|---|---|
| V (in MB) | fixed | var | imp % | fixed | var | imp % | fixed | var | imp % |
| 1 | | | | | | | | | |
| 2 | 0.7 | 0.6 | 14 | | | | | | |
| 4 | 2.1 | 1.8 | 14 | | | | | | |
| 8 | 5.0 | 4.3 | 17 | | | | | | |
| 16 | 12.7 | 10.0 | 27 | 3.1 | 2.9 | 4.2 | | | |
| 32 | 31.7 | 24.7 | 29 | 11.3 | 10.8 | 4.2 | | | |
| 64 | 76.1 | 59.2 | 29 | 27.6 | 26.5 | 4.2 | | | |
| 128 | 177.6 | 138.2 | 29 | 60.4 | 58.0 | 4.2 | 13.0 | 12.8 | 1.3 |
| 256 | 405.9 | 315.9 | 29 | 128.3 | 121.1 | 6.0 | 72.5 | 71.6 | 1.3 |
| 512 | 913.2 | 710.8 | 29 | 285.8 | 256.7 | 11.3 | 191.5 | 189.2 | 1.3 |
| 1024 | 2029 | 1580 | 29 | 668.6 | 571.6 | 17.0 | 429.4 | 423.6 | 1.3 |
| 2048 | 4464 | 3475 | 29 | 1538 | 1314 | 17.0 | 905.4 | 893.2 | 1.3 |

**Table 6.** Performance improvement as a function of operand volume and workspace size.

## 7 Conclusions and Future Work

We have described a novel method for partitioning, the circular track snow plow buffer management strategy. This strategy makes more efficient use of memory compared to traditional methods, and effectively doubles the average block size. Doubling the effective block size can give substantial reductions in I/O transfer time, and by the use of cost models we have shown that performance gains of 10-20% can be expected for reasonable resource configurations.

In this paper we have made the assumption that the operand volume is known. This is not always true, and we are now investigating a dynamic or adaptive strategy, based on the snow plow principle, which is applicable when the operand volume is unknown. We are also working with the new buffer management strategy employed in other areas as file loading and transaction log.

In the development og this methods, we have also discovered how a more efficient set of routines for communication with the SCSI buss system could be used to avoid the unnecessary transfer of data to an internal block buffer. Direct transfer to the user program area can save the internal bus from considerable traffic. This is clearly interesting, and should be studied further.

## References


1. K. Bratbergsengen. Hashing Methods and Relational Algebra Operations. In *Proceedings of the 10th International Conference on VLDB*, 1984.
2. K. Bratbergsengen, R. Larsen, O. Risnes, and T. Aandalen. A Neighbor Connected Processor Network for Performing Relational Algebra Operations. In *Fifth Workshop on Computer Architecture for Non-Numeric Processing, March 11-14, 1980 (SIGIR Vol. XV No. 2, SIGMOD Vol.X No. 4)*, 1980.
3. D. L. Davidson and G. Graefe. Memory-Contention Responsive Hash Joins. In *Proceedings of the 20th International Conference on VLDB*, 1994.

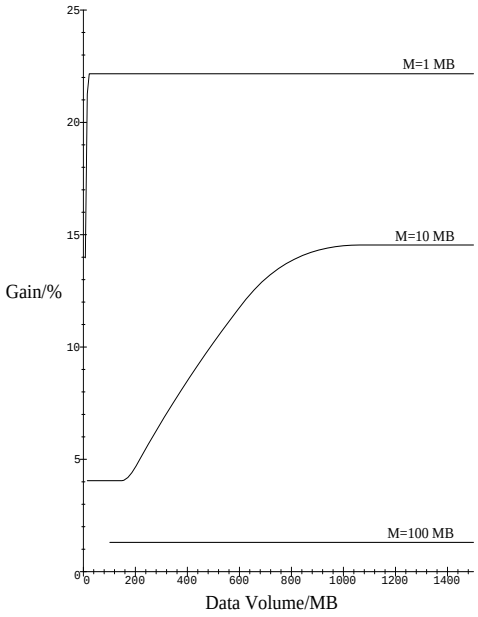


**Fig. 5.** Performance gain by using the snow plow strategy.


4. D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. In *Proc. ACM SIGMOD Conf.*, 1984.
5. G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2), 1993.
6. G. Graefe. Volcano — An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1), 1994.
7. M. Kitsuregawa, H. Tanaka, and T. Motooka. Application of Hash to Data Base Machine and its Architecture. *New Generation Computing*, 1(1), 1983.
8. D. Knuth. *The Art of Computer Programming. Sorting and Searching.* Addison-Wesley Publishing Company Inc., 1973.
9. M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-Partitioned Join Method Using Dynamic Destaging Strategy. In *Proceedings of the 14th International Conference on VLDB*, 1988.
10. L. D. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Transactions on Database Systems*, 11(3), 1986.
11. H. Zeller and J. Gray. An Adaptive Hash Join Algorithm for Multiuser Environments. In *Proceedings of the 16th International Conference on VLDB*, 1990.


This article was processed using the LaTeX macro package with LLNCS style