Grouping and Duplicate Elimination: Benefits of Early Aggregation

Per-Åke Larson
Microsoft Corporation

December 20, 1997

Technical Report
MSR-TR-97-36

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Grouping and Duplicate Elimination: Benefits of Early Aggregation

Per-Åke Larson
Microsoft Corporation

December 20, 1997

Abstract

Early aggregation is a technique for speeding up the processing of GROUP BY queries by reducing the amount of intermediate data transferred between main memory and disk. It can also be applied to duplicate elimination because duplicate elimination is equivalent to grouping with no aggregation functions. This paper describes six different algorithms for grouping and aggregation, shows how to incorporate early aggregation in each of them, and analyzes the resulting reduction in intermediate data. In addition to the grouping algorithm used, the reduction depends on several factors: the number of groups, the skew in group size distribution, the input size, and the amount of main memory available. All six algorithms benefit from early aggregation with grouping by hash partitioning producing the least amount of intermediate data. If the group size distribution is skewed, the overall reduction can be very significant, even with a modest amount of additional main memory.

1 Introduction

SQL queries containing GROUP BY with aggregation are common in decision support applications. Duplicate elimination can be viewed as a special case of grouping: GROUP BY without aggregation functions. A widely used algorithm for evaluating GROUP BY queries is to first sort the input records on the grouping columns and then perform aggregation on the sorted record stream.

The amount of data output from the run formation phase and carried through the merge phase can be reduced by a technique here called *early aggregation*. The basic idea is straightforward: when creating a run, maintain in memory a set of group records, one for each group seen so far; when an input record arrives, combine it with the matching group record if one exists, otherwise initialize a new group record. Combining an input record with a group record simply consists of updating the aggregation functions. When memory becomes full,

the group records are sorted and output as a run. The records carried into the merge phase then represent partially aggregated groups instead of individual input records. Early aggregation can also be applied during the merge phase by combining records from the same group whenever possible. Early aggregation always reduces the number of records processed during the merge phase. If the number of groups is small, all groups may fit in main memory and no merging is required.

Sorting is not the only way to evaluate GROUP BY queries. This paper describes how to incorporate early aggregation into six algorithms: an algorithm simply scanning the input repeatedly, an algorithm based on repeated union, two algorithms based on sorting, and two algorithms based on hash partitioning.

The effect of early aggregation is to reduce the amount of intermediate data transferred between main memory and disk. The main contribution of this paper is an analysis of the reduction resulting from early aggregation. The reduction depends on several factors: number of groups, the skew in the distribution of groups sizes, input size, the amount of main memory available and, of course, the grouping algorithm used. The analysis shows that the reduction can be very significant, even when using a modest amount of main memory.

2 Previous Work

Early aggregation is not a new idea. Most published work deals with duplicate elimination, typically in main memory. Munro and Spira[5] gave a computational bound for the number of comparisons required to sort a multiset with early duplicate removal. Several algorithms, based on various sorting algorithms, e.g., quick sort, hash sort and merge sort, have been proposed for duplicate elimination. Abdelguerfi and Sood[1] gave the computational complexity of the merge sort method based on the number of three-way comparisons. Teuhola and Wegner[7] gave a duplicate elimination algorithm based on hashing with early duplicate removal, which requires linear time on the average and O(1) extra space. Wegner[8] gave a quick sort algorithm for the run formation phase and analyzed its computational complexity.

However, we are mainly interested in large-scale grouping and aggregation requiring external storage. The processing cost is then dominated by the cost of I/O, and the CPU time can be largely ignored. D. Bitton and D.J. Dewitt [2] analyzed the benefits of early duplicate elimination during run merging in external merge sort. Their analysis is based on several simplifying assumptions: all groups are assumed to be of the same size, the only merge pattern considered is balanced two-way merge, and duplicate elimination during run formation is not considered. This analysis is also summarized in [3].

Parallel database systems running on shared-nothing systems normally perform aggregation in two steps: each node first performs grouping and aggregation on its local data and then ships the result to one or more nodes where the partial results are integrated. Shatdal and Naughton [6] pointed out that if the input is large and the duplication factor is low (few records per group), then the first step may do a lot of work for a relatively small reduction

in output size. If so, it is better to skip local aggregation and simply send the input tuples directly to the nodes performing the final aggregation.

A paper by Yan and Larson [9] contains some early results (based on a simulation study) of the benefits of applying early aggregation to grouping by sorting.

3 Preliminaries

In this section we derive three functions that will be needed when analyzing the various GROUP BY algorithms. Assume that the source records are divided among D different groups, labeled $1, 2, ..., D$. Let p_i denote the probability that a record belongs to group i. We call $p_1, p_2, \cdots, p_D$ the *group size distribution*. The actual group labels do not matter so we assume that the labels are assigned so that $p_1 \geq p_2 \geq \cdots \geq p_D$.

For the numerical results reported in this paper, we model the group size distribution with a generalized Zipf distribution [4]. The distribution function is defined by:

$$p_i = \frac{1}{c}(1/i)^\alpha, \quad i = 1, 2, ..., D$$

where α is a positive constant and $c = \sum_{i=1}^{D}(1/i)^\alpha$. Setting $\alpha = 1$ gives the traditional Zipf distribution, and $\alpha = 0$ gives a uniform distribution. Increasing α increases the skew in the group size distribution, which, as we will see, increases the data reduction obtained by early aggregation. Many phenomena, including the distribution of word occurrences in English text, have experimentally been found to follow a traditional Zipf distribution.

An input record will either be absorbed by a group already in memory or create a new group. Let N denote the number of records read so far. We model group labels as being independently and randomly drawn from the distribution $p_1, p_2, \cdots, p_D$. Then the expected number of distinct group labels occurring in a sample of N records equals

$$G(N) = D - \sum_{i=1}^{D}(1 - p_i)^N,$$

where $(1 - p_i)^N$ is the probability that no record with group label i occurs among the N input records. Note that the function G is well defined also for non-integer arguments.

We will also need the *absorption rate* at point N, that is, the probability that record $N + 1$ will be absorbed into one of the groups already in memory. The probability that it will *not* be absorbed and, hence, will create a new group is $G(N+1)-G(N)$. The absorption rate at N is then

$$A(N) = 1 - (G(N + 1) - G(N)) = 1 - \sum_{i=1}^{D} p_i(1 - p_i)^N.$$

The functions $G(N)$ and $A(N)$ are plotted in Figures 1 and 2 for three different group size distributions: a uniform distribution, a Zipf distribution with $\alpha = 0.5$ and a traditional Zipf distribution ($\alpha = 1.0$).

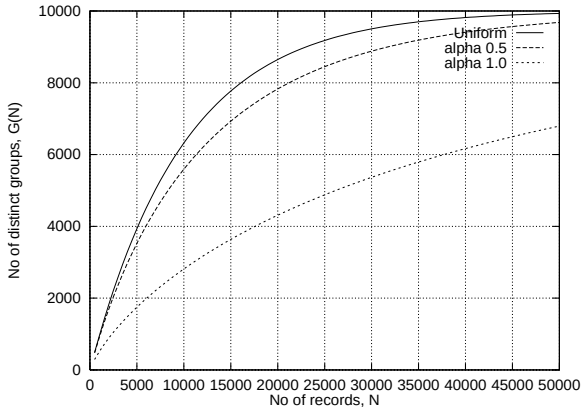

Figure 1: No of distinct groups seen as a function of records read, 10,000 groups.

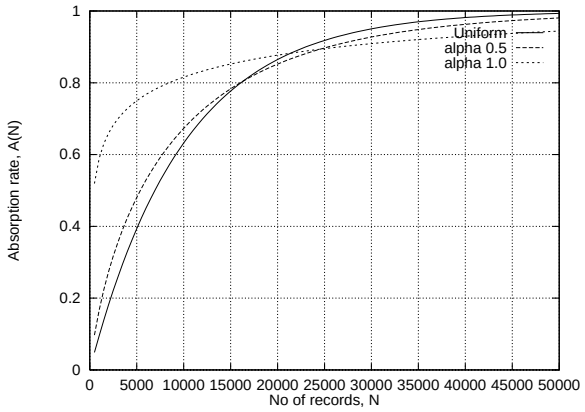

Figure 2: Absorption rate as a function of records read, 10,000 groups.

Now assume that we have memory space for storing at most M group records ($M < D$). We are interested in how many input records we expect to have processed when memory overflows. The expected number of records needed to reach this point is

$$R(M) = G^{-1}(M),$$

that is, the inverse of the function G. $R(M)$ can be computed by numerically solving the equation $M = G(X)$ for X.

We can plug $R(M)$ into the function A to obtain an estimate of the absorption rate obtained when storing M group records in memory. This function equals

$$A(R(M)) = 1 - \sum_{i=1}^{D} p_i (1 - p_i)^{R(M)}.$$

$A(R(M))$ is a measure of the "absorption power" of memory space for M group records. Note that this function applies only when the M group records stored in memory correspond to the first M distinct groups encountered in the input.

For the special case of a uniform distribution ($p_i = 1/D$), we can derive simple closed formulas for the three functions of interest.

$$\begin{aligned} G(N) &= D(1 - (1 - 1/D)^N) \\ R(M) &= \log_{(1-1/D)}(1 - M/D) = \ln(1 - M/D)/\ln(1 - 1/D) \\ A(R(M)) &= M/D \end{aligned}$$

4 Repeated scanning

Let's start with a very simple algorithm. Simply scan the input and apply early aggregation, that is, maintain group records in memory. When memory has been completely filled,

continue scanning the input, absorb all records that match a group already in memory and write all non-matching input records to an overflow file. The groups in memory are output when reaching the end of the input. The overflow file is then used as the input file for the next pass. This process is repeated until a pass produces no overflow records.

It is easy to see that each pass outputs as many groups as there is room for in memory. The number of passes is therefore equal to the total number of distinct groups in the input divided by the number of groups that fit in memory. However, what matters more is the amount of data written to and read from overflow files. This depends on the distribution of group sizes; large groups tend to be processed in the first pass.

Analysis

Assume that we have room for M group records in memory. The algorithm will then extract M complete groups in each pass over the input or over the overflow file produced in the previous pass. We first consider the number of overflow records produced by the initial pass. Based on the analysis in section 3, we expect the first $R(M)$ input records to produce no overflow records - this is the build-up phase. At this point, we have M records in memory with an absorption rate of $A(R(M))$ or, equivalently, a rejection rate of $1 - A(R(M))$. The absorption rate does not change after memory has been filled. Hence we can estimate the number of records output to the overflow file as

$$\begin{aligned} W_1(N) &= \begin{cases} (N - R(M))(1 - A(R(M))) & \text{if } R(M) < N \\ 0 & \text{if } R(M) \geq N \end{cases} \\ &= \begin{cases} (N - R(M)) \sum_{i=1}^{D} p_i (1 - p_i)^{R(M)} & \text{if } R(M) < N \\ 0 & \text{if } R(M) \geq N \end{cases} \end{aligned}$$

The following observation is key to estimating the number of records output after multiple passes. Provided that overflow files are read in the same order as they are written, then n *passes, each extracting M groups, will produce exactly the same groups as one pass extracting nM groups.* (This can be generalized: n passes extracting $M_1, M_2, \cdots, M_n$ groups, respectively, will produce the same groups as one pass extracting $M_1 + M_2 + \cdots + M_n$ groups.) It follows that

$$\begin{aligned} W_n(N) &= \begin{cases} (N - R(nM))(1 - A(R(nM))) & \text{if } R(nM) < N \\ 0 & \text{if } R(nM) \geq N \end{cases} \\ &= \begin{cases} (N - R(nM)) \sum_{i=1}^{D} p_i (1 - p_i)^{R(nM)} & \text{if } R(nM) < N \\ 0 & \text{if } R(nM) \geq N \end{cases} \end{aligned}$$

The total number of group records written to (and read from) overflow files is then

$$W(N) = \sum_{1 \leq i < D/M} W_n(N).$$

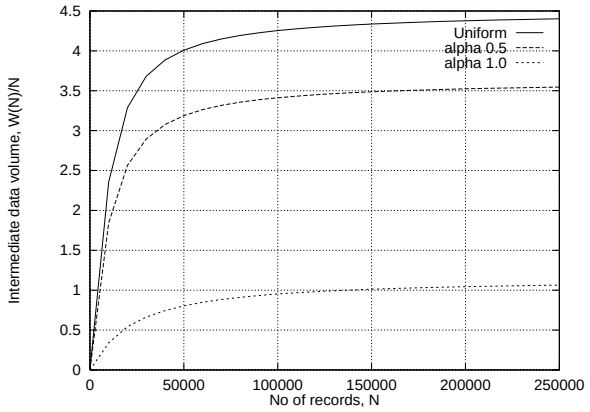

Figure 3: Intermediate data volume as a function of input size when using repeated scanning. 10,000 groups, 10% in memory.

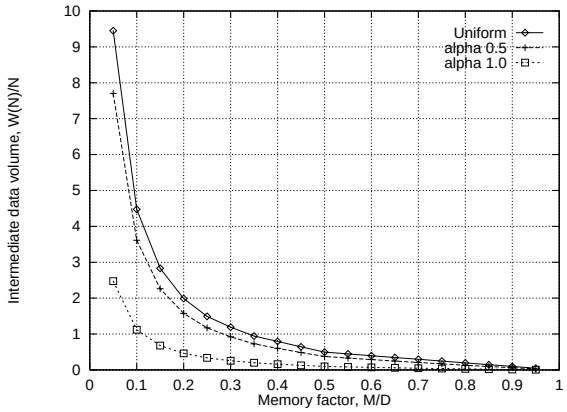

Figure 4: Intermediate data volume as a function of memory size when using repeated scanning. 10,000 groups, 1,000,000 input records.

Figures 3 and 4 illustrate how the volume of intermediate data varies with input size and amount of memory used. The intermediate data volume is expressed as a fraction of the number of input records. Figure 3 plots the intermediate data volume as a function of input size. The relative volume increases fairly sharply with the input size at first but converges to a steady state. The steady state value decreases as the skew in the group size distribution increases. Figure 4 shows the effects of increasing the amount of memory used during grouping. As one might expect, increasing memory has the best payoff when the amount of memory is low, i.e. the marginal benefit of additional memory decreases.

5 Repeated union with aggregation

Our second algorithm resembles a nested-loop join with the group records in memory as the outer table and a temporary file containing all group records created so far as the inner table. The first time memory overflows, we output the group records in memory to a temporary file on disk. We then fill memory again, applying early aggregation. Next we scan the temporary file and, for each record in the file, look for a matching record in memory. If one exists, we merge it with the record from the file, delete it from the in-memory table, and output the updated record (to a second temporary file). If no matching record exists, we output the record read without change. When reaching the end of the temporary file, we output all records remaining in the in-memory table. This process of filling memory and combining it with the result obtained so far continues until there are no more input records. The actual operation performed has more in common with union than with join which is why we call this algorithm *repeated union with aggregation*.

The explanation above indicated that two temporary files would be needed, one for input and one for output. This is not always necessary. Instead we can simply update the record in

the temporary file whenever we find a matching record in the hash table. Records remaining in the hash table are appended to the end of the temporary file. This idea can be applied only if updating a record is guaranteed not to change its length. The record length may change, for example, if the aggregation includes MAX on a variable length column.

It is also possible to do an "index union", that is, we store the groups found so far in an indexed file with the grouping column as the index key. Performing the "union" operation then consists of scanning the records in the in-memory table, locating matching records via the index, and updating the records found. Whenever no matching record is found, a new record is added to the file. It is questionable whether using an index is worthwhile. Even when the number of records in memory is quite small, we can expect to update almost every page in the indexed file. If so, a complete sequential scan is much faster.

Analysis

From the analysis in section 3 we know that the expected number of input records consumed is $R(M)$ when memory overflows. The first memory load will produce exactly M output records. The second memory load triggers a "union" of the M records in memory with the M records output previously. However, the "union" may produce less than $2M$ output records because matching input records will be merged, producing a single output records. The M group records read from the temporary file and the M group records in memory each consumed $R(M)$ input records. The output from the "union" cannot contain two records related to the same group. It follows that *the output after the "union" will consist of as many records as we can expect to find distinct groups among* $2R(M)$ *input records*, which is given by the function G. The expected number of records output from the first "union" is therefore equal to $G(2R(M))$ if $2R(M) < N$ and $G(N)$ otherwise.

It is now easy to see how this generalizes to multiple "union" passes. The output from the nth "union" is a consolidation of $nR(M)$ input records if $nR(M) < N$. Otherwise it is a consolidation of the N input records. Putting it all together we obtain the following formula for estimating the number of records in the intermediate file output after n memory loadings:

$$W_n(M) = \begin{cases} M & \text{if } n = 1 \text{ and } R(M) < N \\ G(nR(M)) & \text{if } n > 1 \text{ and } nR(M) < N \\ G(N) & \text{otherwise} \end{cases}$$

The total number of records of intermediate data produced by the algorithm is then

$$W(M) = \sum_{n=1}^{n<N/R(M)-1} W_n(M)$$

This function is plotted in Figures 5 and 6. A comparison with the corresponding figures in the previous section immediately shows that this method outputs much more intermediate data than even the simple repeated scanning algorithm.

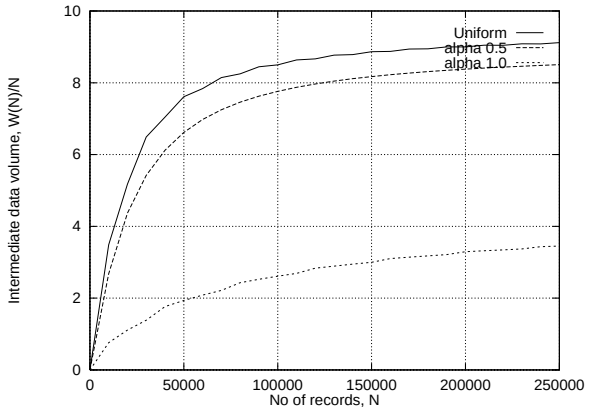

Figure 5: Intermediate data volume as a function of input size when using repeated unioun with aggregation. 10,000 groups, 10% in memory.

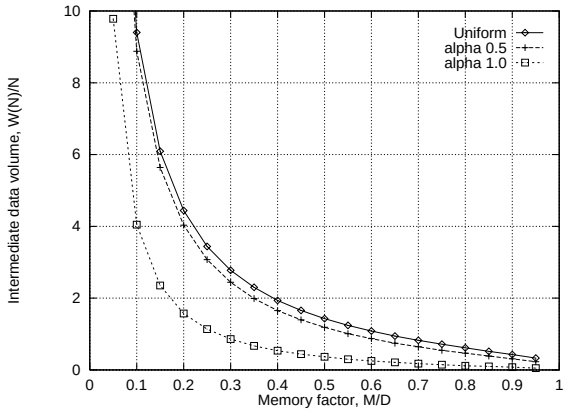

Figure 6: Intermediate data volume as a function of memory size when using repeated union with aggregation. 10,000 groups, 1,000,000 input records.

6 Grouping by sorting

GROUP BY queries are often evaluated by first sorting the input on the grouping columns and then performing aggregation on the sorted record stream. The most widely used sort method (for large files) is merge sort which consists of two phases: run formation and run merging. Each phase is discussed in a separate section.

6.1 Run formation

There are two types of run formation algorithms: those producing fixed-length runs and those producing variable-length runs. Fixed-length run formation algorithms read a certain amount of input into memory (limited by the size of available memory), sort the data in memory using some sorting algorithm, and then write out the result as a run. This process continues until all input records have been processed. We will use the more descriptive name *load-and-sort* run formation for this class of algorithms.

Variable-length run formation algorithms may produce runs that are larger than the available memory. The basic idea is straightforward: when memory becomes full, output just a small part of a run (minimum one record) and read in more records. New records with a key greater than or equal to the key of the last record output, are added to the current run. Those whose key is too low become part of the next run. The standard algorithm is replacement selection, see reference [4] for details of the algorithm. E. F. Moore showed that, for randomly ordered input, the expected length of each run is twice the available memory size [4]. When the input exhibits some degree of pre-sortedness, runs are likely to be even longer.

Replacement selection produces runs that are, on average, twice as large as memory

used during run formation. Even so, load-and-sort algorithms are often used in practice. Replacement selection has two drawbacks: excessive data movement and complex memory management when input records are of variable length. When a record is output (actually copied to an output buffer), it leaves a hole somewhere in memory. If records are of fixed length, any incoming record can be moved into the hole. But that is precisely the problem. It takes time to copy every record from an input buffer to some place in memory. Load-and-sort algorithms do not incur this extra copying step: we use the available memory as input buffers, do pointer sorting, and copy records directly into output buffers. Variable length records compound the problem because there is no guarantee that an incoming record will fit into the hole. Even if it does, it may not be an exact fit, leaving a smaller hole. After a while, we end up with many small, virtually unusable pieces of free space. Unless we occasionally do memory packing to reclaim unused space, the net effect is to reduce the number of records that fit in memory which reduces the benefits of replacement selection.

Early aggregation can easily be incorporated into either type of run formation algorithm. The runs will be of fixed or variable length depending on what action is taken when memory becomes full. We can sort the group records and output them as a run, in which case all runs will be of the same length. The other option is to apply replacement selection to the group records, that is, output one (or more records) and use the space for a new group record. In this case, the runs will be of variable length but with an expected length of $2M$.

Regardless of what run formation algorithm is used, input records now require some processing (lookup, aggregate) and cannot just be left in their input buffers. So when applying early aggregation, they all require the same amount of copying. However, the memory management problem for replacement selection still remains.

Analysis

Assume that runs are created by a load-and-sort algorithm, that is, we read input records applying early aggregation until memory is filled, then sort the group records in memory and output them as a run. A run will then consist of M (group) records and we can expect to have consumed $R(M)$ input records. Hence, we can expect the run formation phase to produce $N/R(M)$ runs, containing a total of $NM/R(M)$ group records. Figure 7 plots the function $M/R(M)$ for three different group size distributions. As expected, the more skewed the distribution is, the more early aggregation pays off.

Now assume that we are using replacement selection for run formation. Replacement selection is similar to the repeated scanning algorithm (section 4) in the sense that there are always M group records in memory. The difference is that the scanning algorithm keeps the same records in memory throughout the whole pass while replacement selection replaces one group record with another whenever memory overflows. We know that the absorption rate is $A(R(M))$ when memory overflows the first time. We approximate the absorption rate for replacement selection by this rate and estimate the number of records output from the run formation phase as $NA(R(M))$. Replacement selection produces runs containing $2M$ group

records, on average, so the number of runs can be estimated as $NA(R(M))/(2M)$.

It is clear that $A(R(M))$ overestimates the absorption rate of replacement selection because the same records are not kept in memory during the whole pass. We have not been able to come up with a more exact analysis but we ran several simulation experiments to get some idea of the discrepancy. The estimated and observed absorption rates are plotted in Figure 8. For a uniform distribution, $A(R(M))$ is the correct absorption rate and the estimated and observed rates coincide. The discrepancy increases with the skew but even for the highly skewed traditional Zipf distribution ($\alpha = 1.0$) the difference is small.

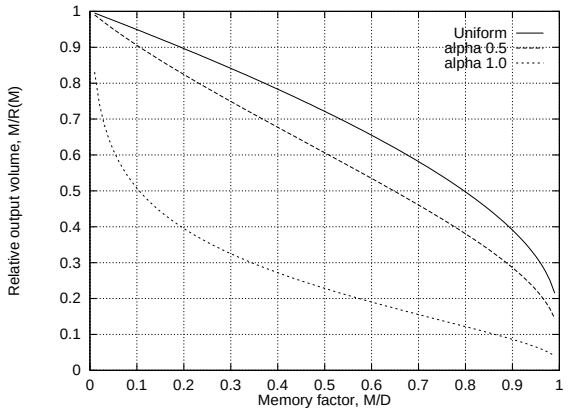

Figure 7: Relative output volume as a function of memory size when creating runs by load and sort, 10,000 groups.

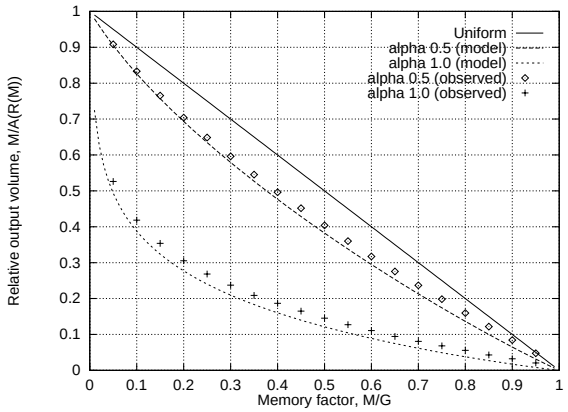

Figure 8: Relative output volume as a function of memory size when creating runs by replacement selection, 10,000 groups.

6.2 Run merging

There are many valid merge patterns because the only requirement is that each merge step must reduce the number of runs so that we eventually end up with a single, completely sorted run. It is not even necessary to have separate run formation and merge phases. We can structure a sort as two concurrent activities connected in a producer-consumer relationship. We can have multiple run-producing tasks running concurrently with multiple merge tasks consuming (and producing) runs. A merge task monitors a list of available runs and starts merging as soon as there are enough runs and memory space available to make a merge worthwhile.

We will, however, consider a traditional approach with a separate merge phase where each merge step has the same fan-in, except possibly the first one. So given S initial runs, possibly of variable length, and a maximum merge fan-in of K, which merge pattern results in the minimum data transmission? The surprisingly simple solution can be found in reference [4], pp 365-366: first add enough dummy runs of length zero to make the number of runs divisible by $K - 1$ and then repeatedly merge together the K shortest existing runs until only one run remains.

Analysis

We first analyze the output from one merge step and then show how to compute the intermediate data volume produced by the optimal merge pattern just described.

Suppose we are merging K initial runs and that we apply early aggregation during the merge. An initial run is a run created during run formation. Let L denote the run length. If runs were formed by load-and-sort, the run length equals M and if runs were formed by replacement selection, the (expected) run length is $2M$. We will read KL group records but the number of records output will be less because all records related to the same group but in different runs will be consolidated into a single output record. How many records can we expect to output? The expected number of input records needed to create an initial run equals

$$C(M) = \begin{cases} R(M) & \text{if using load and sort} \\ A(R(M)) & \text{if using replacement selection} \end{cases}$$

so creation of the K runs consumed a total of $KC(M)$ input records. The run output from the merge step will not contain any duplicates. Consequently, *the number of records in the output run is the same as the number of distinct groups occurring among the input records consumed to create the K initial runs.* The number of distinct groups occurring among x input records is given by the function $G(x)$ so we can estimate the number of output records as

$$G(KC(M)) = D - \sum_{i=1}^{D}(1 - p_i)^{KC(M)}.$$

Now consider what happens if we merge K initial runs into a single run in multiple steps. The key observation is this: *the merge pattern does not affect the size of the output run.* Assume, for example, that we merge 8 initial runs into a single run. The result will be exactly the same if this is done as one 8-way merge, as two 4-way merges followed by a 2-way merge, or as four 2-way merges followed by two 2-way merges and a final 2-way merge. It follows that the formula above can be used to estimate the output size from any merge step by interpreting K as the number of *initial runs* contained in the input to the merge step.

We now know how to estimate the size of the output from any merge step. The easiest way to compute the total intermediate data volume produced during a merge is by, in essence, simulating the merge process. A sketch of the algorithm follows.

Algorithm: Intermediate Data Volume Produced by Optimal Merge

```
Input:  RunList, a list giving the length of each initial run. The length
        is expressed as the number of input records consolidated in the run.
        RunCount, number of runs in RunList.
        FanIn, merge fan in.
Output: DataVolume, the expected volume of intermediate data processed during
        the merge. Expressed in number of group records. Includes the group
        records in the initial runs but not the final output from the merge.
```

```
Locals: PQ, a priority queue with elements of type (RecsOut, SourceRecs) where
        RecsOut is the number of group records in a run and SourceRecs is the
        number of original input records consolidated in the run. The element
        with the lowest RecsOut has the highest priority.

PQ = empty ;
for each element r in RunList do
  insert (G(r.length), r.length) into PQ ;
od

DummyRuns = (FanIn-1) - (RunCount-1) mod (FanIn-1) > 0 ;
if DummyRuns < FanIn-1 then
  insert (0, 0 ) into PQ DummyRuns times ;
fi

DataVolume = 0 ;
while PQ contains more than one element do
  SrcLength = 0 ;
  for i from 1 to FanIn do
    (LOut, LSrc) = extract top element from PQ ;
    SrcLength = SrcLength + LSrc ;
    DataVolume = DataVolume + LOut ;
  od
  insert (G(SrcLength), SrcLength) into PQ ;
od

output DataVolume ;
```

Numerical results are plotted in Figures 9 - 12. The first two figures show the intermediate data volume as a function of the input size and the last two as a function of memory size. Applying early aggregation reduces the data volume very significantly, especially if the group size distribution is skewed. Consider, for example, the situation when processing 200,000 input records and forming runs by replacement selection (see Figure 10). Without early aggregation, the intermediate data produced equals 2.00*200,000 = 400,000 records (regardless of the skew in the input). With early aggregation using memory for 1000 group records, the intermediate data volume is reduced to 1.30*200,000 = 260,000 records for a uniform group size distribution and to as little as 0.43*200,000 = 86,000 records for a Zipf distribution. Note that extra memory is required only during run formation.

7 Grouping by partitioning

GROUP BY queries can also be evaluated by partitioning. The basic idea is to first partition the input into smaller work files by hashing on the grouping columns and then apply

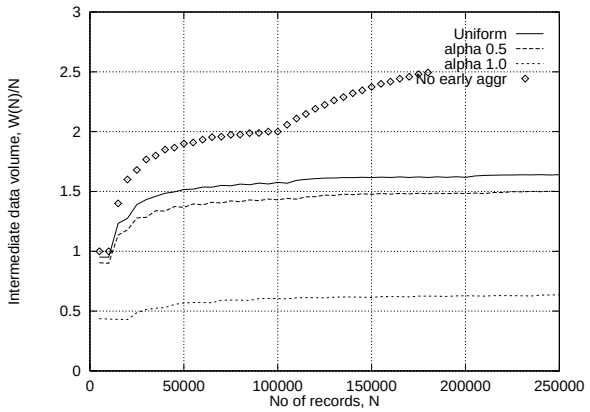

Figure 9: Intermediate data volume as a function of input size for grouping by sorting with run formation by load and sort and using an optimal merge pattern. 10,000 groups, 10% in memory, fan in 10.

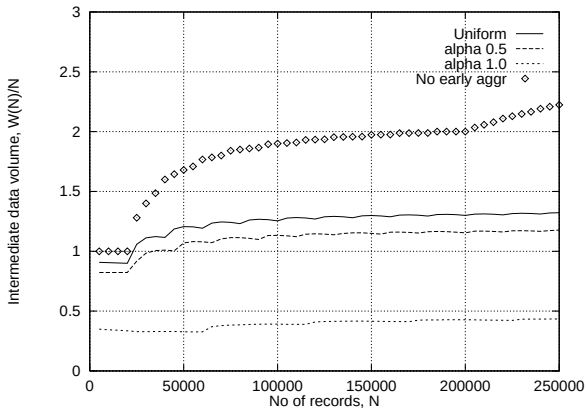

Figure 10: Intermediate data volume as a function of input size for grouping by sorting with run formation by replacement selection and using an optimal merge pattern. 10,000 groups, 10% in memory, fan in 10.

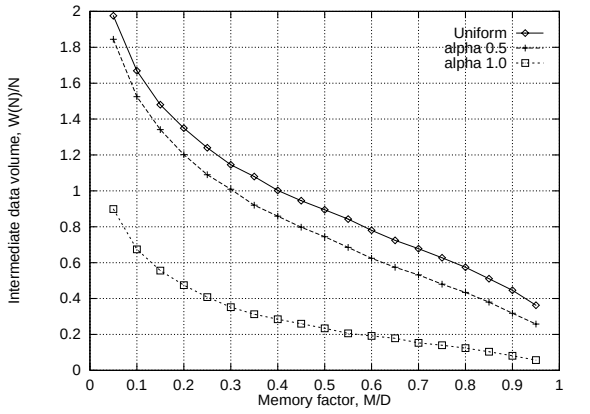

Figure 11: Intermediate data volume as a function of memory size when using sorting with run formation by load and sort. 10,000 groups, 1,000,000 input records, fan in 10.

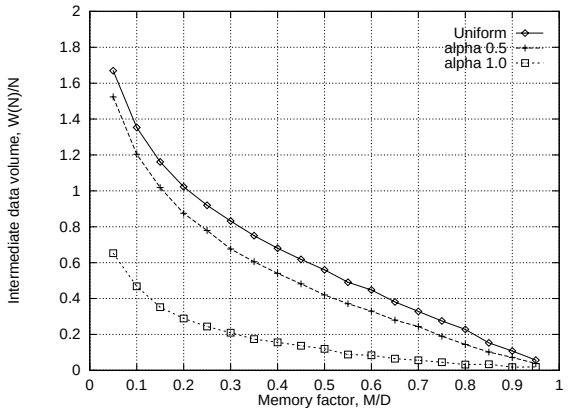

Figure 12: Intermediate data volume as a function of memory size when using sorting with run formation by replacement selection. 10,000 groups, 1,000,000 input records, fan in 10.

simple hash grouping on each partition file. The objective is to create partitions containing sufficiently few distinct groups so that the final grouping can be done in memory. Early aggregation is applied during the partitioning phase to reduce the amount of data written to partition files. We consider two algorithms based on partitioning: one partitioning rejected input records and one partitioning partially aggregated groups.

7.1 Partitioning rejects

The repeated scanning algorithm explained in section 4 writes all rejected records into a single file. We can improve on this by dividing them among P files by hash partitioning. This requires P output buffers, one for each partition file, where rejected records are placed. An output buffer is written as soon as it becomes full. In addition, all non-empty output buffers are written when we reach the end of the input stream.

We need some policy for assigning rejected records to partitions. The simplest solution is a fixed, uniform assignment. We make the hash table size mP where m is an integer and any rejected record hashing to $0, 1, \cdots, m-1$ is written to partition 1, any rejected record hashing to $m, m+1, \cdots, 2m-1$ is written to partition 2, and so on.

When we reach the end of the input stream, the group records in memory are complete and can be output. Exactly the same algorithm is then applied to each one of the partition files, in any order. This may produce additional, second level partition files which may create third level partition files, and so on. This continues until there are no more partition files to process. Note that we must use a different hashing function at each level. Otherwise, all records rejected when processing a partition file will end up in a single partition file at the next level. The algorithm will still produce the correct result but not as efficiently.

We can save on disk storage for partition files by processing them in order of size, from smallest to largest. A smaller partition is expected to contain fewer distinct groups. Hence, we can expect it to generate fewer, if any, overflow records. We must somehow keep track of partitions remaining to be processed. If we organize this as a priority queue with the priority determined by partition size, the smallest available partition will always be processed first.

Allocating a few extra output buffers makes it possible to overlap writing to partition files and processing of input records.

If we are doing duplicate elimination, all we need to store in memory are record keys, not complete records. Whenever we encounter a new group and there is still room in memory, we output the complete record to the result, and store just its key in the hash table. (Rejected records are still output as is to partition files.) This trick may allow many more records to be stored in memory.

Analysis

So far we have ignored the space needed by file buffers because the number of buffers needed has been small. Methods based on partitioning may consume a significant amount of memory

as file buffers for the P partition files used in each step. We measure the size of a buffer in (group) records and denote the size of a file buffer by B.

Partitioning rejects resembles repeated scanning in the sense that they both output rejected records to overflow files. The analysis exploits this similarity. The only, but important, difference is the hash partitioning of the rejected records into P files.

The initial pass of the algorithm will produce exactly the same number of records as the first pass of the repeated scanning algorithm with memory size $M - PB$. This equals

$$\begin{aligned} W_1(N) &= \begin{cases} (N - R(M_1))(1 - A(R(M_1))) & \text{if } R(M_1) < N \\ 0 & \text{if } R(M_1) \geq N \end{cases} \\ &= \begin{cases} (N - R(M_1)) \sum_{i=1}^{D} p_i (1 - p_i)^{R(M_1)} & \text{if } R(M_1) < N \\ 0 & \text{if } R(M_1) \geq N \end{cases} \end{aligned}$$

where $M_1 = M - PB$ and B is the output buffer size (in records).

The next pass processes each of the (at most) P partition files applying the same partitioning algorithm. We model this as P passes of the simple repeated scanning algorithm. (Clearly, this is not entirely correct - it will underestimate the number of records rejected during the second pass. More about this in a moment.) According to the analysis of repeated scanning, the effect is the same as a single pass using memory of size $(M - PB)(1 + P)$. The one in the formula accounts for the initial pass. Continuing in this fashion, the effect of a complete third partitioning pass is the same as a single pass using memory of size $(M - PB)(1 + P + PP)$. This gives us the following formula for estimating the number of records output from the nth partitioning pass:

$$\begin{aligned} W_n(N) &= \begin{cases} (N - R(M_n))(1 - A(R(M_n))) & \text{if } R(M_n) < N \\ 0 & \text{if } R(M_n) \geq N \end{cases} \\ &= \begin{cases} (N - R(M_n)) \sum_{i=1}^{D} p_i (1 - p_i)^{R(M_n)} & \text{if } R(M_n) < N \\ 0 & \text{if } R(M_n) \geq N \end{cases} \end{aligned}$$

where $M_n = (M - PB)(1 + P + P^2 + \cdots + P^{n-1})$. The total number of records output to partition files can then be estimated as

$$W(N) = \sum_{n=1}^{W_n=0} W_n(N).$$

Figures 13 and 14 plot the number of records output to partition files relative to the number of input records. Compared with other grouping algorithms the amount of intermediate data is small except when memory is very tight. The knee in the graph for uniformly sized groups is where the shift from one partitioning pass to two partitioning passes occurs. Similar knees occur when shifting from two to three passes, etc.

We already mentioned that the model underestimates the amount of intermediate data produced by the algorithm. Simulation experiments showed that the error is very small as

long as the number of partitioning levels is no more than two. The model assumes an abrupt switch-over, that is, all partition files on the same level overflow at the same time. That would indeed be true if all partition files received exactly the same number of distinct groups. In reality, the switch-over is more gradual because hash partitioning does not assign groups completely evenly among the partitions.

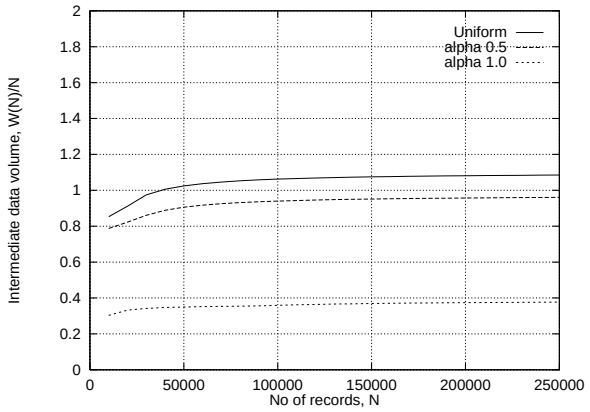

Figure 13: Intermediate data volume as a function of input size when using simple partitioning. 10,000 groups, 10% in memory, 10 partitions, buffer size 25 records.

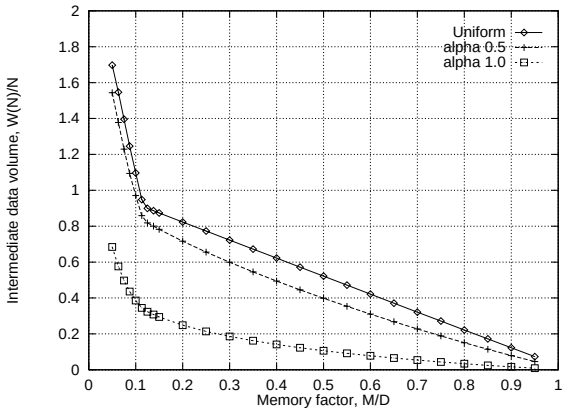

Figure 14: Intermediate data volume as a function of memory size when using simple partitioning. 10,000 groups, 1,000,000 input records, 10 partitions, buffer size 25 records.

7.2 Partitioning partially aggregated records

The previous algorithm outputs rejected input records to P partition files. Instead we can output partially aggregated records in the same way as the sort based algorithms. This idea can be implemented in a variety of ways. We experimented with several different algorithms and the algorithm described in this section emerged as the winner, i.e. it produced the least intermediate data.

We first create a hash table with P entries, called the partition table. The available memory is divided into fixed-size pages; initially all pages are empty and placed on a list of free pages. An entry in the partition table will point to a chain of pages containing group records that hashed to that entry. Each entry is also assigned a separate partition file. Pages overflowing from a partition are written to the associated partition file.

We then start reading and processing input records. As a record arrives, we first attempt to locate a matching group record. If a matching group record is found, its aggregation columns are updated and we are done. Otherwise, we have to create a new group record. If there is enough room on the last (or any) page in the appropriate chain, we simply append the new group record, update the aggregation fields and we are done. It there is insufficient space on the last page of the chain, we first have to get another page from the free list and add it to the end of the chain.

This initial phase continues until there are no more free pages or no more input records. If the input runs out before we run out of memory space, we simply output all the group records in memory and the process is finished.

The more interesting case occurs when we run out of memory, i.e. we need another page and there are no more free pages. To create free space we must empty a page by writing it to its partition file. The question is: which page do we select as a victim? We will get back to this question but, for now, assume that we have settled on some policy for selecting which page to output. We write out the selected page, disconnect it from its current chain, clear it, and append it to the end of the chain that needed more memory.

We also keep track of whether a partition has overflowed or not and whether a page is clean or dirty. A page is clean if all its (group) records were created before the partition overflowed, otherwise it is considered dirty. This distinction becomes important when reaching the end of the input stream. We know for certain that a group record on a clean page cannot merge with any other group records so clean pages can be output directly without further processing. Dirty pages contain group records that may merge with other group records and hence dirty pages have to be written to the partition file. A more sophisticated alternative is to distinguish between clean and dirty group records. A group record is marked dirty if it was created after its partition overflowed. However, this level of detail may not be worth it.

Now back to the replacement policy, that is, which page(s) to output when we need a free page and memory is full. The goal is to minimize the intermediate data volume, which implies that (a) we should output only full or almost full pages and (b) select pages with a low absorption rate.

The following variant of LRU (least recently used) replacement was found to perform well. We maintain several LRU queues and pages are assigned to the queues based on fill factor and whether they are clean or dirty. Assume for the moment that records are of fixed length and that a page has room for 10 records. We then create a total of 20 LRU queues (plus a list of free pages): one for dirty pages storing 10 records, one for clean pages storing 10 records, one for dirty pages storing 9 records, one for clean pages storing 9 records, and so on. When we need to free up a page, we search for a victim in the same order: the (10, dirty) queue, the (10, clean) queue, the (9, dirty) queue, the (9, clean) queue, and so on. The first page found is written to its partition file, cleaned and freed for reuse. Other search orders, representing different trade-offs, are of course possible.

This approach can be easily adapted to variable length records: define some number of fill factor ranges and have two LRU queues for each fill factor range. For example, the fill factor ranges could be: over 90% filled, 80% to 90% filled, 70% to 80% filled, down to less than 10% filled. It is not necessary that all ranges be of the same size; any division into ranges will do. However, uniform ranges makes it much easier to determine to which queue a page belongs.

To speed up search, one might be inclined to use a large partition table and assign multiple entries to the same partition file. This turned out to be a bad idea: it results in poor memory utilization and a low absorption rate, thereby increasing the volume of

intermediate data. Suppose that we have memory space for 100 pages and that we create a partition table with 100 entries. On average, each entry will then be assigned one page but there will be considerable variation: some entries will have several pages assigned and many will have none. At some point while processing input records, we will run out of free pages so we have to output a page. This is likely to happen well before all pages are full. In fact, it may happen before *any* page is full. Suppose pages are, on average, half filled when we need to output a page. In effect, we are then making use of only half the memory for early aggregation. Furthermore, we may have to write out pages that are not completely filled.

Now suppose we use a much smaller table of, say, 20 entries. On average, each entry will be assigned five pages, again with some variation. However, we are now guaranteed that at least 80 pages will be completely filled because only the last page on each of the 20 chains can remain partially filled. This improves the effective memory utilization, which increases the absorption rate. In addition, we never have to output a partially filled page (with one exception: flushing out pages when reaching end of the input stream).

So the partition table size should be determined by the number of pages available. In the experiments reported in the next section, the table size was set to the number of pages divided by five, with an upper bound of 50. This achieves an intermediate data volume close to minimum. To speed up searching, it is better to maintain a separate, much larger, hash table which is used strictly for lookup.

To overlap writing to partition files and processing, we hold back a few pages on the free list. When we need to write a page from a partition, we immediately allocate a page from the free list so that processing can continue while writing.

The discussion above assumes that we output one page at a time. If pages are small, we can reduce disk overhead (total seek time and latency) by writing multiple pages in a single write operation. This works particularly well if the system supports gather-write and scatter-read. Note, however, that all pages written together must belong to the same partition which makes selecting the best set of pages more complex.

Analysis

This algorithm is quite complex and we have not been able to come up with a mathematical model. Instead we have to rely on simulation experiments. The results of one series of experiments are plotted in figures 15 and 16. The page size is 25 records and the number of partitions was chosen as the number of pages divided by 5, with a maximum of 50 partitions. The graphs are very similar to the corresponding graphs for the algorithm partitioning rejects.

8 Comparison of Algorithms

Figures 17 and 18 compare the volume of intermediate data produced by the six algorithms considered in this paper. The line labeled "Lower bound" may need some explanation.

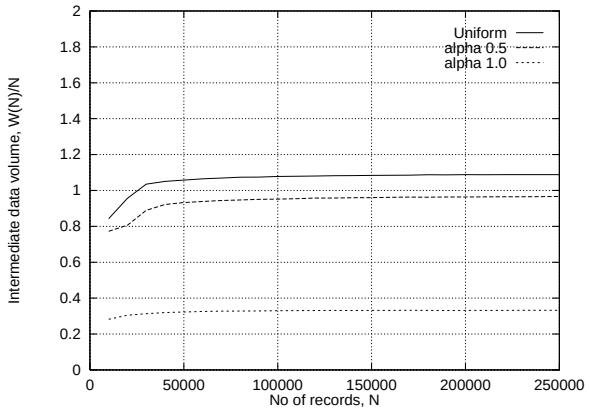

Figure 15: Intermediate data volume as a function of input size when partitioning partially aggregated records. 10,000 groups, 10% in memory, page size 25 records, min 5 pages/partition.

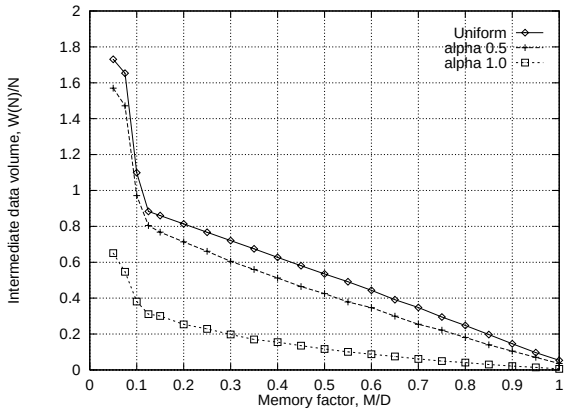

Figure 16: Intermediate data volume as a function of memory size when partitioning partially aggregated groups. 10,000 groups, 100,000 input records, page size 25 records, min 5 pages/partition.

It shows the volume of intermediate data produced by partitioning rejects but under the unrealistic assumptions that output buffers take zero space and one partitioning step is always sufficient. We claim that no method can produce less intermediate data provided that (a) the input data is truly random, without clustering and (b) no look-ahead or preprocessing of the input is allowed. Any method producing less intermediate data must achieve a higher absorption rate. All of memory is used for early aggregation so we cannot increase the absorption rate by storing more records in memory. So the only possibility would be replace some of the records now in memory with records having a higher absorption rate, i.e. from a group with a higher occurrence propability. If the group size distribution is uniform, switching which records are kept in memory will not help because every group has the same probability of occurring in the rest of the input. If the group size distribution is not uniform, the reasoning is slightly different. Consider the first time memory overflows. Is the new record expected to have a higher absorption rate than any of the records already in memory, i.e. does it belong to a larger group? If so, we can increase the absorption rate by replacing that group. The answer is no. The larger the group, the sooner we can expect to encounter its first record in the input. The group records in memory represent those groups that were encountered the earliest. So any record that does not match one of the groups in memory is expected to be from a smaller group and therefore have a lower absorption rate.

Three of the algorithms considered, namely repeated union, repeated scanning and sorting with run formation by load-and-sort, produce much more intermediate data than the other three both when groups are of uniform size and when the group size distribution is highly skewed. They show slightly different behaviour though. Repeated union is worst regardless of the amount of memory available for early aggregation. Repeated union and sorting with run formation by load-and-sort perform surprisingly poorly even when as much as 80% or

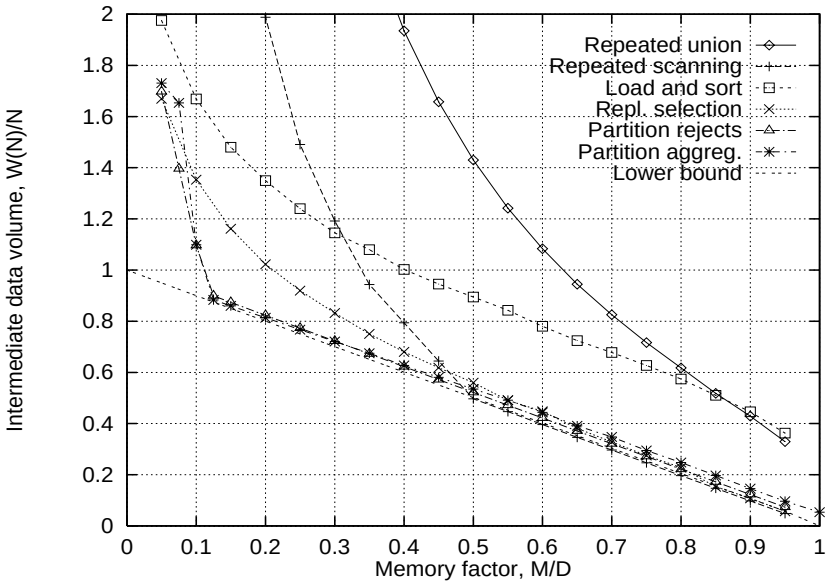

Figure 17: Comparison of intermediate data volume produced by different grouping algorithms, 10,000 groups, 1,000,000 records, 10% in memory,uniform distribution.

90% of the groups fit in memory. The explanation is the same for both algorithms. Suppose we have memory space for n records in memory. The problem is that the whole memory contents is output as soon as we encounter the $(n+1)$st distinct record in the input stream. All the other methods take less drastic action, either outputting one record or one page, thereby achieving much higher absorption rates. Repeated union, repeated scanning, and sorting with run formation by load-and-sort are not discussed further in this paper.

The two versions of partitioning show virtually identical performance, which is also close to the lower bound when more than 10% of the groups fit in memory. Sorting with run formation by replacement selection performs almost as well. The difference is significant only for groups of uniform size when a moderate amount of memory is available (10-40%).

9 Implementation Considerations

Record size

A word of caution regarding algorithms that output partially aggregated records may be appropriate first. Early aggregation never increases the number of output records, but the output records (group records) may be larger than the records that would be output without early aggregation. This may happen when several aggregation functions are computed on the same column; for example, when computing MIN(PRICE), AVG(PRICE), and MAX(PRICE). The column PRICE in an input record will then be expanded to four

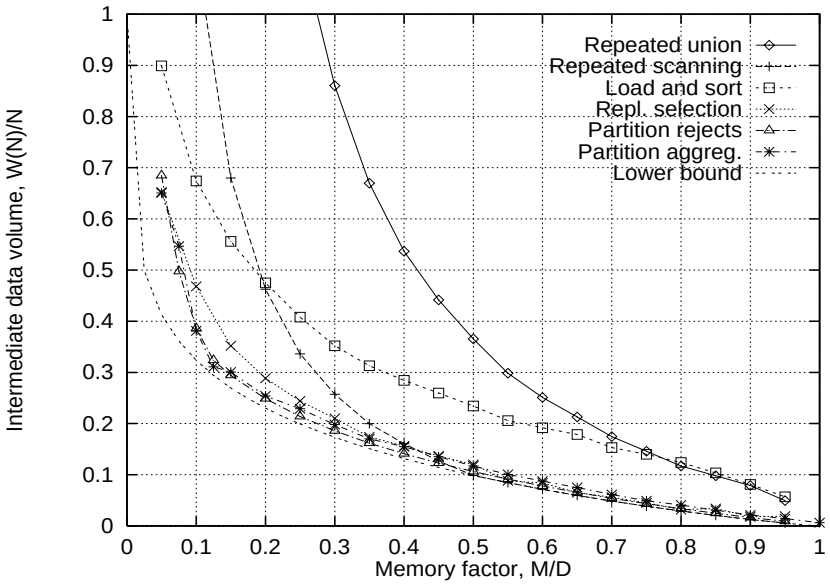

Figure 18: Comparison of intermediate data volume produced by different grouping algorithms, 10,000 groups, 1,000,000 records, 10% in memory, Zipf distribution ($\alpha = 1$).

columns in group records. (Four because AVG has to be expanded into SUM and COUNT If output (group) records are larger than input records and the number of records is reduced only slightly, the total amount of data output may increase.

Code complexity

Partitioning rejects is clearly the easiest algorithm to implement. The only difficult decisions are (a) how many partition files to use and (b) how large to make their output buffers. This will determine how much memory remains for early aggregation. It is not clear what the optimal allocation is even if we were able to get reliable estimates of input size and number of distinct groups. In the absence of reliable estimates, a reasonable approach is to allocate a modest fraction of memory for output buffers, say no more than 10-20% and to use 5-20 partition files.

Implementing sorting with run formation by replacement selection is fairly difficult. The tricky part is efficient memory management for variable length records. However, adding early aggregation to an existing implementation does not appear overly difficult. To be able to locate matching group records quickly, we add a hash table used strictly for fast lookup. Maintaining this hash table requires some care if records may move while residing in memory.

The algorithm for partitioning of partially aggregated records requires a fair amount of code but the code is reasonably straightforward. In addition to the partition table we need to maintain three auxiliary structures: a list of free pages, a set of LRU queues and a hash table for fast lookup. Searching for a matching record via the partitioning table is too slow;

there are too many records behind each entry.

Data movement

Given the relatively slow speed of main memory (compared to cache memory), it is important to minimize the amount of in-memory data movement. The two partitioning based algorithms do not incur any extra data movement: data is moved immediately from input buffers to output pages that can be written directly. Output records need never be moved. Sorting with replacement selection, on the other hand, always requires an extra copying step. Every output record has to be copied from somewhere in memory into an output buffer. Some output records may also require additional copying during memory compaction.

Large units of I/O

The total I/O time depends not only on the amount of data transferred but also on the number of I/O operations. Each disk operation (read or write) incurs significant overhead in the form of seek time and rotational delay. The overhead can be reduced by using large units of I/O, that is, having each read and write operation transfer multiple pages. We might, for example, have each I/O operation transfer 64KB (8 times 8KB pages or 16 times 4KB pages). (For the sake of brevity, we will not discuss how to overlap I/O and processing.)

Sorting with replacement selection can easily be adapted to use large units of I/O both for writing and reading of intermediate data. For run formation, all that is required is to allocate a large output buffer. During the merge phase, we need one large buffer for each input file and one for the output file.

The algorithm partitioning rejects can also easily be adapted to use large units of I/O simply by making the output buffers for each partition file large. However, this may consume a large fraction of available memory space, especially if the number of partitions is high. This reduces the memory available for early aggregation, thereby increasing the intermediate data volume. It is not clear what the best tradeoff is, i.e. when it is better to use memory for output buffers than for early aggregation.

The algorithm partitioning rejects does not use separate output buffers; all data is packed onto pages which can be written directly. If the system supports gather-write, a single write operation can transfer several pages (from the same partition) to disk. The only change required is to the policy for selecting which page to output: instead of selecting a single page, it now needs to select a set of pages from one of the partitions. For input from a partition file, all that is needed is a single large input buffer.

Hash table organization

All three algorithms make use of a hash table to quickly locate a matching group record. This will be a frequent operation so careful attention to the organization of this table is warranted. To reduce the cost of key comparisons, it may be worthwhile storing in the

hash table itself *key signatures*, that is, the hash value of the key (before taking it modulo the table size to obtain the entry number). To check whether two keys are equal, we first compare their key signatures and only if they are equal do we proceed with a comparison of the keys. Not only is it faster to compare signatures, it also improves cache behaviour.

Dynamic memory adjustment

Dynamic memory adjustment is the ability to reduce or increase the amount of memory used by an algorithm during run time. Few database systems implement dynamic memory adjustment today but this is likely to increase in the future. Dynamic memory adjustment makes it much easier to manage memory space and improves system throughput.

We will first consider sorting with run formation by replacement selection. Adjusting memory usage during run formation is easy. To decrease memory usage, we output some extra records and compact the records in memory to free up a contiguous piece of memory. To make use of additonal memory we just just stop outputting records until the additional memory is filled. The merge stage normally does not use large amounts of memory so adjusting memory usage during this stage is less important.

Dynamic memory adjustment is also easy when partitioning aggregated records. To free up memory, we just output one or more pages and return the pages. It may be necessary to move some pages around if contiguous pieces of memory have to be returned. If additional memory becomes available, all we need to do is add it to the list of free pages.

The algorithm partitioning rejects does not lend itself to dynamic memory adjustment. Memory cannot be freed up during a partitioning step (except early in the process when little aggregation has occurred). A group record in memory can be output to a partition file only if it contains exactly one on input record and it cannot be output to the final result until we reach the end of input. As soon as one record has been output to a partition file, we can no longer create new group records in memory if the group, if rejected, would belong to that partition. This means that we cannot always make use of additional memory.

10 Conclusion

Early aggregation reduces the amount of intermediate data that has to be stored on disk when evaluating a GROUP BY query. We described how to incorporate early aggregation into six algorithms for grouping and aggregation.

For five of the algorithms we were also able to accurately model the data reduction obtained by early aggregation. The reduction obtained by a given algorithm depends on

1. the memory factor, that is, what fraction of the distinct groups fit in memory (more is better),

2. the group size distribution (the more skewed, the better), and

3. the duplication factor, that is, the number of input records relative to the number of distinct groups (higher is better),

Three of the algorithms, namely sorting with replacement selection and the two algorithms based on hash partitioning, achieve much higher data reduction than the other three. The two partitioning based algorithms yield virtually the same data reduction which is somewhat better than that of sorting with replacement selection.

The data reduction can be very significant even with a modest amount of memory. For example, if only 10% of the groups fit in memory, early aggregation can reduce the intermediate data volume by 80% if the group size distribution is sufficiently skewed. If the number of groups is small, all group records may fit in memory completely and no intermediate data is produced.

The difference in intermediate data volume is not sufficient to declare a clear winner among the three algorithms; each algorithm has advantages and drawbacks. Partitioning rejects is the easiest to implement but memory cannot be adjusted at run time and a significant part of memory may be taken up by output buffers (thereby increasing the intermediate data volume). Sorting with replacement selection generates more intermediate data and requires extra in-memory copying but produces sorted output which may reduce the cost of subsequent operations. Partitioning partially aggregated records requires more code than partitioning rejects but memory adjustment at run time is easy and all the available memory space is used for early aggregation.

Previous analyses (of early duplicate elimination) assumed that groups are all of the same size. The model in this paper extends this to arbitrary group size distributions. A remaining open problem is how to take into account clustering of data, that is, when records from groups are clustered together more closely than what is expected under a random model of the data source.

References

[1] M. Abdelguerfi and A. K. Sood. Computational complexity of sorting and joining relations with duplicates. *IEEE Transactions on Knowledge and Data Engineering*, 3(4):497–503, December 1991.

[2] D. Bitton and D. J. Dewitt. Duplicate record elimination in large data files. *ACM Transactions on Database Systems*, 8(2):255–265, June 1983.

[3] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[4] D. E. Knuth. *The art of computer programming*, volume 3. Addison Wesley, Reading, Massachusetts, 1973.

[5] I. Munro and P.M. Spira. Sorting and searching in multisets. *SIAM Journal of Computing*, 5(1), March 1976.

[6] A. Shatdal and J. F. Naughton. Adaptive parallel aggregation algorithms. In *Proceedings of ACM SIGMOD 1995 International Conference on Management of Data, San Jose, CA*, pages 104–114, San Jose, CA, May 1995. ACM.

[7] J. Teuhola and L. Wegner. Minimal space, average linear time duplicate deletion. *Communications of ACM*, 34(3):62–73, March 1991.

[8] L. Wegner. Quicksort for equal keys. *IEEE Computer*, C-34(4):362–367, April 1985.

[9] Weipeng P. Yan and Per-Åke Larson. Data reduction through early grouping. In *Proceedings of the 1994 IBM CAS Conference*, Toronto, Ontario, November 1994.