Reihe Informatik
9 / 2002

# Early Grouping Gets the Skew

Sven Helmer   Thomas Neumann
Guido Moerkotte

This page left intentionally blank.

# Early Grouping Gets the Skew

Sven Helmer Thomas Neumann
Guido Moerkotte
helmer|tneumann|moer@pi3.informatik.uni-mannheim.de

Fakultät für Mathematik und Informatik,
University of Mannheim, Germany

## Abstract

We propose a new algorithm for external grouping with a large result set. Our approach handles skewed data gracefully and lowers the amount of random IO on disk considerably. In contrast to existing grouping algorithms, our new algorithm does not require the optimizer to employ complicated or error-prone procedures adjusting the parameters prior to query plan execution. We implemented several variants of our algorithm as well as the most commonly used algorithms for grouping and carried out extensive experiments on both synthetic and real data. The results of these experiments reveal the dominance of our approach. In case of skewed data we outperform the other algorithms by a factor of two.

## 1 Introduction

In database systems, grouping is used for a variety of purposes, including aggregation and duplicate elimination. Although usually a well-behaved operator, grouping becomes a nuisance when the result does not fit into main memory. We propose a new algorithm for grouping that shows good performance for arbitrarily large inputs and handles data skew gracefully. In addition, we show the results of extensive experiments detailing the behavior of our algorithm and several other currently used techniques.

Recently, aggregation and duplicate elimination have become a focus of attention for the database community again. Several researchers investigated the possibilities of improving query evaluation and optimization with regard to aggregation [4, 12]. There has also been work on materializing aggregates for speeding up the processing, especially in a data warehouse environment [2]. Furthermore, there are ongoing activities looking into the exchange of precision for performance [1]. Finally, we mention the studies concerning the extension of standard aggregate functions [5].

Our main concern is the processing of aggregation encompassing a large number of groups, so this work complements the work on query optimization. Our scheme is simple to apply, so adding it to an optimizer is straightforward. Consequently, the optimizer does not need to expend resources on an inevitably inaccurate estimation of parameters. We focus on exact, ad-hoc queries with standard aggregation functions as, for example, used in SQL. So, tradeoffs involving precision and materialization are not in our scope. We concentrate on external aggregation, i.e. the (intermediate) results of the aggregation are several times larger than the available main memory, as the main memory case is well covered by simple hashing.

Up to now only rough estimations have been given for the performance of different aggregation algorithms. The reason for this is that the theoretical analysis of practical cases is very difficult. We implemented different algorithms and compared them to each other under realistic conditions. Here we present some results of our experiments with synthetically generated and real data, which provide new insights into the subject of aggregation.

Moreover, in our opinion the matter of data skew has not received enough attention yet. Often it is assumed that the data is uniformly distributed or that data skew can be effectively counteracted by an adequate hashing scheme [10]. Under realistic conditions this is rarely the case. We show that our scheme is able to handle non-uniformly distributed data very efficiently.

This paper is structured as follows. In Section 2 we describe our approach for combining early aggregation with hashing. A brief introduction to existing aggregation algorithms can be found in Section 3. We specify the environment in which the experiments were conducted in Section 4. The results are explained in Section 5. A summary in Section 6 concludes the paper.

## 2 Our approach

We start by giving a brief outline of the ideas behind our approach in Section 2.1. This is followed by a more detailed look in Section 2.2.

### 2.1 Basic Ideas

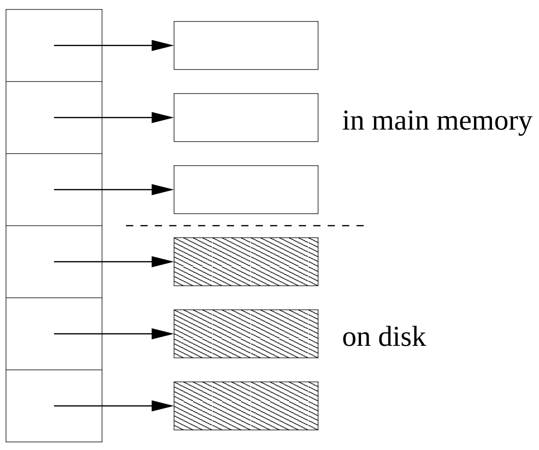

Figure 1: Hybrid Hashing

As there are some similarities between aggregation and join operations, many of the algorithms applied for aggregating tuples bear a semblance to join algorithms. One of the most popular algorithms for aggregation as well as joining is hybrid hashing (for a brief introduction to aggregation algorithms, see Section 3).

However, when joining tuples with hybrid hashing, it does not matter which partitions are swapped to disk and which partitions remain in main memory, as long as the partitions are roughly of equal size. During aggregation and duplicate elimination things look different. We want to keep certain groups in main memory as long as possible. On the one hand, these are groups that have a high reduction factor, i.e. a large number of tuples belonging to these groups exist. The longer we can keep these groups in main memory, the more early aggregation we can achieve. On the other hand, if the data is clustered, we want to keep those groups in main memory, whose cluster we are currently traversing.

If an overflow occurs in hybrid hashing, a whole partition is swapped to disk. Figuratively speaking, we divide the data "horizontally" (see Figure 1). All groups that belong to the same partition (because of their hash value) are swapped to disk and cannot be reduced further, as tuples belonging to these groups are written directly (or via a small buffer) to their corresponding partition on disk. Usually, the hash keys do not respect the reduction factor or clustering of the data (as this is very difficult to achieve in practice), so groups are swapped out arbitrarily from the viewpoint of early aggregation.

In our approach we propose to swap out the data "vertically" (see Figure 2). We keep those groups of

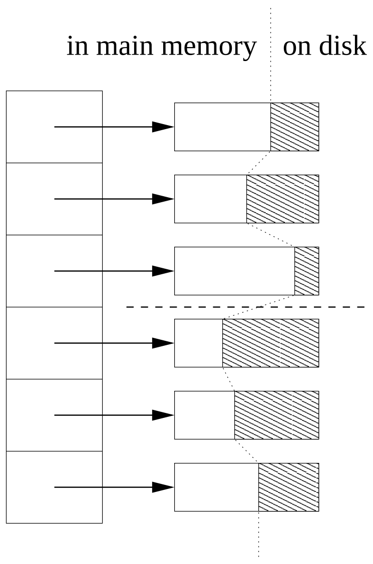

Figure 2: Our approach

each partition in main memory where most of the aggregation takes place and swap out all other groups, i.e. the partitioning scheme does not control which groups are swapped out.

We investigated two principal approaches. In the first approach, we add a counter to each group that registers how many tuples have been aggregated in this group. When an overflow occurs, we sweep through main memory to identify the groups that have aggregated fewer than average tuples. These groups are written to disk and the counters of all the other groups are reset to zero to allow adaption to changing frequencies of tuple values.

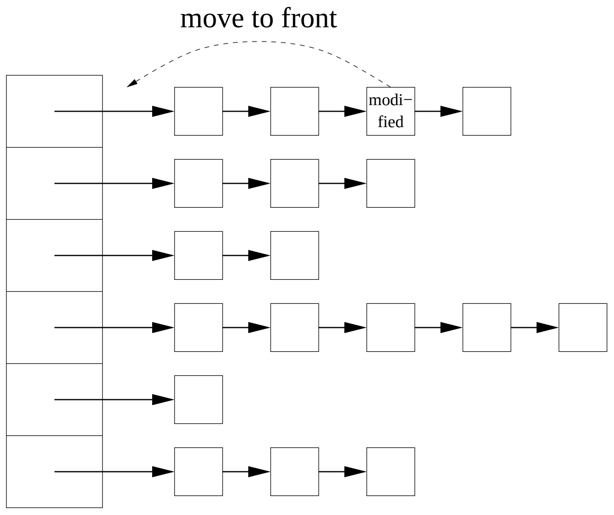

Figure 3: LRU strategy

The second approach uses an LRU strategy (similar to the management of cache lines [14]). The groups are stored in collision chains in the hash table (see Figure 3). Each time a tuple is added to an aggregate, we move the corresponding group to the start of the collision chain. When an overflow occurs, we traverse all collision chains and swap out the entries found towards the end of the chains. We take care to consider the variable lengths of the collision chains. Compared to the first approach, the LRU strategy has the advantage that it needs no additional storage overhead.

After scanning the whole input, we have to start the algorithm recursively for each partition on disk that has not been fully processed yet. We use a smaller number of partitions than hybrid hashing (during the experiments the number of partitions was usually set to four). This keeps the amount of random IO low when writing partitions to disk. We can afford the smaller number of partitions because our algorithm reduces the data much better.

### 2.2 Detailed Description

In the first step of the algorithm we scan the whole input relation, hashing each tuple into a hash table that contains the aggregate values for the groups currently processed. If the corresponding group is present, we update the aggregate with the tuple value. Otherwise a new group is allocated and inserted into the hash table. If there is not enough memory left to allocate a new group, we have to swap out some of the groups. How do we determine the groups that are to be written to disk? As already mentioned we have developed two variants to do this.

In the first variant, each group has a frequency counter that stores the number of tuples aggregated in this group. We go through the hash table and look at the counters of all entries. All groups that aggregated a number of tuples that is below average are marked for replacement. We only have to record the number of groups and the number of processed tuples to do this.

In the second variant of our algorithm, the collision chains of the hash table are simultaneously LRU queues. We mark the entries towards the end of each queue for replacement. We know the number of groups and the number of queues and use this information to calculate the expected queue length $l$. We skip the first $l/2$ entries of the first queue and mark the entries from $l/2 + 1$ to the end for replacement. As it is very improbable that $l/2$ entries will be marked for replacement in this way, we compute the difference of actually marked entries minus $l/2$. This difference is added to the skip value of the next queue, i.e. if we have marked more than $l/2$ entries in a list, we can skip more than $l/2$ entries in the next list. If we have marked less, we need to skip less than $l/2$. We do this to swap out approximately half of the entries without calculating the length of each queue.

After determining which groups to replace we now describe the actual process of swapping. We divide the groups into a small number of partitions $k$. For our experiments this $k$ was usually equal to four. For this we calculate $k - 1$ separators. We looked at two different techniques for determining separators. In the first technique we calculate the medians for dividing the hash keys evenly. The second technique involves dividing the domain into equidistant intervals. In the first step of the algorithm we use the currently known minimum and maximum of the hash keys. This may result in a deviation from the actual values because we have not seen the whole relation yet. In the subsequent (recursive) steps of the algorithm, however, we have the exact values of the minimum and maximum in form of the separators. While the first technique is more accurate, i.e. it divides the data more evenly, the costs of the second technique are much lower. We used the second technique as it was sufficient for our needs.

Next we scan the hash table $k$ times, swapping out the marked groups of one partition each time. Although this results in multiple scans in main memory, we are able to write the data to disk sequentially. The avoidance of random IO compensates for the multiple scans more than enough. Now we have enough memory to continue the aggregation process. If we run out of memory again, we have to apply the steps above over again. We continue until the input stream of tuples ends. Any groups in a partition that remained in main memory during the whole step are completely aggregated and can be output. This concludes the first step of the algorithm.

The groups swapped to disk have to be processed recursively in the further steps, i.e. for each partition we repeat our aggregation algorithm. The only difference to the first step is that we now work with partially aggregated data instead of "raw" tuples. So, when inserting groups into the hash table, we have to combine their aggregate values. Any SQL-like aggregate can be computed in this way (AVG is put together with SUM and COUNT).

## 3 The Competitors

In the following sections we describe different existing approaches for grouping and aggregating tuples. Many of the algorithms used for aggregation and duplicate elimination are similar to techniques applied for joining relations. Nevertheless, we point out two important differences. These differences make it questionable to transfer the knowledge of join algorithms to grouping in a straightforward way.

While join operators combine two or more relations, aggregation operators work on a single relation, i.e. we do not have to coordinate the access and merging of tuples from different relations. Also, almost all join algorithms are divided into two phases: a preprocessing phase (involving the sorting or partitioning of tuples) and the actual join phase. (Notable exceptions to this are nested-loop joins and Diag-Join [11].) These two phases may overlap to some extent (like in hybrid hash joining), but most tuples are joined during the second phase. When grouping tuples, these two phases are not as distinct. We can start aggregating right from the start, collapsing many tuples to a single value.

## 3.1 Nested-loop Grouping

Nested-loop Grouping is the most straightforward way of aggregation [10], in which we accumulate the output in a temporary file. An outer loop traverses the relation and for each tuple, an inner loop searches the output file for a matching aggregate. If we find one, we compute the new aggregate value, else we add a new item to the output.

Clearly, this technique is very inefficient if the result is too large for the available main memory. On the other hand, it can can support unusual aggregations (e.g., where a single tuple contributes to more than one group). However, we focus on standard aggregation types, so we do not consider the nested-loop algorithm further, listing it only for the sake of completeness.

## 3.2 Sort-based grouping

Sorting the tuples prior to the grouping is the traditional way to compute aggregates [8, 10]. The concept of this technique is easy to grasp. First we sort the tuples on their grouping attributes, and then we aggregate all tuples with identical values. This is not difficult because identical values can be found together in one chunk.

As long as all tuples fit into main memory, this approach is quite fast. Otherwise we have to sort externally, which reduces the performance considerably. We can speed up this method by aggregating as early as possible, i.e. while generating the runs [3]. In spite of early aggregation we may still be forced to sort externally, even if the result fits into main memory. This is the case when intermediate results are too large.

We implemented three different variants of sort-based grouping. One does a replacement selection using a weak heap [7] to save CPU-time. No early aggregation was integrated, so this algorithm serves as a reference. The second variant uses the same weak heap as above, but does an early aggregation. We call this algorithm "eager sort". The third and final one applies quicksort (with a median of three) and early aggregation. We call this variant "eager quicksort".

## 3.3 Hash-based grouping

Hash-based grouping follows the same outline as nested-loop grouping. We loop through the tuples of the relation. However, instead of appending new groups to the end of an output file, we store them in a hash table for quick lookups. As long as we can store the result in main memory, we do not need to swap out tuples to disk and a simple hashing scheme suffices. If the result turns out to be too large for the available main memory, we have to use a more sophisticated method.

One such approach is Grace hash [9, 17], where the input is partitioned to disk such that each partition can be grouped in main memory. Compared to hybrid hash grouping, however, Grace hash is inferior, due to a suboptimal IO behavior. Therefore, hybrid hashing is commonly used.

Hybrid hashing [6, 17] tries to keep groups in main memory as long as possible. When space runs out, one partition is swapped to disk. Further tuples belonging to this partition are appended to the file on disk. Ideally, one partition can be kept in main memory during the whole execution of the aggregation. Groups belonging to this partition can be processed immediately. All other partitions are worked off in a second step. The main problem of this approach is determining the number of partitions, such that the resulting groups of each partition fit into main memory. If the data is distributed uniformly, this poses no great problem. In practice, however, this is very seldom the case. Even if statistics on base relations are known, complex queries involving the filtering of tuples might change the distribution drastically. Choosing the wrong number of partitions renders the partitioning scheme suboptimal, because all partitions that cannot be processed in main memory have to be partitioned recursively.

Nakayama, Kitsuregawa, and Takagi propose a dynamic hybrid hash strategy in [13, 15] to adapt to data skew by choosing a large number of partitions. This lowers the probability that a partition will not fit into main memory. In order to accelerate the processing in the second phase of the algorithm, they fill the buffer with partitions as completely as possible and work on those partitions simultaneously. At first glance this approach seems superior to hybrid hashing because it avoids overflowing partitions. On the other hand, it is still not clear how an optimal number of partitions should be determined. It also involves a lot of random IO for writing groups into partitions in the first phase of the algorithm and for reading those groups again in the second phase.

## 3.4 Brief Comparison

Our approach is superior to the sort-based method, because we do not have the overhead of completely sorting the input. The performance of hybrid hashing deteriorates when overflows occur, as the size of the swapped out partitions will not decrease from that point on. The reason for this is that once a partition is swapped to disk, no further aggregation takes place until this partition is read into main memory again [10]. Determining the optimal number of partitions to avoid overflows is very difficult for irregularly distributed data. In our approach, we keep on aggregating the most active groups in all partitions, which reduces their size considerably. Additionally, we adjust the partitioning of the groups according to the data distribution (the calculation of the separators is data-driven rather than space-driven). Nakayama et al. prevent overflows by playing for safety when determining the number of partitions, i.e. they overestimate the number of buckets significantly [13, 15]. This, however, leads to excessive random IO. In contrast to this we write large blocks of groups sequentially to disk.

After this comparison we will give a more thorough experimental evaluation of the different approaches in the following section.

## 4 Environment of Experiments

When comparing algorithms, there are several principal avenues of approach: mathematical modeling, simulations, and experiment. We decided to do extensive experiments, because it is very difficult, if not impossible, to devise a formal model that yields reliable and precise results for non-uniform data distribution and average case behavior. In the following sections we present the system parameters and the specification of our experiments. This is followed by the presentation of the results of the experiments.

### 4.1 System Parameters

We implemented our algorithm (and several other competing algorithms) in our database system SOD [1] and ran extensive experiments. Since the problem of external aggregation is not as straightforward as it may seem at first glance, we compared the algorithms using several different metrics. Among these are overall running time, size of intermediate results, and ratio of random IO to overall IO. Additionally, we employed several different data sets to benchmark the algorithms. For synthetically generated data we utilized the specification of TPC-R (with a scale factor of 1), Zipf-distributed data, and normally distributed data. For real data we had access to a chemical database used at BASF. The experiments were conducted on a lightly loaded PC (1 GHz Athlon processor) with 512 MByte main memory running under Windows NT4.0 SP6. We implemented the algorithms in C++ using the Borland C++ Compiler Version 5.5.

### 4.2 Synthetic Data

We used three different sets of synthetically generated data in our experiments: Zipf-distributed data, normally distributed data, and TPC-R data. In this section we describe the data sets. The groupIDs seen in the figures for synthetically generated data are not necessarily equal to the attribute value we grouped by. We attain the attribute values by hashing the groupID (taking care not to cause any collisions). The position of each tuple in a generated relation is also randomized, i.e. the relations are not sorted in any way.

[1] http://sod.tneumann.de

#### 4.2.1 Zipf-distributed Data

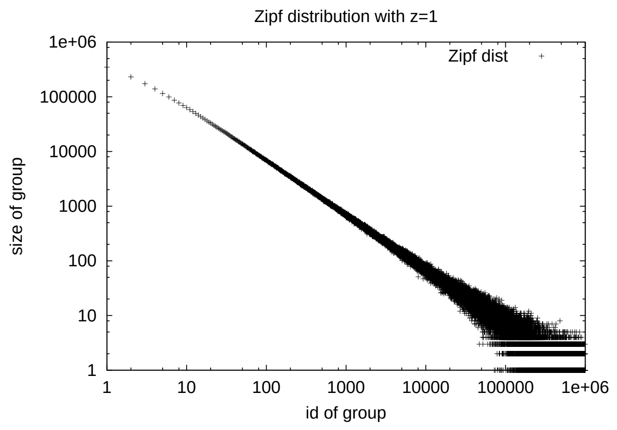

Figure 4: Zipf Distribution

We decided to use a Zipf distribution (with $z = 1$ and $z = 0.5$), since various naturally occurring phenomena exhibit a certain regularity that can be described by it, e.g. word usage or population distribution [16]. A discrete Zipf distribution is defined by $P_z(x)$, which denotes the probability of event $x$ occurring, with $x \in \{1, 2, \dots, n\}$.

$$P_z(x) = \frac{1}{x^z} \cdot \frac{1}{H_n} \tag{1}$$

with

$$H_n = \sum_{i=1}^{n} \frac{1}{i^z} \tag{2}$$

Figure 4 shows the sizes of the groups generated for a Zipf distribution with $z = 1$. Please note the logarithmic scale on the x- and y-axis. The cardinality of the input relation for Zipf-distributed data was 10,000,000 tuples.

#### 4.2.2 Normally Distributed Data

Another distribution that is quite common is the normal distribution. We used the parameters $\mu = 500$K, $\sigma = 50$K and $\mu = 500$K, $\sigma = 75$K, respectively. Figure 5 shows the sizes of the groups generated for $\mu = 500$K, $\sigma = 50$K. The cardinality of the input relation for normally distributed data was 100,000,000 tuples. We increased the cardinality for this distribution to get a substantial number of groups that have more than one element.

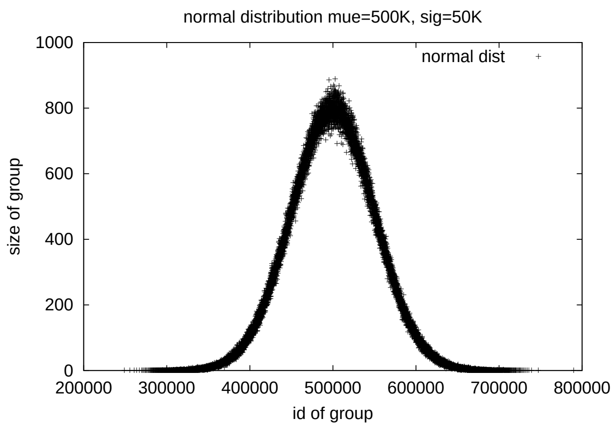

Figure 5: Normal Distribution

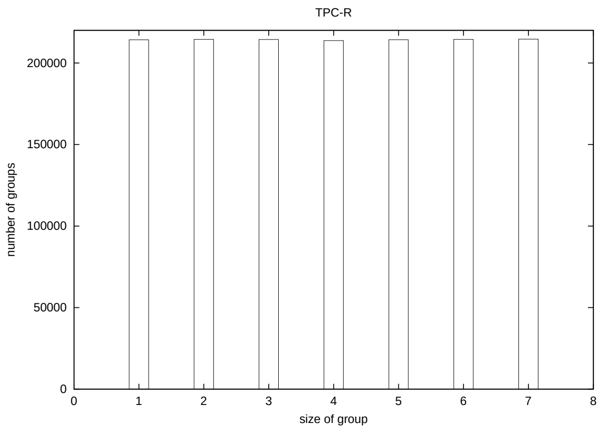

Figure 6: TPC-R

### 4.2.3 TPC-R

For the TPC-R data benchmark the number of groups for each size is nearly uniformly distributed. We have groups with a size ranging from 1 to 7. The differences in frequency of each group size are minimal. Figure 6 depicts the number of groups found for each group size. We used the lineitem relation with a scaling factor of 1, which results in 6,001,215 tuples.

### 4.2.4 Real Data

The real data was taken from a chemical database for structure elucidation [19]. The data has a distribution that is approximately distributed normally in the number of groups that share a certain size (see Figure 7). This is different from the normal distribution in Figure 5 (where the size of the groups is normally distributed). The cardinality of this relation was 5,832,781 tuples.

## 5 Results

We present an excerpt of the results of our extensive experiments emphasizing overall running time, size of intermediate result, and the ratio of random IO to total IO. First we deal with the results for synthetically generated data, in Section 5.2 with those for real data.

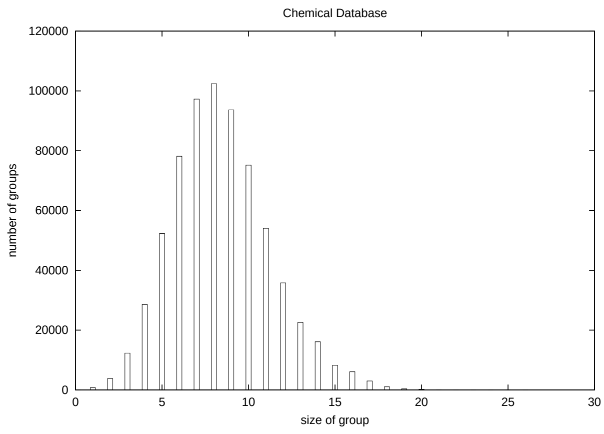

Figure 7: Chemical Data

As units of measurement we use seconds for the overall running time, number of tuples for the size of the intermediate results, and a percentage between 0 and 1 for the ratio of random IO to total IO. We count a page access as random if we access two pages consecutively that are not adjacent to each other on disk, regardless of the buffering done by the buffer manager. Consequently, only the running time is machine dependent.

Each algorithm was given the same amount of main memory for each experimental run. The division of this main memory into buffer space and memory for the actual aggregation was optimized individually for each algorithm. We also tuned the number of partitions for hybrid hashing for each run (determining an optimal value experimentally instead of using the usual formulas). The variants of our algorithm are marked with “ctr hash” (for the variant with counters) and with “lru hash” (for the variant with LRU-queues). For all results presented here, the number of partitions was set to four for our algorithms.

### 5.1 Synthetic Data

#### 5.1.1 Zipf-distributed Data

In Figure 8 the total running time for all algorithms for Zipf distributed data is shown. Our algorithm outperforms the others by at least a factor of two. Hybrid hashing, while starting out strong, starts to degenerate for small memory sizes.

In Figure 9 we plotted the sum of the intermediate results output by all algorithms. Our algorithm performs even better than the eager sorting algorithms, because we do not need to sort the runs. Groups allowing a high reduction that are surrounded by scarcely visited groups have to be kept together in sorted runs, while our algorithm has the freedom to swap out arbi-

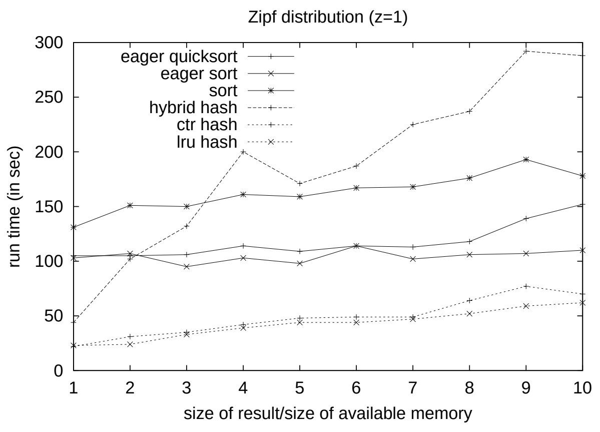

Figure 8: Total Running Time (Zipf)

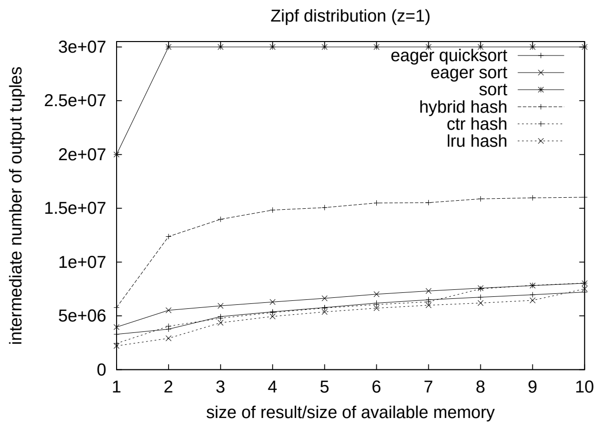

Figure 9: Reduction of Data (Zipf)

trary groups.

We also have the smallest amount of random IO of all algorithms. We have a better ratio than hybrid hashing, because we write out all data sequentially. Reading back in partitions during a recursive step of our algorithm involves some random IO. But as the number of partitions is smaller than that of hybrid hashing, we have less random IO during reads as well. The random IO in the sort-based algorithms is caused by the merge steps, where data from different runs has to be collected. The random IO for hybrid hashing and the sorting algorithms is not as bad as it may seem at first glance. Some of the negative effects of random IO (long seek and latency times) can be compensated by a smart buffering strategy. However, the memory used by the buffer manager is not available for the grouping operator.

The LRU-variant of our algorithm outperforms the counter-based variant in all three different metrics we looked at. We are able to determine which groups to swap out more precisely with counters, but the overhead is not worth it because it reduces the memory

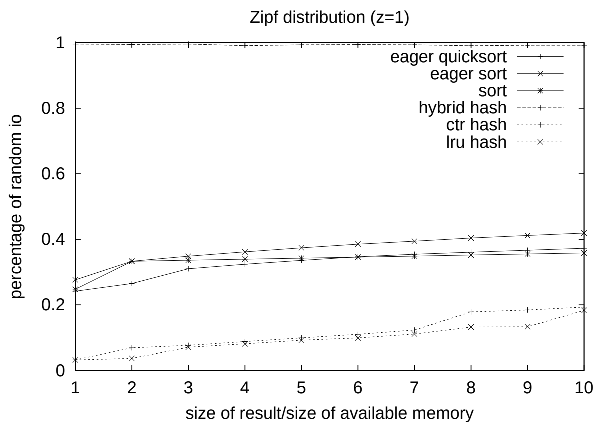

Figure 10: Percentage of Random IO (Zipf)

available for aggregation.

The sort-based aggregation algorithms employing early aggregation outperform the standard sort-based algorithm in all metrics except for percentage of random IO. Although the standard sort-based algorithm has less random IO in relative terms, the absolute amount of IO is much higher (as can be seen by the size of the intermediate results in Figure 9).

### 5.1.2 Normally Distributed Data

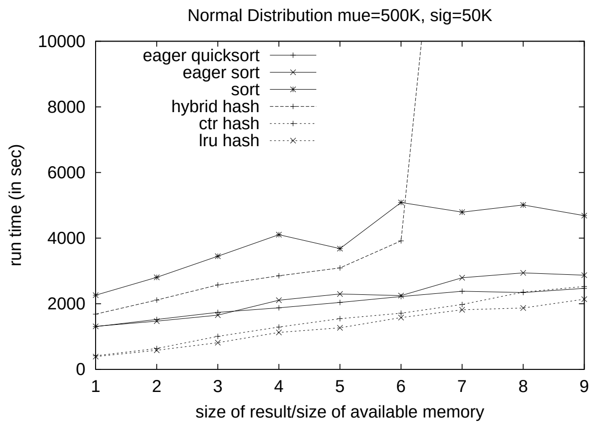

Figure 11: Total Running Time (Normal)

The results for normally distributed data are similar to those for the Zipf-distributed data. The LRU-variant of our algorithm is still the fastest in terms of total processing time (see Figure 11). Hybrid hashing shows the same problems with small memory sizes, while the eager sorters outperform the standard sorting algorithm.

For small memory sizes the reduction factor of hybrid hashing approaches almost the level of the eager sorters and our algorithms (see Figure 12). As the normally distributed data is not as heavily skewed as

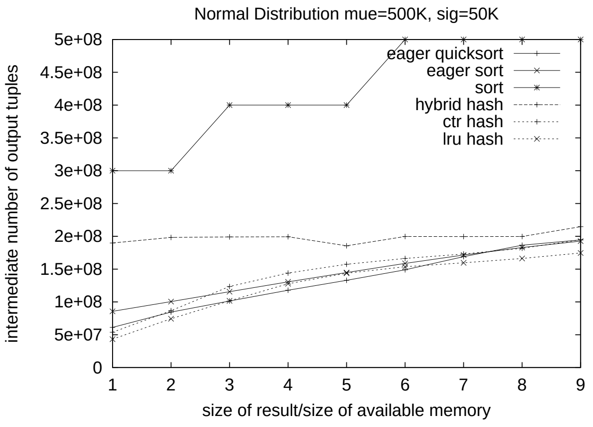

Figure 12: Reduction of Data (Normal)

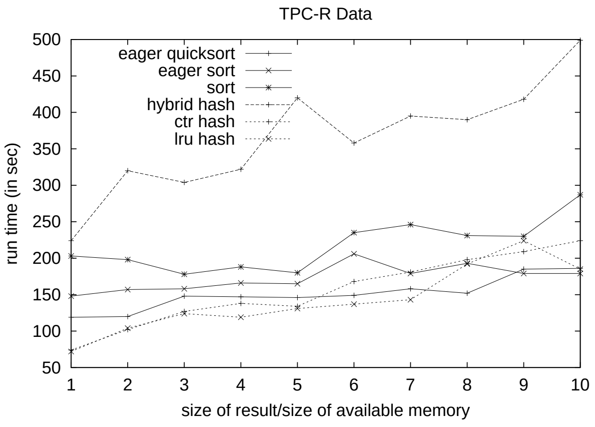

Figure 14: Total Running Time (TPC-R)

the Zipf-distributed data, the early aggregation is not quite as effective.

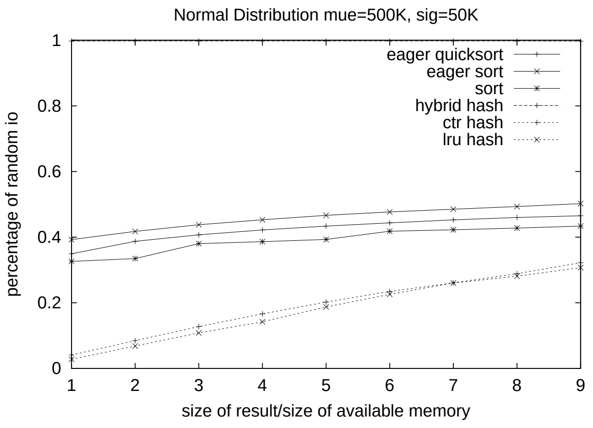

Figure 13: Percentage of Random IO (Normal)

While hybrid hashing is able to reduce the data much better, the amount of random IO is even larger (see Figure 13).

Furthermore, the LRU-variant of our algorithm has proven to be superior to the counter-variant in all three metrics again.

### 5.1.3 TPC-R Data

The TPC-R data is as close to uniformly distributed data as we get. Moreover, the size of the groups is very small (in the range from one element to seven elements). This represents a kind of worst case for early aggregation algorithms. The drastic improvement of the standard sorting algorithm illustrates this fact (see Figure 14).

The reduction factor of our algorithms deteriorates as can be seen in Figure 15, but the percentage of random IO is still very low (see Figure 16). This make it possible for our algorithms to stay on a par with the

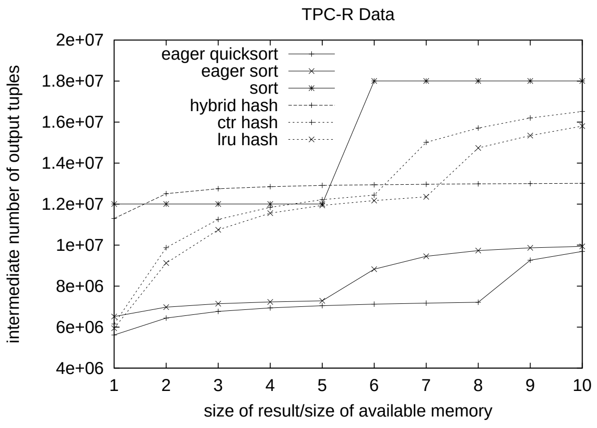

Figure 15: Reduction of Data (TPC-R)

others. Nevertheless, the LRU-variant still bests the counter-variant.

## 5.2 Real Data

Judging from the behavior of the algorithms, the real data has to be classified as lying between uniformly and heavily skewed data.

Although not being on top in terms of data reduction (see Figure 18), the LRU-variant of our algorithm outperforms all other algorithms (see Figure 17). Partly this has to do with the superior IO behavior (see Figure 19).

## 6 Conclusion

Our contribution is twofold. On the one hand, we illustrated that early aggregation is very effective in realistic cases. For sort-based aggregation this has been suspected [10]. On the other hand, we proposed a new algorithm that combines hashing and early aggregation in a clever way using an LRU strategy. Our algorithm regularly outperforms the other methods in

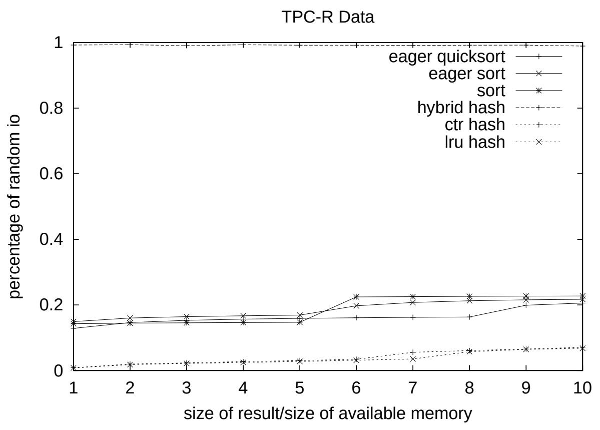

Figure 16: Percentage of Random IO (TPC-R)

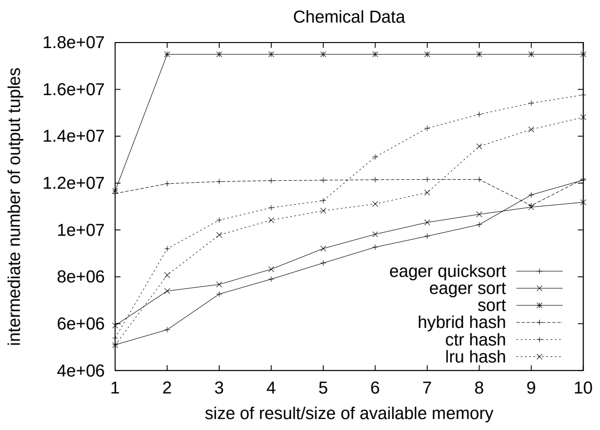

Figure 18: Reduction of Data (Real)

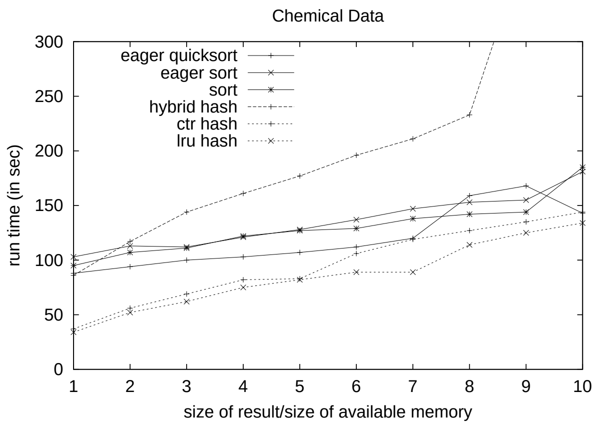

Figure 17: Total Running Time (Real)

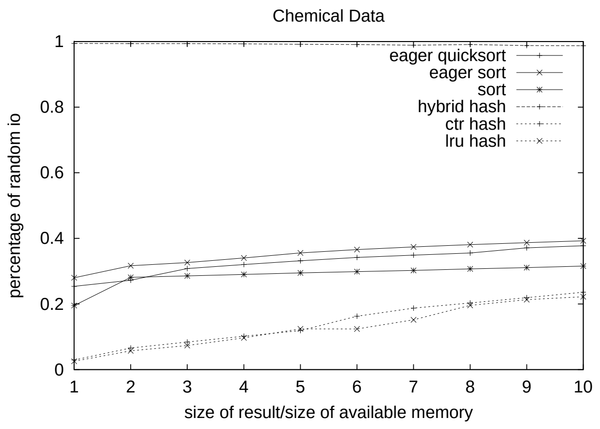

Figure 19: Percentage of Random IO (Real)

terms of total processing time (by up to a factor of two). Although the algorithm does not always have the best reduction factor, this is compensated for by a very small ratio of random page accesses to total page accesses. Additionally, it gets by with a very small number of partitions (four was sufficient in our case), which means that an optimizer does not risk a suboptimal execution of the aggregation operation due to a miscalculation of the parameters.

We plan to adapt our algorithm to parallel execution. As our algorithm has a distinct separation between CPU phases and IO phases, the CPU phases of one process can be interleaved with the IO phases of another process. We are also searching for faster main memory data structures. At the moment we are experimenting with exchanging the hash table with a splay [18]. Extending the algorithm for processing non-standard, overlapping groups using order-preservable hashing is also conceivable.

## Acknowledgments

We thank Martin Will of BASF for providing us with the chemical data.

## References

[1] S. Acharya, P.B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *19th SIGMOD Conference*, pages 487–498, Dallas, Texas, 2000.

[2] S. Agarwal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. 22nd Int. Conf. Very Large Databases, VLDB*, pages 506–521, Mumbai (Bombay), India, 1996.

[3] D. Bitton and D.J. DeWitt. Duplicate record elimination in large data files. *Database Systems*, 8(2):255–265, 1983.

[4] S. Chaudhuri and K. Shim. An overview of cost-based optimization of queries with aggregates. *Data Engineering Bulletin*, 18(3):3–9, 1995.

[5] S. Cluet and G. Moerkotte. Efficient evaluation of aggregates on bulk types. In *Workshop on Database Programming Languages*, page 8, 1995.

[6] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 1–8, 1984.

[7] S. Edelkamp and I. Wegener. On the performance of weak-heapsort. In *17th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 254–266, Lille, France, 2000.

[8] R. Epstein. Techniques for processing of aggregates in relational database systems. Technical report, University of California, Berkeley, California, 1979.

[9] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the systems software of a parallel relational database machine: GRACE. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 209–219, 1986.

[10] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):75–170, June 1993.

[11] S. Helmer, T. Westmann, and G. Moerkotte. Diag-join: An opportunistic join algorithm for 1:n relationship. In *Proc. of the 24th VLDB Conference*, pages 98–109, New York, August 1998.

[12] A. Kemper, D. Kossmann, and C. Wiesner. Generalized hash teams for join and group-by. In *Proc. of the 25th VLDB Conference*, pages 30–41, Edinburgh, 1999.

[13] M. Kitsuregawa, M. Nakayama, and M. Takagi. The effect of bucket size tuning in the dynamic hybrid grace hash join method. In *14th Conference on Very Large Data Bases*, pages 257–266, Amsterdam, Netherlands, 1989.

[14] M. Moudgill. Techniques for fast simulation of associative cache directories. Technical report, IBM, 1998.

[15] M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-partitioned join method using dynamic destaging strategy. In *14th Conference on Very Large Data Bases*, Los Angeles, California, 1988.

[16] V. Poosala. Zipf's law. Technical report, University of Wisconsin Madison, 1995.

[17] L.D. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3):239–264, September 1986.

[18] D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[19] M. Will, W. Fachinger, and J.R. Richert. Fully automated structure elucidation - a spectroscopist's dream comes true. *J. Chem. Inf. Comput. Sci.*, 36:221–227, 1996.