Generalized Hash Teams for Join and Group-by*

Alfons Kemper Donald Kossmann Christian Wiesner

Universität Passau
Lehrstuhl für Informatik
94030 Passau, Germany
$\langle$lastname$\rangle$@*db.fmi.uni-passau.de*

Abstract

We propose a new class of algorithms that can be used to speed up the execution of multi-way join queries or of queries that involve one or more joins and a group-by. These new evaluation techniques allow to perform several hash-based operations (join and grouping) in one pass without repartitioning intermediate results. These techniques work particularly well for joining hierarchical structures, e.g., for evaluating functional join chains along key/foreign-key relationships. The idea is to generalize the concept of hash teams as proposed by Graefe et.al [GBC98] by indirectly partitioning the input data. Indirect partitioning means to partition the input data on an attribute that is not directly needed for the next hash-based operation, and it involves the construction of bitmaps to approximate the partitioning for the attribute that is needed in the next hash-based operation. Our performance experiments show that such generalized hash teams perform significantly better than conventional strategies for many common classes of decision support queries.

1 Introduction

Decision support is emerging as one of the most important database applications. Managers of large businesses, for example, want to study the development of *sales* for certain *products* by *region*, and they expect the database system to return the relevant information within seconds or at most few minutes.

Decision support typically involves the execution of complex queries with join and group-by operations. To support these kinds of queries, database vendors have significantly extended their query processors and researchers have just recently developed a large variety of new query processing techniques; e.g., the use of bitmap indices, special joins that exploit bitmap join indices, new join methods [HWM98, CKK98], or multi-query optimization for decision support to name just a few. In addition, a whole new industry, data warehouses, has appeared with products that materialize (i.e., pre-compute) query results and cache the results of queries. Furthermore, the TPC-D benchmark [TPC95] has been proposed in order to evaluate the performance of a database product for decision support queries.

In this work, we present a new class of algorithms that can be used to speed up the execution of decision support queries that involve one or more joins and a group-by operation. The idea is to partition the input data and, then, carry out all join and group-by operations in one pass. To make this possible, we propose to construct bitmaps in the partitioning phase of a table and use these bitmaps in the partitioning phase of other tables. The advantage of our approach is that a great deal of disk IO can be saved, if the data base and intermediate query results do not fit into the available main memory: only one partitioning step per table is required, rather than partitioning the inputs of every join and group-by operation individually, as done by conventional query execution engines today. Due to the use of bitmaps, however, our approach might suffer from so-called *false drops* in the partitioning phase and result in overall increased disk IO and CPU cost in certain cases. A query optimizer should, therefore, enumerate query evaluation plans based on our new approach in addition to traditional query evaluation plans, and we will give formulae that can be used by an optimizer in order to decide when to use our approach.

Our approach can be seen as a generalization of hash teams, as proposed in [Gra94, GBC98]. Our technique adopts the main idea of hash teams to partition base data once and carry out joins in one pass afterwards. Hash teams, however, can only be applied if all the joins within a team are carried out using the same join/group-by columns. Our approach, on the other hand, can be applied to any kind of (equi) join, and it works best for joining hierarchical structures, e.g., for evaluating functional join chains along key/foreign-key relationships. We, therefore, refer to our approach as *generalized hash teams*.

The remainder of this paper is organized as follows:

*This work was partially supported by the German National Research Council DFG Ke 401/7–1

Proceedings of the 25th VLDB Conference,
Edinburgh, Scotland, 1999.

In Section 2 we introduce the use of generalized hash teams by way of a simple binary join followed by a grouping/aggregation. Section 3 provides more details on implementing generalized hash teams. In Section 4 the application of generalized hash teams for multi-way joins with or without a subsequent grouping is described. In Section 5 the number of false drops resulting from the indirect, bitmap-based partitioning is analyzed. In Section 6 a few representative decision support queries are benchmarked. Section 7 compares our work to other related proposals, and Section 8 concludes this paper with a summary.

2 Binary Joins with Aggregation

In this section, we will show how generalized hash teams work for queries that involve one join and one group-by operation. We will, furthermore, present a simplified variant, called *partition nested loops*. As a running example, we will use the following query which asks for the total *Value* of all *Orders* grouped by the *Customer City*.

Query 1: **select** c.City, **sum**(o.Value)
from Customer c, Order o
where c.C# = o.C#
group by c.City;

2.1 Generalized Hash Teams

The traditional (state-of-the-art) plan to execute our example query is shown in Figure 1. This plan uses hashing in order to execute the join and the group-by operation. This plan would first partition (abbreviated *ptn* in the figures) both the *Customer* and the *Order* tables by $C\#$ such that either all the *Customer* or all the *Order* partitions fit in memory; that is, this plan would carry out a (grace or hybrid) hash join between these two tables [Sha86]. After that, the traditional plan would use hashing (possibly with early aggregation [Lar97]) to group the results of the join by *City*. If there are more *Cities* than fit into main memory, this group-by operation would, again, involve partitioning such that every partition can be aggregated in memory. In all, there are three partitioning steps in this traditional plan, incurring IO costs to write and read the *Customer* table, the *Order* table, and the result of the join. As an alternative, *sorting*, rather than *hashing*, can be used for the join and/or the group by. In many cases, sorting has higher (CPU) cost than hashing; in any case, however, a traditional plan based on sorting would also involve IO costs to write and read the *Customer* table, the *Order* table, and the result of the join.

Figure 2 shows a plan that makes use of generalized hash teams in order to execute our example query. Like the traditional plan shown in Figure 1, this plan is based on hashing to execute the join and the group-by operation. The trick, however, is that the *Customer* table is partitioned by *City*, rather than by *C#*, so that the result of the join is partitioned by *City* as well and the group-by operation does not require an additional partitioning step. To make this work, this plan generates bitmaps while partitioning the *Customer* table. These bitmaps indicate in which partition every *Customer* tuple is inserted and these bitmaps are used to partition the *Order* table so that *Order* tuples and matching *Customer* tuples can be found in corresponding *Order* and *Customer* partitions. That is, the *Order* table is partitioned *indirectly* using the bitmaps.

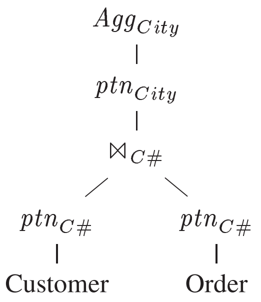


Figure 1: Traditional Plan

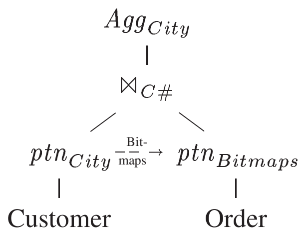


Figure 2: Generalized Hash Team

To make this clearer, let us look at Figure 3 which illustrates the whole process in more detail. The figure shows a small example extension of the *Customer* table and how this *Customer* table is partitioned by *City* into three partitions: the first partition contains all *Customers* located in PA and M, the second partition contains all *Customers* located in B and HH, and the third partition contains all *Customers* located in NYC and LA. Just as in a traditional (grace or hybrid) hash join, the goal is to generate partitions that fit into main memory, and database statistics would be used for this purpose. Corresponding to every partition, there is one bitmap that keeps track of the *C#*'s stored in the partition; in this small example, there are three bitmaps of length ten each. If a *Customer* tuple is inserted into a partition, the 1 + (*C#* `mod 10`)'th bit of the corresponding bitmap is set. So, the fourth and sixth bit of the first bitmap are set because the first partition contains *Customer* tuples with *C#* = 5, 13, 25, and 23. Likewise, the first, third, seventh, and tenth bit are set in the second bitmap.

The next step is to partition the *Order* table using the bitmaps. To see how, let us look at the first *Order* tuple which refers to *Customer* 4. This *Order* is placed into the third *Order* partition because the bit at position $1 + (C\# \bmod 10) = 5$ of the third bitmap is set. Likewise, the second *Order* which refers to *Customer* 9 is placed into the second partition, and the third *Order* which refers to *Customer* 25 is placed into the first partition. Following this approach, all *Orders* which refer to *Customers* stored in the first *Customer* partition are placed into the first *Order* partition, and the equivalent holds for *Orders* referring to *Customers* of the second and third *Customer* partitions. Thus, the query result can be computed by joining in memory the first *Order* partition with the first *Customer* partition, thereby immediately carrying out the aggregation in memory, and then doing the same procedure with the second and third *Order* and *Customer* partitions.

It is important to notice that in certain cases, *Order* tuples must be placed into two or even more *Order* partitions. In Figure 3, for instance, *Order* 10 (highlighted by bold face) is placed into the first and third *Order* partitions because this *Order* refers to *Customer* 3 and the fourth bit of

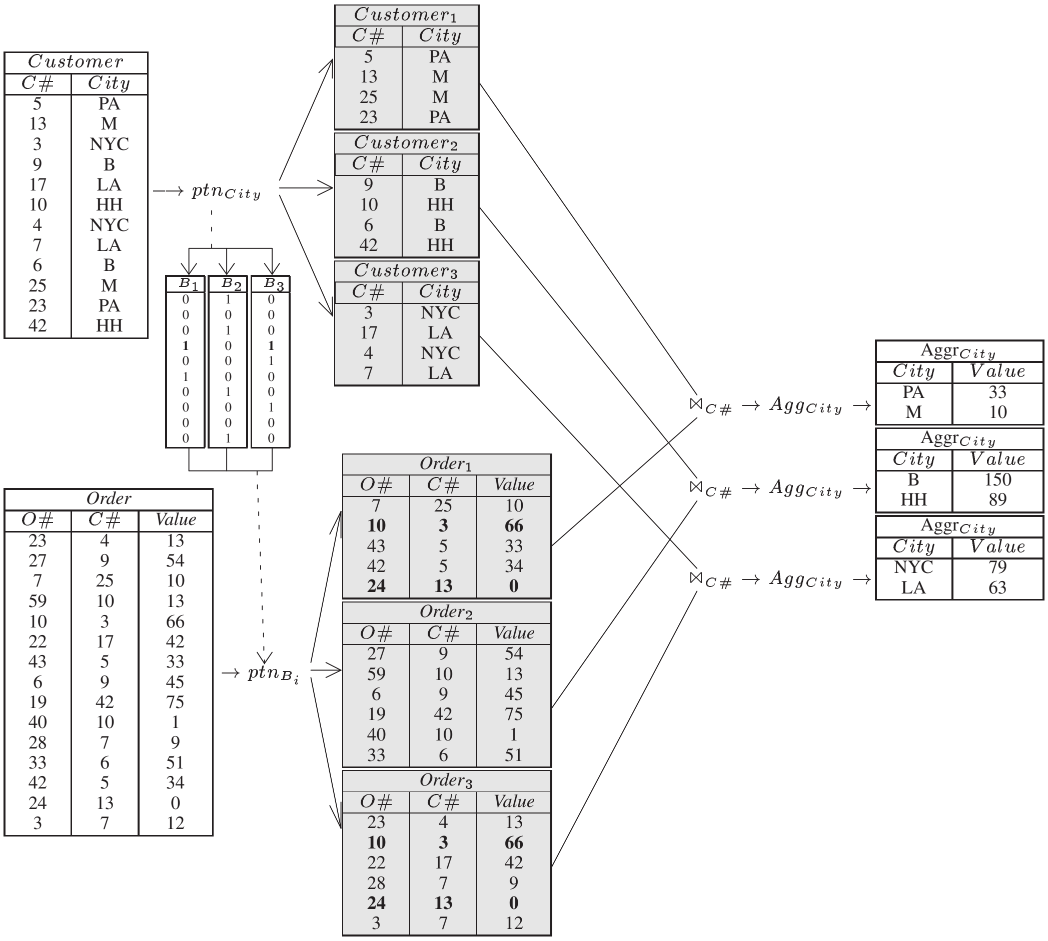


Figure 3: Example Execution of a Generalized Hash Team

the first and third bitmaps are set. We refer to the accidental placement of *Order* 10 in the first *Order* partition as a *false drop*. False drops do not jeopardize the correctness of the overall approach for regular joins because they are filtered out in the join phase[1], but false drops do impact the performance: the more false drops, the higher the IO cost to partition and re-read the *Orders*. The number of false drops depends on the length of the bitmaps, and we will give formulae that can be used in a cost model of a query optimizer in Section 5. Furthermore, *Order* duplicates occur if *Customer* tuples with the same *C#* are placed into different *Customer* partitions. Such a situation does not arise in our example query because *C#* is the key of the *Customer* table. In general, such situations cannot arise if there is a functional dependency between the join attribute (i.e., *C#*) and the partitioning attribute (i.e., *City*). In the absence of such a functional dependency, *Orders* must be duplicated in order to find their join partners in the different *Customer* partitions. In the remainder of this paper, we will assume that such a functional dependency exists or that there is at least a strong correlation between the join and partitioning attributes, and we recommend not to use generalized hash teams in other cases. One example, in which generalized hash teams are not appropriate, according to this criterion, would be a query in which the key of the group-by operation involves a column of the *Order* table, e.g., *OrderDate*.

To summarize, generalized hash teams save disk IO costs for partitioning intermediate query results if these intermediate results do not fit into the available main memory. On the negative side, generalized hash teams require additional main memory for the bitmaps in the partitioning phase, they might involve additional disk IO due to false drops, and they involve additional CPU costs to construct and use the bitmaps. Also, the application of generalized hash teams should be limited to situations in which a functional dependency can be inferred from the join attributes to the partitioning attributes. The optimizer of a database system should, therefore, be extended to enumerate generalized hash team plans (where applicable) in addition to traditional query plans.

2.2 Partition Nested Loop Joins

We now turn to another (novel) approach to execute our example query; we refer to this approach as *partition nested loops*. As with generalized hash teams, the key idea is to partition the *Customers* by *City* before the join so that the

[1] Outer joins cannot always filter out false drops so that generalized hash teams are not directly applicable for all outer join queries.

group-by operation does not require an additional partitioning step. In this approach, however, the join is carried out as a (blockwise hashed) nested loop join rather than using a (grace or hybrid) hash join, and the partition nested loop join approach is somewhat simpler than generalized hash teams because no bitmaps need to be constructed.

In detail, partition nested loop joins work as follows for our example query:

1. partition the *Customer* table by *City* into memory-sized partitions (as for generalized hash teams or any traditional hash join, if *City* were the join column)

2. read the *Order* table, project out the relevant columns (i.e., *C#* and *Value*), apply *Order* predicates (if any), and write the *reduced Order* table to disk

3. read the first *Customer* partition into memory and build a main-memory hash table on the *C#* column. Read the *reduced Order* table from disk and find the *Orders* that refer to the *Customers* of the first partition using the main-memory *C#* hash table. Carry out the aggregation on the fly. (Details on this step can be found in Section 3.2.)

4. repeat Step 3 for the second, third, fourth, and so on *Customer* partition.

Step 2 and Step 3 for the first *Customer* partition can be carried out together in order to save disk IO costs. If no or only marginal selections and projections are applicable on the *Order* table, then Step 2 can be omitted altogether and Step 3 is carried out using the full *Order* table.

The tradeoffs between generalized hash teams and partition nested loop joins are fairly much the same as between (grace and hybrid) hash joins and blockwise nested loop joins; see, e.g., [HR96, HCLS97]. If the *Customer* table is large and must be partitioned into many partitions, partition nested loop joins are likely to perform poorly for re-reading the reduced *Order* table many times. On the other hand, partition nested loop joins might perform better than generalized hash teams if it can be expected that there are many false drops. Also, of course, generalized hash teams require more main memory for the bitmaps in the partitioning phase. This additional main memory, however, is really only needed in the partitioning phase which usually requires much less main memory than the join phase (or the group-by operation). So, the bitmaps can be stored in the *extra* space which is allocated for the join but not needed during the partitioning phase so that the overall main memory requirements of the join and the whole query do not increase.

3 Implementation Details

In this section we will describe the indirect partitioning of generalized hashed teams and the actual execution (join and grouping phase) of generalized hash teams and partition nested-loop joins in more detail.

3.1 Fine-Tuning the Indirect Partitioning Phase

We will use our *Customer* and *Order* example schema to illustrate this discussion. In the initial partitioning step the *Customer* table (abbreviated *C*) is partitioned according to the *City*-attribute into n partitions $C_1, \ldots, C_n$. For this purpose some partitioning (hash) function p is needed that maps *City*-values into $\{1, \ldots, n\}$. For each partition C_i a separate bitmap B_i of length b is maintained to approximate the partitioning of the *C#* values. These bitmaps are initialized to 0. For setting and probing these bitmaps a second hash function, say h is needed that maps *C#* values into $\{1, \ldots, b\}$. Now, consider a particular element $c \in C$: it is inserted into the i*-th* partition C_i for $i = p(c.City)$ and the k*-th* bit of B_i is set where $k = h(c.C\#)$. So, the first partitioning of *C* is done as follows:

forall $c \in C$ **do**
  $i := p(c.City)$;
  $k := h(c.C\#)$;
  insert c into C_i;
  $B_i[k] := 1$;
od

Having partitioned C into $C_1, \ldots, C_n$ the n bitmaps $B_1, \ldots, B_n$ approximate the partitioning function for *Customer* on *C#*. Then, when partitioning the *Order* table (abbreviated *O*) into $O_1, \ldots, O_n$ any element o has to be inserted into partition O_i if the $h(o.C\#)$*-th* bit of the i*-th* bitmap B_i is set. Due to false drops, it is possible that an *Order* o is placed into more than one partition. Thus, the partitioning function for *Orders* is as follows:

forall $o \in O$ **do**
  $k := h(o.C\#)$;
  forall $i \in \{1, \ldots, n\}$ **do**
    if $(B_i[k] = 1)$ insert o into O_i;
  od
od

We can tune this basic partitioning code in two ways: First, we can identify those *O*-objects for which the inner loop can be exited early. Second, we can increase the cache locality when accessing the bitmaps.

Short-Cuts in the Inner Partitioning Loop

There are two kinds of objects for which the inner partitioning loop can be entirely bypassed or exited early:

1. *Objects Without Join Partner*: For those $o \in O$ that definitely do not have a join partner in C we need not execute the inner loop at all. We will compute a separate bitmap, called *used*, to identify those objects. (This kind of bitmap has also been proposed to speed up traditional hash join operations [Bra84].)

2. *Objects Without Collisions*: For those $o \in O$ that are definitely not inserted into more than one partition (i.e., objects that won't drop into a false partition) we can exit the inner loop as soon as they are inserted into some partition C_i. Again, we maintain a separate bitmap, *collision*, for identifying these objects.

The $used$ bitmap can easily be computed as follows:

$$used := B_1 \mid B_2 \mid \ldots \mid B_n$$

where $|$ denotes the componentwise *or* operation.

The *coll* bitmap is set at position k if two (or more) bitmaps B_i and B_j are set at position k, that is:

$$coll[k] := \begin{cases} 1 & : \text{ if there exists } i \neq j \in \{1, \ldots, n\} \\ & \quad \text{ such that } B_i[k] = B_j[k] = 1 \\ 0 & : \text{ otherwise} \end{cases}$$

In our system, both bitmaps are actually computed during the partitioning of the *Customer* table. For our example the two auxiliary bitmaps are shown below:

$used$	$coll$	B_1	B_2	B_3
1	0	0	1	0
0	0	0	0	0
1	0	0	1	0
1	1	1	0	1
1	0	0	0	1
1	0	1	0	0
1	0	0	1	0
1	0	0	0	1
0	0	0	0	0
1	0	0	1	0

The tuned partitioning pseudo code for the *Orders* then looks as follows:

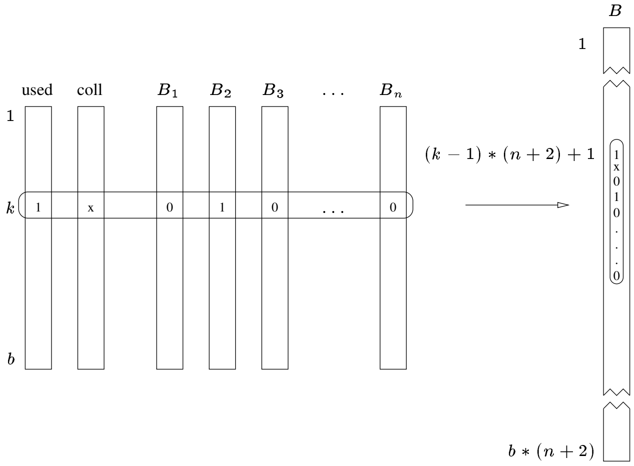
```
forall o ∈ O do
  k := h(o.C#);
  if (used[k] = 0) // definitely no join partner, proceed
                     with next o ∈ O
    continue;
  if (coll[k] = 0) // no collisions
    forall i ∈ {1, ..., n} do
      if (B_i[k] = 1)
        { insert o into O_i;
          break; } // this was the one and only, proceed
                     with next o ∈ O
    od
  else // collisions and false drops
    forall i ∈ {1, ..., n} do
      if (B_i[k] = 1)
        insert o into O_i;
    od
od
```

Increasing Locality on Bitmaps

We can also tune the storage structure of the bitmaps in order to increase cache locality. We observe that the code for partitioning O accesses sequentially the k-*th* position of every bitmap, *used*, *coll*, $B_1, \ldots, B_n$. This observation allows us to achieve higher cache locality. Let's view the $n + 2$ bitmaps of length b as a two-dimensional array with $n+2$ columns and b rows. To achieve higher cache locality we store this array in a single bitmap B of length $(n + 2) * b$ by mapping the two-dimensional array in *row major sequence* into a one-dimensional vector. Then, the only bitmap B contains the elements in the following order

$$\begin{array}{rllll} B = [& u[1], & c[1], & B_1[1], B_2[1], \ldots, B_n[1], \ldots \\ & u[k], & c[k], & B_1[k], B_2[k], \ldots, B_n[k], \ldots \\ & u[b], & c[b], & B_1[b], B_2[b], \ldots, B_n[b] &] \end{array}$$

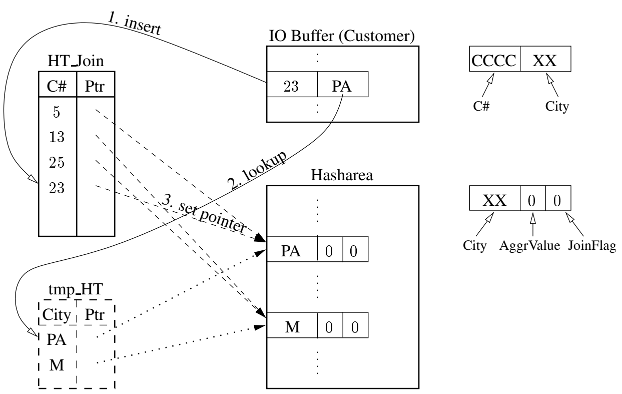
That is, $u[k]$ is found at position $B[(k-1)*(n+2)+1]$, $c[k]$ at $B[(k-1)*(n+2)+2]$, and $B_i[k]$ at $B[(k-1)*(n+2)+2+i]$. This way, the inner partitioning loop for the *Orders* can typically be carried out with a single processor cache miss. The resulting organization is illustrated below:



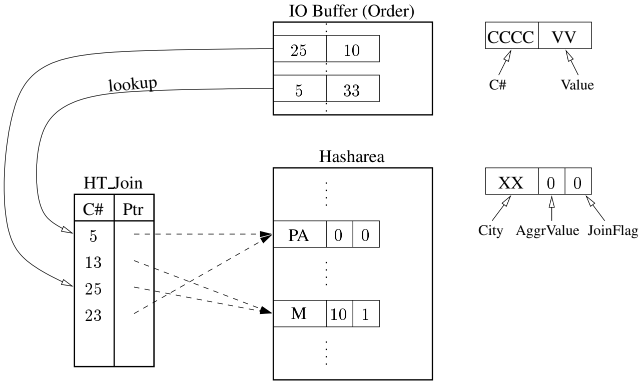
3.2 Teaming up the Hash Join and the Aggregation

The bitmap-based partitioning of O and C is the prerequisite for teaming up the hash join and the grouping/aggregation operator such that the join operator can directly deliver its result tuples to the aggregation operator—without having to repartition the data and write it to disk. The straight-forward implementation requires two separate hash tables: one hash table on $C_i.C\#$ for performing the join with the probe input O_i and a second hash table on $C_i.City$ for grouping/aggregating the join result. These two operators have to be managed by a so-called "team manager"—as it was called in [GBC98]—such that they switch to the next partition synchronously.

We will now devise a further optimization which is based on combining the join and the aggregation operator such that they share a common hash table on the build input C. This is illustrated in Figure 4. Let us first concentrate on the build phase during which the hash table for the i-*th* partition C_i is constructed—shown in Figure 4(a). While loading the partition C_i, two hash tables are maintained: one hash table called *HT_Join* on the join column $C_i.C\#$ and a second, temporary hash table, called *tmp_HT*, on the grouping column $C_i.City$. Both hash tables contain pointers into the *Hasharea* in which the group entries of the join/aggregation query are constructed. That is, the *Hasharea* will contain one entry for every *City* value of partition C_i. Let's look at a particular build input tuple $c \in C_i$ of the form $c = [C\# = 23, City = PA]$ and trace how it is installed in the hash tables and the hash area. First, its $C\#$ value, 23, is inserted into the *HT_Join* hash table; second, the aggregation tuple for its *City* value, *PA*,



(a) Build Phase



(b) Probe Phase

Figure 4: Implementation of the Hash Tables

is looked up via the *tmp_HT* hash table. If this was the first C_i tuple with *City=PA*, a new group entry is installed in the *Hasharea* and the corresponding pointer is inserted into the tmp_HT. Third, the pointer to this group entry of the *Hasharea* is installed in the *HT_Join* hash table. After inserting all tuples of the current build input partition C_i, the probe phase with partition O_i of the probe input starts—shown in Figure 4(b). Let's now trace the *Order* tuple $[C\# = 25, \mathit{Value} = 10]$: The *HT_Join* hash table is inspected and the pointer to the group entry in the *Hasharea* is traversed. The *Value* is added to the *AggrValue* and the *JoinFlag* is set to indicate that the group entry "has found" a join partner (otherwise it would be discarded from the result when flushing the *Hasharea* of the i*-th* partition). After the current probe partition is exhausted, the result tuples are retrieved ("flushed") from the *Hasharea* and the computation of the next *Customer/Order* partitions starts.

While this organization sounds complicated at first glance, it is very easy to implement. The advantages are that a great deal of main memory is saved because long strings with, say, *City* names need only be stored once in the *Hasharea* rather than for each *Customer* individually, and that a great deal of CPU costs is saved in many cases because hashing by *City* is carried out once for every *Customer* rather than once for every tuple of the result of the *Customer* $\bowtie$ *Order*. This organization can be used for generalized hash teams as well as for partition nested-loop joins.

4 Multi-Way Joins

Generalized hash teams and partition nested-loop joins can also be applied to multi-way joins. In the following, we will discuss this for generalized hash teams. (For partition nested-loop joins the tradeoffs are similar so we will omit the discussion for brevity.) For illustration, let us look at the following SQL query:

Query 2: **select** c.City, **sum**(l.Price)
from Customer c, Order o, Lineitem l
where c.C# = o.C# **and** l.O# = o.O#
group by c.City;

This is a three-way (functional) join of *Customer*, *Order*, and *Lineitem* followed by a grouping on the *City* attribute of *Customer*. Generalized hash teams are applicable by partitioning the *Customer* table by *City*, thereby constructing bitmaps in order to guide the partitioning of the *Order* table, as in the binary case described in Section 2. While partitioning the *Order* table another set of bitmaps is constructed and this set of bitmaps is then used to partition the *Lineitem* table. After that, corresponding *Customer, Order*, and *Lineitem* partitions can be joined and the result can be aggregated in one pass in memory. After partitioning, the join can be carried out in any particular order. Figure 5 shows two possible join orders for our example; the polygons surround a team of three operators. In the first plan, the *Customer-Order* join is carried out first; in the second plan, the *Order-Lineitem* join is carried out first. One of the arguments of the first join serves as the probe input of the whole team. In our example query *Lineitem* is the best choice as the probe input, because of its high cardinality, so that the second plan of Figure 5 would be better than the first plan.

It should be noted that the memory requirements of generalized hash teams increase with the number of operations that are teamed up. In our example, if *Lineitem* is chosen as probe input we need to keep information of all *Orders*, *Customers*, and *Cities* of a partition in memory as part of executing the team. (Our special organization described in Section 3.2, however, does help to reduce the memory requirements.) In the partitioning phase, memory for two sets of bitmaps are required: While partitioning the *Orders*, the *Customer* bitmaps must be probed and the *Order* bitmaps must be constructed; when partitioning the *Lineitems*, only the *Order* bitmaps are relevant (the *Customer* bitmaps can be discarded at that point).

This Query 2 is a "classical" case for employing generalized hash teams because the join/grouping columns form a hierarchy as can be derived from the functional dependencies

$$\mathit{City} \leftarrow C\# \leftarrow O\#$$

This hierarchy of the relations is illustrated in Figure 6. Indirect partitioning works particularly well for such hierarchical structures because, conceptually, the *cross-relation*

(a) *Customer* or *Order* as Probe Input of the Team

(b) *Order* or *Lineitem* as Probe Input

Figure 5: Alternative Query Evaluation Teams For The Three-Way Join

partitions (denoted as *Partition 1*, *Partition 2*, and *Partition 3*, and indicated by the shading) do not overlap. That is, as part of the partitioning, all matching tuples of all relations could be placed into a single cross-relation partition, and we are able to "team up" the two joins and the group-by operators. This way, we save the cost of two re-partitioning steps that would be carried out in a conventional hash-join/hash-aggregation plan (one for the second join and one for the aggregation). Of course, in practice, the partitions do overlap due to false drops resulting in extra cost, but this extra cost is usually much smaller than the cost of the additional partitioning steps carried out by a conventional plan. We should stress that the generalized hash team technique does not require disjoint cross-relation partitions for correctness—it has only performance relevance. Therefore, it could be applied to non-hierarchical cross-relation partitions. However, the performance gain will decrease as more tuples need to be inserted into multiple partitions.

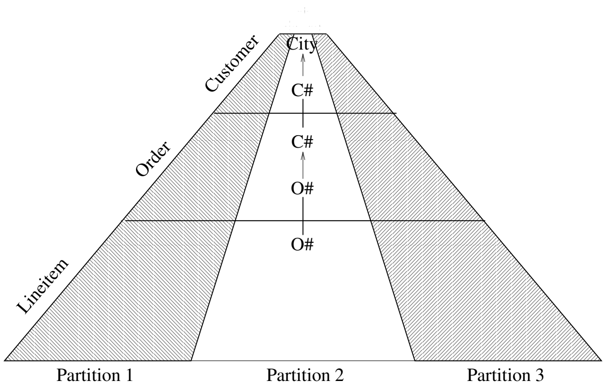


Figure 6: Indirectly Partitioning a Hierarchical Structure

5 False Drop Analysis

In this section, we will devise formulae in order to estimate the number of false drops that occur when executing generalized hash teams. These formulae can be used during query optimization in order to decide whether generalized hash teams are beneficial to execute parts of a query or whether traditional join techniques or partition nested-loop joins are more favorable. In addition to these formulae, the optimizer must be extended by formulae that estimate the overall cost of generalized hash teams (based on our false drop analysis) and by enumeration rules that generate plans with generalized hash teams. These extensions, however, are straightforward and/or are virtually the same as the extensions made in Microsoft's latest SQL Server product to integrate ordinary hash teams [GBC98].

5.1 Binary Joins and Aggregation

We begin and estimate the number of false drops for binary joins such as the Customer-Order query of Section 2. (We will consider multi-way joins in the next subsection.) To re-iterate, Figure 7 shows how false drops occur. The figure shows that the ▲ *Customer* and the ■ *Customer* are assigned to different partitions but have the same hash value for setting the bitmaps. As a result, all *Orders* that refer to the ▲ *Customer* will produce one false drop because they will be (accidently) copied into the second partition. Likewise, all the *Orders* that refer to the ■ *Customer* will produce a false drop because they will accidently be copied into the first partition. If there were another *Customer* with the same hash value, i, and stored in the third partition (not shown), then all the *Orders* referring to the ▲, ■, or this third *Customer* would produce two false drops. *Orders* which refer to the ● and □ *Customers*, on the other hand, do not produce any false drops: these two *Customers* have the same hash value, but they are stored in the same *Customer* partition.

Statistically, the number of false drops can be estimated fairly easily; similar formulae have, e.g., been devised in [Car75, HM97].

To simplify the discussion, we will assume that the join is a functional join and that there is a referential integrity constraint so that every *Order* refers to exactly one *Customer* in the join. (These assumptions can easily be relaxed for cases in which there is e. g. a predicate that restricts the *Customers* participating in the join.) Furthermore, we will use n to denote the number of partitions, b for the length of every bitmap, c for the number of *Customers*, and o for the number of *Orders*. Under these assumptions, an *Order* must be placed into at least one partition, and it is falsely

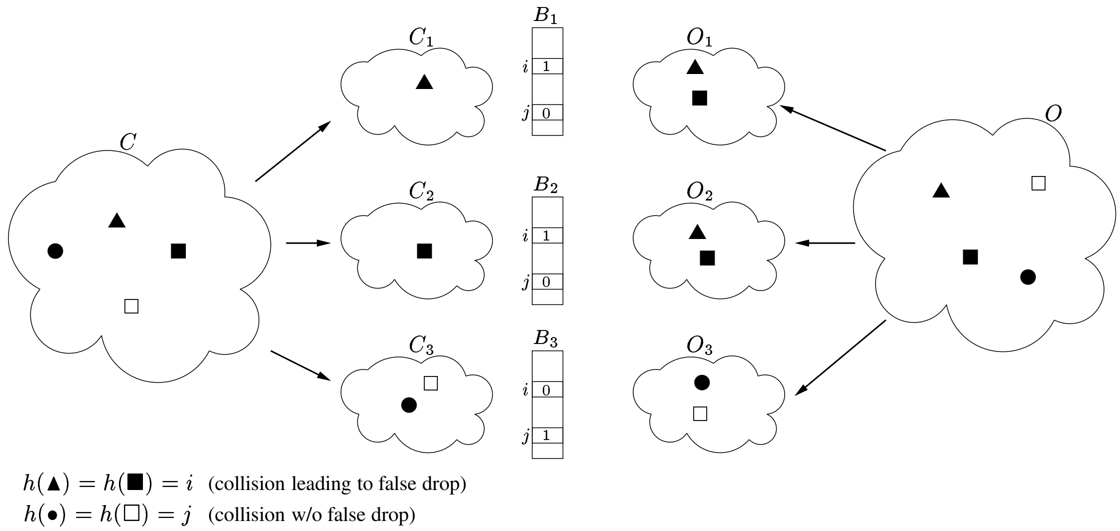


Figure 7: False Drops in Binary Joins

copied into one of the other $n-1$ partitions, if one of the other $c-1$ *Customers* to which the *Order* does not refer has set the corresponding bit in the bitmap of that partition. Putting it differently, the probability of a false drop for an *Order* in a partition is:

$$1 - \left(1 - \frac{1}{n*b}\right)^{c-1}$$

(Here, $\frac{1}{n*b}$ is the probability that a *Customer* sets the relevant bit; $1 - \frac{1}{n*b}$ is the probability that a *Customer* does not set the relevant bit; $(1 - \frac{1}{n*b})^{c-1}$ is the probability that none of the $c-1$ *Customers* sets the relevant bit; and finally, $1 - (1 - \frac{1}{n*b})^{c-1}$ is the probability that at least one of the $c-1$ *Customers* sets the relevant bit.)

In all, the number of false drops for all *Orders* considering all of the $n-1$ "critical" partitions can be estimated as follows:

$$o * (n-1) * \left(1 - \left(1 - \frac{1}{n*b}\right)^{c-1}\right) \qquad (1)$$

It should be noted that this formula (and the actual number of false drops) is independent of skew between *Orders* and *Customers*; that is, if some *Customers* generate more *Orders* than others, this fact will (statistically) not affect the number of false drops. This formula does assume that the hash function used to hash the *C#* spreads evenly; if not, the number of false drops will obviously be higher. The formula also assumes that the *Customers* are partitioned evenly into partitions (this is the partitioning function applied to the *City* attribute). If the partitioning function is skewed, the number of false drops *decreases*. To see why, consider again Figure 7 in which *Orders* referring to the $\square$ and $\bullet$ Customers do not produce false drops because these two *Customers* are placed into the same partition. In the extreme case in which all *Customers* are placed into the same *Customer* partition, no false drops at all occur. (This extreme case, however, is obviously not desirable for other reasons.)

Unfortunately, Formula (1) cannot be used in a practical query optimizer. If c and b are large, which they usually are, computing the result of this formula with reasonable accuracy is prohibitively expensive. Also, computing the (standard) approximation using $e^{\frac{x}{y}}$ for $(1 - \frac{1}{y})^x$ is prohibitively expensive. We, therefore, propose to use the following very simple approximation in order to estimate the number of false drops in a query optimizer:

$$o * (n-1) * \frac{c-1}{n*b} \qquad (2)$$

This formula simply estimates the probability that the relevant bit in a "critical" partition is set by one of the other $c-1$ *Customers* as $\frac{c-1}{n*b}$. The simplification consists in assuming that no two customers set the same bit in a bitmap. This formula is conservative: it can be shown that $\frac{c-1}{n*b} > 1 - (1 - \frac{1}{n*b})^{c-1}$. Thus, a query optimizer using this formula will overestimate the number of false drops and will use generalized hash teams cautiously. For Query 1 of Section 2, we measured how accurate this approximate formula is depending on the amount of memory available to execute a query, and we show the results in Figure 8. (The amount of available main memory determines n and b; we present details of our experimental environment in Section 6, the database cardinalities are summarized in Table 1, Page 10.) We see that the estimates of the approximate formula are quite precise compared to the actual number of false drops measured while executing the query. Only for small memory sizes (and correspondingly short bitmaps, i.e., small b), the approximate formula visibly overestimates the number of false drops.

5.2 Multi-way Joins

We now turn to multi-way join queries. Estimating the number of false drops is complicated in this case, and we have not yet found a statistically precise formula. Even if we did, such a formula would, again, probably be prohibitively expensive to compute. We, therefore, concen-

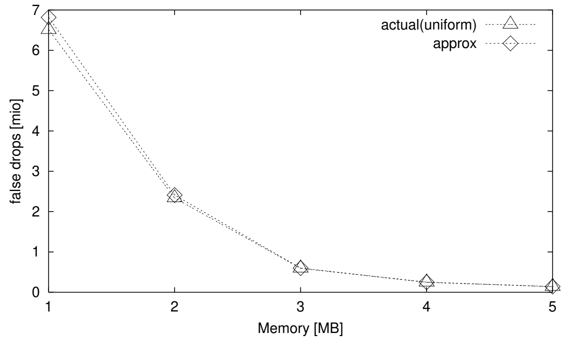


Figure 8: False Drops for Query 1

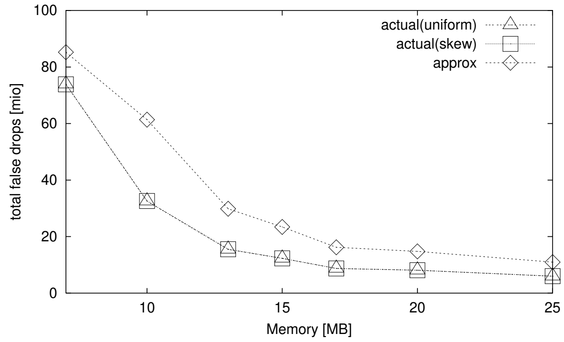


Figure 9: False Drops for Query 2

trate on a very simple approximate formula that can be implemented and evaluated by a query optimizer with very little effort. We will, furthermore, concentrate on Query 2 of Section 4, as an example, and note that our results can easily be generalized to other queries.

First of all we note that there are *Order* and *Lineitem* false drops when using generalized hash teams for our three-way join example query. The *Order* false drops can be computed using exactly the same (approximate or exact) formulae described in the previous subsection. Second, we note that the *Lineitem* false drops can occur in one of two ways:

1. *Orders* placed into different *Order* partitions can have the same *O#* hash value; all *Lineitems* referring to such *Orders* produce false drops. This is the same phenomenon as depicted in Figure 7, just transposed to the *Order-Lineitem* join.

2. False Drop Propagation: If an *Order* produces a false drop, all the *Lineitems* that refer to that *Order* produce a false drop as well. Consider, as an example, again Figure 7. All *Lineitems* that refer to an *Order* which in turn refers to the ▲ *Customer* are (falsely) copied into the second *Lineitem* partition.

The first kind of false drop can be approximated using Formula (2). Using l as the number of *Lineitems* and b_o as the length of the *Order* bitmaps, we get:

$$l * (n-1) * \frac{o-1}{n * b_o} \tag{3}$$

The second kind of false drop can be estimated as

$$f_o * \frac{l}{o} \tag{4}$$

where f_o is the estimated number of *Order* false drops, estimated using again Formula (2). In all, we approximate the number of *Lineitem* false drops as the sum of the number of these two kinds of false drops. This is again a conservative approach that overestimates the number of false drops and makes the optimizer be overly cautious to use generalized hash teams because this approach assumes that there is no *overlap* between the two kinds of false drops. (Modeling the number of false drops precisely, this overlap would have to be subtracted from the estimated total number of false drops.)

Again, we would like to note that the number of false drops is (statistically) independent of skew. The number of false drops is also independent of the join order within the generalized hash team. As stated in Section 4, joins can freely be ordered within a team, but the partitioning order is fixed and false drops only occur in the partitioning phase. The number of false drops, however, does depend on the quality of the hash function used to set the bitmaps and on the partitioning function used to partition the first relation (i.e., *Customer* in our examples).

To measure the accuracy of our approximate formula for multi-way joins, we ran Query 2 from Section 4 and compared the actual number of false drops with the estimated number of false drops. For these experiments, we used two different database instances: (a) *uniform* with *Orders* referring to *Customers* using a uniform distribution, and (b) *skewed* with *Orders* referring to *Customers* according to a 80-20 self-similar distribution as defined in [GSE$^+$94]. Figure 9 shows the total number of false drops for each case. We see that our approximations overestimates the number of false drops significantly in some cases; however, for the purpose of query optimization the approximations seems to be accurate enough. Furthermore, we see that the actual number of false drops is independent of skew, as expected.

6 Experimental Results

In this section, we will present experimental results conducted using an experimental implementation of generalized hash teams, partition nested loop joins, and traditional (hash-based) ways to carry out joins and aggregation. We will present the running times of our examples, Query 1 and 2, using a synthetic *Customer-Order-Lineitem* database.

6.1 Experimental Environment

We integrated our implementation of generalized hash teams and partition nested-loops into an experimental query engine that is based on the iterator model [Gra93]. That query engine also provides iterators for traditional (hash-based) joins and aggregation. All code is written in

Table	*Tuple Width*	*Cardinality*	*Size in MB*
Customer	88 bytes	750,000	66 MB
Order	112 bytes	7,500,000	840 MB
Lineitem	72 bytes	30,000,000	2,160 MB

Table 1: Database Characteristics

C++. We installed the query engine on a Sun UltraSPARC station with a 167 MHz processor, 512 MB of main memory, and running Solaris 2.6. In all experiments, we varied the amount of main memory available for query processing. We used relatively small memory sizes in order to simulate a multi-user environment in which many queries run concurrently an only little main memory is available for each query.We made use of Solaris' *direct IO* feature in order to avoid caching at the operating system level. The database was stored on a 9 GB Seagate Barracuda disk drive and another 9 GB Barracuda disk drive was used to store intermediate query results.

Our test database is characterized in Table 1. It consists of a *Customer, Order*, and *Lineitem* table with the usual TPC-D-style schema [TPC95]. The cardinalities of the tables are set according to the TPC-D specifications at a scaling factor of five. In some experiments, however, we varied the cardinality of the *Order* table in order to demonstrate the scalability of the approaches along that dimension. We generated random tuples using a uniform distribution wherever appropriate. That is, the *Customer.City* fields are uniformly distributed among 75,000 cities, the *Orders* referred uniformly to *Customers*, and *Lineitems* referred uniformly to *Orders*. As stated in Section 5, we also experimented with skewed databases, but we will not show the results here because they were identical with the results obtained using such a uniform database. The *C#* and *O#* fields, the keys of the tables, are also generated randomly, rather than just sequentially. A database with sequential *C#* and *O#*[2] would have made the use of generalized hash teams even more attractive because absolutely no false drops would occur in such a database if $b \geq c$ and $b_o \geq o$.

As benchmark queries, we used Query 1 and 2 from Sections 2 and 4. These are just two example queries for which generalized hash teams and, to some extent, partition nested loops are useful. Of course, many other examples can be found and it is just as easy to find example queries for which our new approaches are not useful (e.g., grouping by an attribute of the *Order* table). In all cases, we used the best possible plans (including the join order) and the best possible main memory allocation for each group of operators that run concurrently in a plan. Every operator (e.g., scan or partitioning) that reads and writes data to disk gets memory so that blocks of at least 64K are read and written to disk in order to avoid excessive random IO. For generalized hash teams, this minimum allocation was given to the partitioning operators and the rest of the available memory was used for the bitmaps in the partitioning phase. For the traditional plans (in particular if early aggregation was used), memory allocation was somewhat trickier because it is difficult to decide how much memory to allocate to the partitioning phase of the group-by which runs concurrently with the last join—again, we experimented with different configurations and report the best results.

[2]This is not unrealistic in practice because keys are typically generated sequentially by the application or database system.

6.2 Running Time of Query 1

Figure 10 shows the running time of Query 1 using generalized hash teams, partition nested loop joins, and two traditional plans that use an ordinary hash join and hash aggregation to execute the query. The difference between the two traditional plans is that early aggregation (as described in [Lar97]) is effected in one of the two plans. Early aggregation reduces the size of the intermediate results that must be written to disk in the partitioning phase of the group-by operator. We observe that, as expected, generalized hash teams and partition nested loop joins significantly outperform the traditional plans in the whole range of main memory sizes. The traditional plans perform particularly poorly if there is only little memory available—in this case, the IO costs of the join and group-by operators are very high because many small partitions must be created and thus, the benefits of saving the partitioning step for the group-by operation is high. Note that for small memory size, the number of false drops is also particularly high for generalized hash teams (Figure 8), but the extra cost due to false drops is much lower than the cost of an extra partitioning step. Increasing the size of the memory, the advantages of our new approaches get smaller, but only for very large memory sizes, when the join and/or group-by can completely be carried out in memory, the traditional plans would perform as well as our new approaches. For Query 1, generalized hash teams and partition nested loop joins have almost the same performance; in this case, processing false drops is as expensive as re-reading the reduced *Order* table for all memory sizes.

6.3 Running Time of Query 2

Figure 11 shows the running time of Query 2 for various different plans. Again, generalized hash teams are the overall winner. In this case, however, generalized hash teams are only beneficial if a certain amount of memory is available. (Recall from Section 4 that the memory requirements increase with the number of operations that participate in the team.) For the traditional plans, the best memory configuration involves carrying out the whole group-by in memory so that early aggregation does not improve the running time in these experiments. The traditional plans loose here because they require re-partitioning for the second join (i.e., the join with *Lineitem*). For Query 2, we studied two plans that make use of partition nested loops: the *PNL O, L* plan which carries out partition nested loops for both joins, and the *PNL L* plan which carries out a (traditional) hash join for *Customer* $\bowtie$ *Order* and partition nested loops for the join with *Lineitem*. Both plans show poor performance if there is only little memory available, for re-reading the reduced *Lineitem* and *Order* (for PNL O, L) tables several

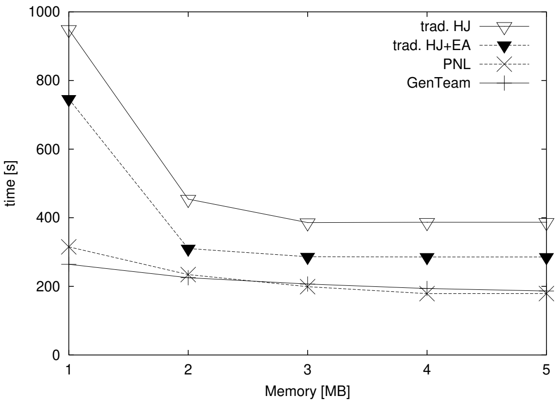


Figure 10: Response Time Query 1 [secs]

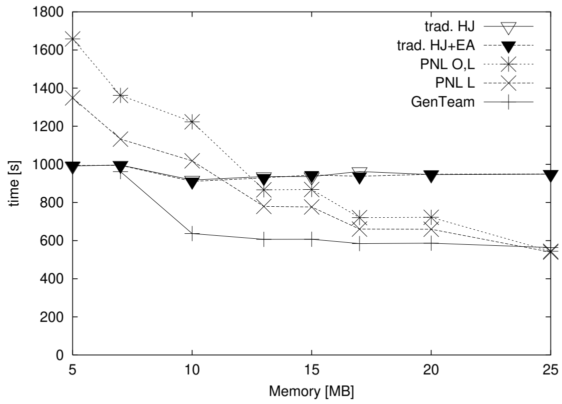


Figure 11: Response Time Query 2 [secs]

times, but both plans become better the more memory is available. All other plans, in contrast, are fairly flat. Beyond 25 MB of memory (not shown), the partition nested loop join plans flatten out as well and show the same performance as generalized hash teams.

7 Related Work

The use of bitmaps is becoming increasingly popular to support decision support queries. In the database context, bitmaps have been used to speed-up the execution of joins in distributed [Bab79, VG84] as well as centralized systems [Bra84]. In these proposals so-called Bloom-filters [Blo70] are used to filter out tuples without join partners. [HM97] use bitmap signatures for processing joins involving predicates on nested sets. Also, bitmap indexing is a well-known concept; see, e.g., the early work on signature files [CS89] or the bitmap indices in Model 204 [O'N87]. Indexing attribute values via bitmaps [OQ97, CI98, WB98] and bitmap join indices [GO95] have recently received renewed attention in the context of query processing for data warehouses. To the best of our knowledge, however, so far nobody has used bitmaps for indirectly partitioning arguments of hash joins (or grouping operators).

The most relevant related work are hash teams, which were proposed in [GBC98]. As stated in the introduction, our work extends hash teams so that they become applicable in situations in which the columns of the join and group-by operations are not the same. Sort/merge-joins and sort-based aggregations can also be used to execute join/group-by queries. Like regular hash teams, such sort-based query techniques are only attractive if the columns of at least some of the join and group-by operations are the same. Generalized hash teams, on the other hand, are applicable and attractive for a much wider spectrum of queries.

Furthermore, there have been a couple of proposals to integrate join and group-by processing and for special multi-way join operators [Ull89, RRS91, CM95]. In [CKK98] we devised a technique for combining sorting and hash join processing. The techniques proposed in that work, however, differ significantly from the approach proposed in this paper, and they would not perform as well as generalized hash teams in the situations described in this paper.

Another line of related work are approaches to optimize queries with group-by operations [YL94, CS94]. Our example *Customer-Order* query, for instance, could be implemented by grouping the *Order* table by *C#* before the join and then grouping the result of the join by *City*. In this particular example, such a strategy would not be beneficial because it would involve the execution of an additional expensive group-by operation without reducing the cost of the other operations substantially. In general, however, generalized hash teams and early group-by processing can be used independently, and they can, in particular, both be used together to speed-up certain decision support queries.

8 Conclusion

Graefe et. al. [GBC98] developed a new hash-based processing technique called *hash teams* which was integrated into Microsoft's SQL Server product. This technique allows to "team up" several join (and grouping operators) that are based on the *same* column. This way intermediate re-partitioning is avoided in quite the same way that re-sorting intermediate join results is avoided in sort/merge-joins.

[GBC98]'s technique requires that all operators of a team are based on the same column. In this work, we proposed generalized hash teams which allow to "team up" join and grouping operators even if they are based on different columns. This, of course, makes generalized hash teams applicable for a much larger class of queries. The key idea is indirect partitioning: A relation is partitioned on an attribute that is used in a later operation and bitmaps are constructed in order to guide the partitioning of other relations which are involved in the next operation. This technique can (in theory) be applied to an arbitrary number of relations and join and group-by operations; in practice, however, the number of operations that participate in a team is limited by the available memory needed to execute the team (as in traditional hash teams) and to construct the bitmaps.

In this paper, we presented details of such *generalized* *hash teams* and carried out experiments demonstrating the usefulness of the approach for certain classes of decision support queries. We also presented formulae that can be used by a query optimizer in order to cost out plans with generalized hash teams and thus decide when they are beneficial. Furthermore, we described and evaluated another new algorithm which we called *partition nested loops* and that can, in some sense, be seen as a simplified variant of generalized hash teams. In our experiments, however, we could not find cases in which partition nested loops outperform generalized hash teams, but we did find cases in which partition nested loops perform much worse.

Acknowledgements

We would like to thank Christoph Pesch for his help on the estimation of the false drops.

References

[Bab79] E. Babb. Implementing a relational database by means of specialized hardware. *ACM Trans. on Database Systems*, 4(1):1–29, March 1979.

[Blo70] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.

[Bra84] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 323–333, Singapore, Singapore, 1984.

[Car75] A. F. Cardenas. Analysis and performance of inverted data base structures. *Communications of the ACM*, 18(5):253–263, May 1975.

[CI98] C.-Y. Chan and Y. Ioannidis. Bitmap index design and evaluation. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 355–366, Seattle, WA, USA, June 1998.

[CKK98] J. Claussen, A. Kemper, and D. Kossmann. Order-preserving hash joins: Sorting (almost) for free. Technical Report MIP-9810, Universität Passau, 94030 Passau, Germany, August 1998.

[CM95] S. Cluet and G. Moerkotte. Efficient evaluation of aggregates on bulk types. In *Proc. Int. Workshop on Database Programming Languages*, Gubbio, Italy, Sept 1995.

[CS89] W. W. Chang and H.-J. Schek. A signature access method for the starburst database system. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 145–153, Amsterdam, Netherlands, 1989.

[CS94] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 354–366, Santiago, Chile, September 1994.

[GBC98] G. Graefe, R. Bunker, and S. Cooper. Hash joins and hash teams in Microsoft SQL Server. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 86–97, New York, USA, August 1998.

[GO95] G. Graefe and P. O'Neil. Multi-table joins through bitmapped join indices. *ACM SIGMOD Record*, 24(3):8–11, Oct 1995.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[Gra94] G. Graefe. Sort-Merge-Join: An idea whose time has(h) passed? In *Proc. IEEE Conf. on Data Engineering*, pages 406–417, Houston, TX, USA, 1994.

[GSE+94] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. Weinberger. Quickly generating billion-record synthetic databases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 243–252, Minneapolis, MI, USA, May 1994.

[HCLS97] L. Haas, M. Carey, M. Livny, and A. Shukla. Seeking the truth about *ad hoc* join costs. *The VLDB Journal*, 6(3):241–256, 1997.

[HM97] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with subset join predicates. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 386–395, Athens, Greece, August 1997.

[HR96] E. Harris and K. Ramamohanarao. Join algorithm costs revisited. *The VLDB Journal*, 5(1):64–84, 1996.

[HWM98] S. Helmer, T. Westmann, and G. Moerkotte. Diag-Join: An opportunistic join algorithm for 1:N relationships. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 98–109, New York, USA, August 1998.

[Lar97] P. A. Larson. Grouping and duplicate elimination: Benefits of early aggregation. Microsoft Technical Report, Jan 1997. `http://www.research.microsoft.com/~palarson/`.

[O'N87] P. O'Neil. Model 204 architecture and performance. In *Proc. of the 2nd Intl. Workshop on High Performance Transaction Systems*, Springer Verlag, Lecture Notes in Computer Science LNCS#359, pages 40–59, Asilomar, CA, USA, 1987.

[OQ97] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 38–49, Tucson, AZ, USA, May 1997.

[RRS91] A. Rosenthal, C. Rich, and M. Scholl. Reducing duplicate work in relational join(s): a modular approach using nested relations. Technical report, ETH Zürich, 1991.

[Sha86] L. Shapiro. Join processing in database systems with large main memories. *ACM Trans. on Database Systems*, 11(9):239–264, September 1986.

[TPC95] Transaction Processing Performance Council TPC. TPC benchmark D (decision support). Standard Specification 1.0, Transaction Processing Performance Council (TPC), May 1995. `http://www.tpc.org/`.

[Ull89] J. Ullman. *Principles of Database and Knowledge-Base Systems*, volume II. Computer Science Press, Woodland Hills, CA, 1989.

[VG84] P. Valduriez and G. Gardarin. Join and semijoin algorithms for a multiprocessor database machine. *ACM Trans. on Database Systems*, 9(1):133–161, March 1984.

[WB98] M.-C. Wu and A. Buchmann. Encoded bitmap indexing for data warehouses. In *Proc. IEEE Conf. on Data Engineering*, pages 220–230, Orlando, FL, USA, 1998.

[YL94] W. Yan and P. A. Larson. Performing group-by before join. In *Proc. IEEE Conf. on Data Engineering*, pages 89–100, Houston, TX, USA, 1994.