The VLDB Journal (2000) 9: 190–213

The VLDB Journal

Exploiting early sorting and early partitioning for decision support query processing

J. Claussen[*], **A. Kemper, D. Kossmann**[**], **C. Wiesner**

Universität Passau, Lehrstuhl für Informatik, 94030 Passau, Germany; E-mail: {claussen,kemper,kossmann,wiesner}@db.fmi.uni-passau.de

Edited by M.P. Atkinson. Received April 4, 2000 / Accepted June 23, 2000

Abstract. Decision support queries typically involve several joins, a grouping with aggregation, and/or sorting of the result tuples. We propose two new classes of query evaluation algorithms that can be used to speed up the execution of such queries. The algorithms are based on (1) *early sorting* and (2) *early partitioning* – or a combination of both. The idea is to push the sorting and/or the partitioning to the leaves, i.e., the base relations, of the query evaluation plans (QEPs) and thereby avoid sorting or partitioning large intermediate results generated by the joins. Both early sorting and early partitioning are used in combination with hash-based algorithms for evaluating the join(s) and the grouping. To enable early sorting, the sort order generated at an early stage of the QEP is retained through an arbitrary number of so-called *order-preserving hash joins*. To make early partitioning applicable to a large class of decision support queries, we generalize the so-called hash teams proposed by Graefe et al. [GBC98]. Hash teams allow to perform several hash-based operations (join and grouping) on the same attribute in one pass without repartitioning intermediate results. Our generalization consists of indirectly partitioning the input data. Indirect partitioning means partitioning 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 QEPs based on *early sorting*, *early partitioning*, or both in combination perform significantly better than conventional strategies for many common classes of decision support queries.

Key words: Decision Support Systems – Query processing and optimization – Early sorting and partitioning – Hash joins and hash teams – Performance evaluation

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 a few minutes.

Decision support typically involves the execution of complex queries with join, group-by, and sort 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 [CI98], special joins that exploit bitmap join indices [GO95], new join methods [HWM98], or multi-query optimization for decision support [ZDNS98], 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-H and TPC-R benchmarks [TPC99] – derived from the TPC-D benchmark – have been proposed in order to evaluate the performance of a database product for decision support queries.

We propose two new classes of query evaluation algorithms that can be used to speed up the execution of decision support queries. The algorithms are based on (a) *early sorting* and (b) *early partitioning* – or a combination of both. The idea is to push the sorting and/or the partitioning to the leaves, i.e., the base relations, of the query evaluation plans and thereby avoid sorting or partitioning large intermediate results generated by the joins. Both, early sorting and early partitioning are used in combination with hash-based algorithms for evaluating the join(s).

The idea of early sorting is not completely new. There has been work on query optimization to generate plans in which *sort* operators are carried out before joins (e.g., [SAC+79, SSM96]). Given the current set of available join methods, however, today's optimizers often face a dilemma: On the one hand, *sort* operators should sometimes be placed early in a plan so that they are cheap, because they are applied to small intermediate results. On the other hand, such early sorting limits the options for join processing, resulting in very high costs for join processing. To solve this dilemma, we propose a new

Some excerpts of this work – limited to early partitioning – appeared in the conference publication [KKW99].

This work was partially supported by the German National Research Council under contract DFG Ke 401/7-1.

[*] Current address: SAP AG, Advanced Technology Group, 69190 Walldorf, Germany; e-mail: jens.claussen@sap.com

[**] Current address: Institut für Informatik, Technische Universität München, 81667 München, Germany; e-mail: kossmann@in.tum.de

approach which makes it possible to do early sorting and have cheap joins at the same time. Our approach is based on a new technique that we call *order-preserving hash joins* (OHJs), which can be used instead of nested-loop joins in order to preserve the order generated by early sorting.

In the case of early partitioning, we generalize the so-called hash teams proposed by Graefe et al. [GBC98]. Hash teams allow several hash-based operations (join and grouping) to be performed in one pass without repartitioning intermediate results. Our generalization, which makes hash teams applicable to a much larger class of decision support queries, consists of indirectly partitioning the input data. Indirect partitioning is the partitioning of 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. These bitmaps are used to partition the other argument relation of this hash-based operation.

Our performance experiments show that such query evaluation plans based on early sorting, early partitioning, or a combination of the two perform significantly better than conventional strategies for many common classes of decision support queries.

1.1 Related Work

There has been a great deal of work on join techniques, sorting, grouping and query processing in general. A good overview of all join techniques used in practice today is given in [ME92], and [Gra94, GBC98] describe details and tuning techniques for hash joins which are relevant and useful for our approach, too. [Gra93] describes the "textbook" architecture for query processing (i.e., the iterator model). We integrated all our techniques into an existing query processor that is based on that architecture as part of our experimental work; indeed, our techniques could be integrated with very little effort into any other query processor that is based on that architecture. Designers of query optimizers have also paid attention to *interesting orders* since the 1970s; see, e.g., [SAC$^+$79] or [SSM96] for more recent work specifically addressing early sorting. Our work builds on that work, and its purpose is to provide the optimizer with new options to construct plans that exploit early sorting. Related query optimization work also includes work on "group-bys before joins" [YL94, CS94]; our study complements that work as we propose ways to further improve the performance of the "eager" group-by plans proposed there. In our own previous work, we proposed the $P(PM)^*M$ algorithm [BCK98], which is based on an idea similar to that of the OHJs presented here. The $P(PM)^*M$ algorithm, however, was specifically devised for so-called pointer-based joins with nested sets in object-oriented and object-relational database systems. In contrast, OHJs work for any kind of equi-join; they are order-preserving (not just "nested-set" preserving) and applicable in pure relational as well as object-oriented and object-relational database systems.

Li and Ross [LR99] describe two new join algorithms that are based on join indices. One of the algorithms is sort-based, the other is partition-based. Both algorithms draw profit from not completely materializing the join result, rather they store the temporary result on disk in two ordered files, which need to be merged to obtain the join result. The proposed technique avoids random disk I/O by sequentially scanning the input relations, the join index, and the temporary files. In contrast to our work their approach is based on precomputed join indices.

Graefe et al. [GBC98] proposed hash teams. They can combine multiple hash operations (join, aggregation) on the same attribute to a team and save disk accesses by avoiding partitioning of intermediate results. The drawback of their approach is that real-world applications often do not perform joins (and aggregations) on the same attribute – as required when applying hash teams. In [KKW99] we generalized this concept to allow hash teams to be applied to different attributes. These generalized hash teams are most useful for joining hierarchical structures, i.e., when the join attributes form a chain of functional dependencies. In this work we show that generalized hash teams work in a wider context and can be combined with other novel techniques such as OHJs.

In [MR94] the TID join technique was introduced, which allows attributes that are not essential for processing the join to be projected out. We found this idea very useful for bypassing bulky attributes around joins in order to utilize the main memory more efficiently both for the OHJ algorithm and the generalized hash teams. We describe in detail how this works in this paper.

1.2 Organization of the Paper

In Sect. 2 we introduce the order-preserving hash joins used to enable early sorting. The generalized hash teams to realize early partitioning are introduced in Sect. 3. We also demonstrate how to combine the two techniques for early sorting and early partitioning. Section 4 shows how to reduce the I/O volume of both query evaluation techniques by bypassing bulk data around the joins. The necessary extensions of a state-of-the-art dynamic programming-based optimizer are discussed in Sect. 5. Section 6 describes the experimental results we obtained from our implementation of the algorithms. Section 7 provides conclusions for this study.

2 Order-preserving hash joins

2.1 Motivation

Throughout the paper we will use a TPC-H/R style database, which is presented in Fig. 1. This sample database involves *Customer*, *Order*, and *Lineitem* tables with the usual information, where $C\#$ denotes the *Customerkey*, $O\#$ denotes the *Orderkey*, $L\#$ denotes the *Linenumber* within an order, $N\#$ denotes the *Nationkey*, and *MktSegment* denotes the *Marketsegment*. The keys of the tables are underlined. We assume, as in reality, that the *Customer* table contains significantly less tuples than the *Order* and *Lineitem* tables.

To demonstrate the mechanisms and the benefits of order-preserving hash joins (OHJs), we will use the two example queries Query 1 ($\mathcal{S}_{Mkt,\mathcal{N}\#,\mathcal{C}\#}(\mathcal{C} \bowtie \mathcal{O})$) and Query 2 ($\mathcal{S}_{Mkt,\mathcal{N}\#,\mathcal{C}\#}(\mathcal{C} \bowtie \mathcal{O} \bowtie \mathcal{L})$). The first query involves a join between the *Customer* and *Order* tables and requires the results to be produced in the following order: *Customer.Mktsegment*, *Customer.N#*, *Customer.C#*. The second query involves, in addition, a join with the *Lineitem* table. Both queries

Customer

C#	Name	N#	City	MktSegment

Order

O#	C#	Totalprice	Discount

Lineitem

O#	L#	Quantity	Extendedprice	Discount

Fig. 1. Relational schema of the sample database (Keys are underlined)

could, for example, be initiated by a middleware product in order to analyze the orders and lineitems of groups of customers from different market segments and countries. These queries could also occur as query blocks that produce the input of a *rollup* operator implemented as part of an extended relational database system.

Figure 2 shows three alternative "traditional" query evaluation plans for the first query. These three plans demonstrate the dilemma of today's query processors: the optimizer must choose between high sorting or high join costs. To see why, let us take a closer look at the costs of the three plans. The first plan is applicable if there is an index that can be used to read the *Customer* tuples in the right order. If this index is clustered with respect to the *Customer* table, the cost to bring the *Customer* tuples into the right order will be very low in this plan, but the cost to process the (index) nested-loop join, which is the only known order-preserving join method applicable in this case, will be very high because an (index) nested-loop join will cause excessive random disk I/O in this case. The second plan has a similar cost profile: the *sort* operator and, thus, bringing the *Customer* tuples into the right order is quite cheap because there are not many *Customer* tuples, while the nested-loop join has again a very high cost. In the third plan, the join is executed in the cheapest possible way (i.e., using hashing), but the *sort* at the top of that plan is expensive because the result of the join is very large – much larger than the *Customer* table. For the second query, today's optimizers face a similar dilemma: either cheap ordering with one or two expensive nested-loop joins or cheap hash joins and expensive sorting at the end.

The goal of the OHJ is to break this dilemma and allow the optimizer to generate plans like the ones shown in Figs. 3 and 4. The key idea is to split the different phases of external sorting and carry out join operations in between the *generate_runs* and *merge_runs* phases. The join operations are carried out by a special OHJ operator which essentially is a hash join augmented with a fine-grained partitioning and re-merging step. Thus, the OHJ exhibits the same high performance as standard hash joins. As shown in Fig. 4, it is possible to have any number of OHJ operators in a query, and as we will see in the next subsection, this aspect requires special attention in the implementation of OHJ operators. The plans with early sorting followed by OHJ joins to preserve the order are denoted SOHJ. Of course, we can also benefit from an existing order (via a clustered index) such that the sorting becomes obsolete. These plans will simply be called OHJ plans.

2.2 Binary order-preserving hash join plans

OHJs are based on Grace hash joins as described in [HCLS97].[1] That is, both input relations are partitioned using hashing in such a way that each partition of the inner (build) relation fits into the memory, and a pair of partitions are then joined by building an in-memory hash table for the partition of the build relation and probing every tuple of the corresponding partition of the outer (probe) relation using that hash table. The key idea of order-preserving hash joins lies in the following very simple observation: if the whole probe relation is ordered to begin with, then the result of the join of a pair of (probe and build) partitions is ordered too. Putting it differently, the results of joining pairs of partitions can be seen as *sorted runs* so that these runs only need to be *merged* to obtain an ordered join result. This process is visualized in Fig. 5, which demonstrates how the order of R, the probe relation, is preserved after the join with S, the build relation. In the figure, R and S are partitioned into two partitions (*ptn* denotes partitioning and *mrg* stands for the merge).

In general: Assume we have two relations R, with attributes A and B, and S, with attributes B and C. Let R be ordered by attribute $R.A$ and let B be the join attribute. (In practice, obtaining R in sorted order means scanning the relation via a [clustered] index on the order attribute.) To evaluate the join $R \bowtie_B S$ by a hash join, we first partition the probe input R and the build input S into $R_1, \ldots, R_k$ and $S_1, \ldots, S_k$, respectively, as in traditional Grace hash joins. In particular, we use the same hash function to partition both inputs' data and do not require special order-preserving hash functions. We then join the partitions pairwise (i.e., $R_i \bowtie S_i$ for $1 \leq i \leq k$), just as in traditional Grace hash joins. Then, we write the results of joining every pair of partitions to disk and merge those *runs*; this is the only special step for the simple OHJs. Here and throughout the paper, we will assume that S can be partitioned in one phase to generate memory-sized partitions. Our algorithms can, however, easily be adapted if multiple partitioning steps (e.g., due to skew or large relations and small main memory) or no partitioning at all is required. (In the latter case, the corresponding merge step is also omitted.)

2.3 Multi-way order-preserving hash join plans

Now, assume we want to compute the join

$$R \bowtie_{R.B=S.B} S \bowtie_{S.C=T.C} T$$

and preserve the order of R according to attribute $R.A$. This query corresponds to Query 2.

One way to achieve this is to first join S and T and then apply the binary OHJ on R and $(S \bowtie_C T)$, as described in the previous subsection. This way to order the joins might, however, not always be attractive and, therefore, we will show in this section how plans with two OHJs can be produced: one OHJ for $R \bowtie S$ and one OHJ for the join with T, as in the plan of Fig. 4. Here, we must be careful, however, because we cannot afford using two simple OHJs. Such a naive implementation would involve a fully fledged *merge* step as part of the R

[1] Hash joins without partitioning, i.e., the complete build input fits into the memory, are order-preserving with respect to the probe input, anyway.

select c.∗, o.Totalprice
from Customer c, Order o
where c.C# = o.C#
order by c.Mktsegment, c.N#, c.C#

Query 1: Binary join
$(\mathcal{S}_{Mkt,\mathcal{N}\#,\mathcal{C}\#}(\mathcal{C} \bowtie \mathcal{O}))$

select c.∗, l.Extendedprice
from Customer c, Order o, Lineitem l
where c.C# = o.C# **and** o.O# = l.O#
order by c.Mktsegment, c.N#, c.C#

Query 2: Multi-way join
$(\mathcal{S}_{Mkt,\mathcal{N}\#,\mathcal{C}\#}(\mathcal{C} \bowtie \mathcal{O} \bowtie \mathcal{L}))$

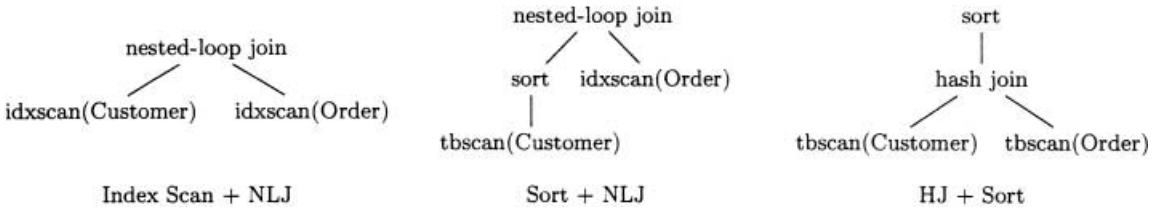


Fig. 2. Traditional plans for query 1

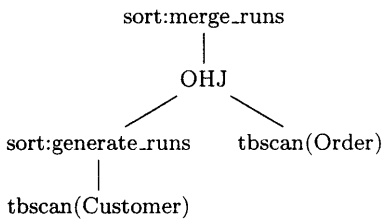


Fig. 3. SOHJ plan for query 1

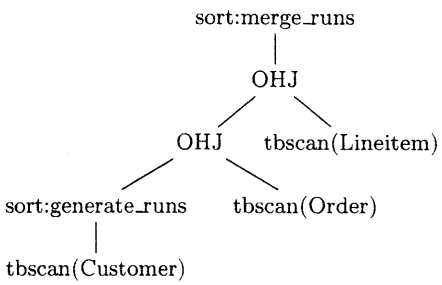


Fig. 4. SOHJ plan for query 2

ohj S, and this additional *merge* step would be too expensive because it would involve writing and re-reading the whole result of $R \bowtie S$ to/from disk. Instead, we directly partition the *runs* produced by the R *ohj* S and merge corresponding partitions before the join with T.

In more detail: If the third relation T is partitioned into $T_1, \ldots, T_l$ then the join partition $RS_i = R_i \bowtie S_i$ resulting from joining R_i with S_i is partitioned into $RS_{i1}, \ldots, RS_{il}$, which are all written to disk. Doing this for all intermediate result partitions $RS_1, \ldots, RS_k$ results in $k \cdot l$ partitions on disk. These $k \cdot l$ fine-grained partitions are then re-merged into the l partitions: for all $1 \leq j \leq l$, $RS_{1j}, \ldots, RS_{kj}$ are merged into a single partition $RS_{\{1,\ldots,k\}j}$, and this partition is then joined with T_j as part of the OHJ with table T. The whole process is shown in Fig. 6 for $k = 2$ and $l = 2$, and step by step the algorithm works as follows:

1. Scan S and partition S into k main memory-sized[2] partitions $S_1, \ldots, S_k$ using a hash function h_k on $S.B$.
2. Scan R via a (cluster) index on $R.A$ and partition R into k partitions $R_1, \ldots, R_k$ using the hash function h_k on $R.B$.
3. Scan T and partition T into l main memory-sized[3] partitions $T_1, \ldots, T_l$ using a hash function h_l on $T.C$.
4. For each $1 \leq i \leq k$ do:
   (a) Create l initially empty partitions $RS_{i1}, \ldots, RS_{il}$ on disk.
   (b) Load partition S_i into a main memory hash table.
   (c) For each tuple $r \in R_i$ probe the hash table to determine the join result tuple(s) *rs* and append *rs* to partition RS_{ij} with $j = h_l(rs.C)$.

   Having finished Step 4, there are $k \cdot l$ partitions $RS_{11}, \ldots, RS_{1l}, \ldots, RS_{kl}$ stored on disk.
5. For each $1 \leq i \leq l$ do:
   (a) Create an initially empty partition RST_i on disk.
   (b) Load partition T_i into a main memory hash table.
   (c) Merge the partitions $RS_{1i}, \ldots, RS_{ki}$ and for each tuple *rs* probe the hash table to determine the join result tuple(s) *rst* which are appended to partition RST_i.

   Having finished Step 5, there are l partitions $RST_1, \ldots, RST_l$ stored on disk.
6. Merge the partitions $RST_1, \ldots, RST_l$ to obtain the join result *RST* in the order of $R.A$.

This approach can be applied to join any number of tables: after every join, the result is directly partitioned for the next join, and the partitions are then re-merged in order to carry out the next join. Tracing the ordered relation (i.e., R in our example), the following pattern of operators is applied to that relation:

$$P\ J\ (P\ M\ J)^*\ M.$$

[2] More precisely, we need to partition S such that the individual S partitions fit in memory, as requested by the Grace hash join method, and at the same time, there is enough memory left to partition the results of $R \bowtie S$ – cf. Step 4.

[3] More precisely, we need to partition T such that the individual T partitions fit in memory, and at the same time, there is enough memory left to merge k RS partitions – cf. Step 5.

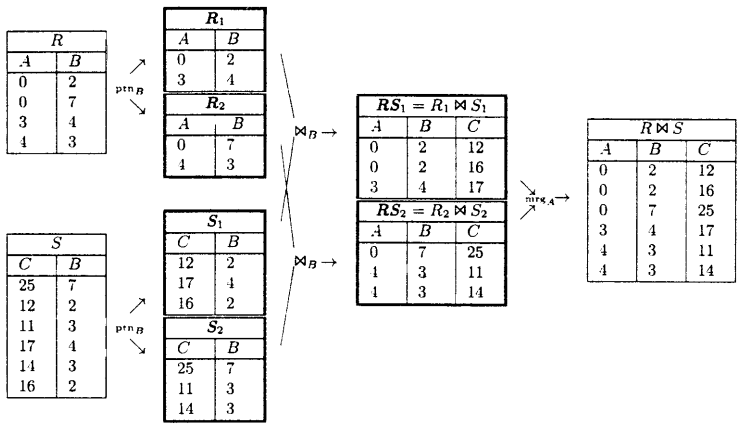


Fig. 5. Order-preserving binary hash join plan: build input S and probe input R, (Join and partitioning attribute B, sort attribute A)

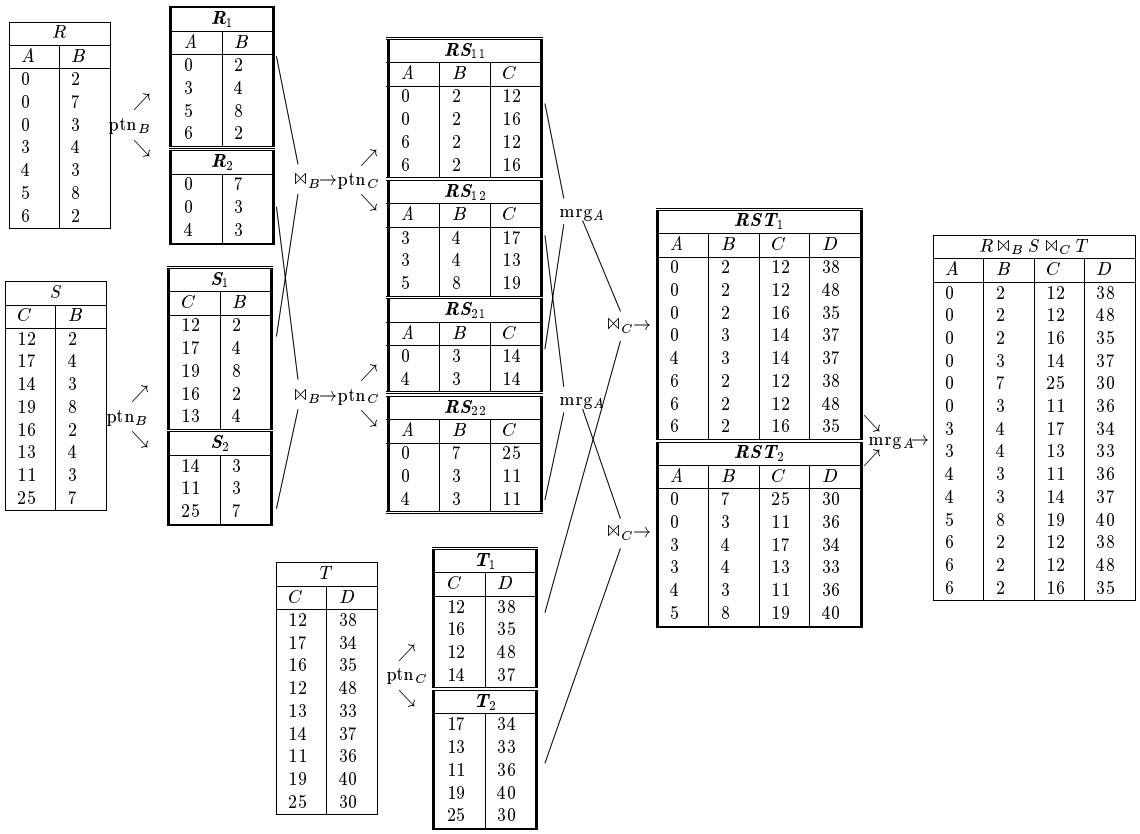


Fig. 6. Order-preserving three-way hash join plan: $(R \bowtie_B S) \bowtie_C T$ (Disk partitions are marked with thick rules)

Here, P denotes partitioning, M denotes merging, and J denotes the in-memory (hash) join phase.

2.4 Early sorting and order-preserving hash join plans

One might argue that our OHJ technique is only efficient if there is a clustered index on the sort attribute of R. Fortunately, however, we can generate the desired order on the fly during the initial partitioning step. This way we entirely avoid any additional I/O cost for sorting, and therefore, as we will show in Sect. 6, we get (almost) the same performance in the presence as in the absence of a clustered index; that is, we get sorting (almost) for free.

The trick is to combine the initial partitioning step of the OHJ plan with sorting runs. That is, we sort memory-sized runs of the probe input and partition each run individually. The partitions of every run are then re-merged during the processing of the first join. Step by step, the algorithm for the two-way join

$$R \bowtie_{R.B=S.B} S$$

works as follows:

1. Scan S and partition S into k main memory-sized[4] partitions $S_1, \ldots, S_k$ using a hash function h_k on $S.B$

[4] More precisely, we again need to reserve some space to merge partitions of R – cf. Step 3.

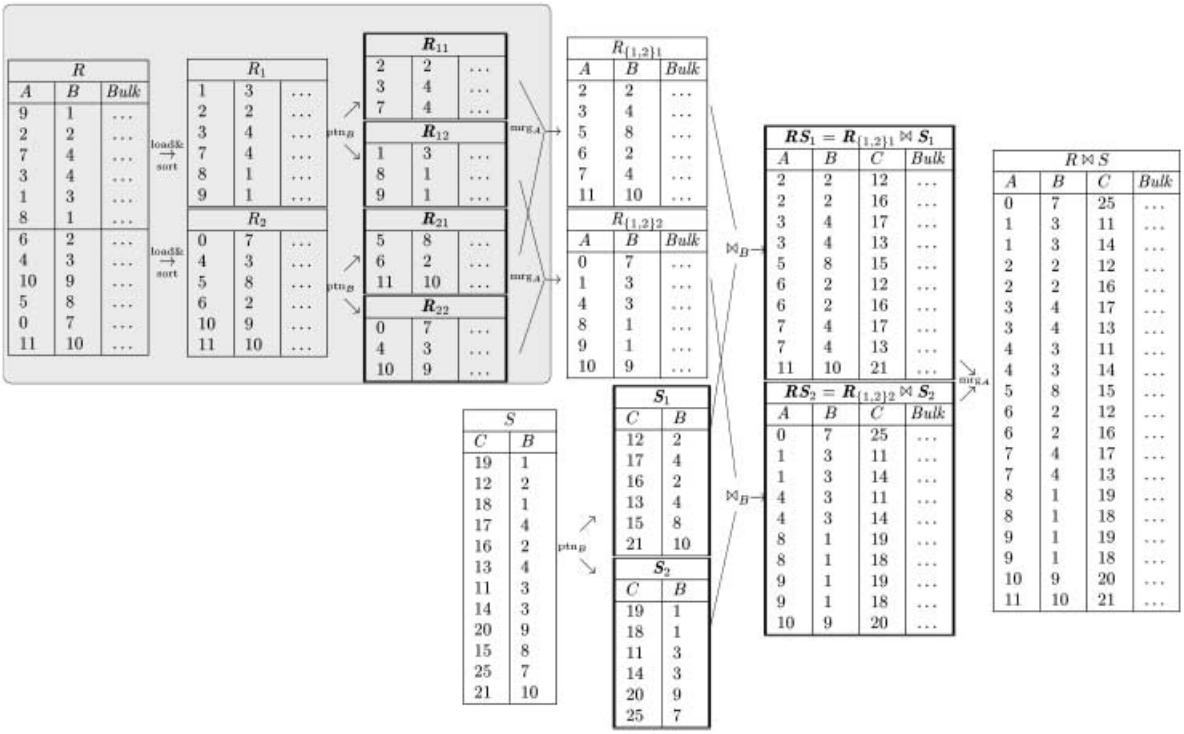


Fig. 7. Sorting on the fly: disk partitions are marked with thick rules; the sorting-on-the-fly step is highlighted by shading

2. Assume R is m times bigger than the available main memory. Then, for each $1 \leq i \leq m$ do:
   (a) Load the (next) memory sized chunk R_i into the memory and sort it according to attribute A.
   (b) Partition R_i into k partitions $R_{i1}, \ldots, R_{ik}$ by applying h_k on attribute B. Each partition constitutes a valid run according to attribute A. The partitioning can be done in a single linear iteration through the main memory resident run R_i – see below.
   (c) Write the partitions $R_{i1}, \ldots, R_{ik}$ sequentially to disk.
   Having finished this combined sort/partitioning step, $m \cdot k$ partitions $R_{11}, \ldots, R_{1k}, \ldots, R_{mk}$ – each constituting a valid sort run – are stored on disk.
3. For each $1 \leq i \leq k$ do:
   (a) Create an initially empty partition RS_i on disk.
   (b) Load partition S_i into a main memory hash table.
   (c) Merge the runs $R_{1i}, \ldots, R_{mi}$, and with each tuple r – in merge order – probe the hash table of S_i to determine the join result tuple(s) *rs* and append *rs* to partition RS_i.
   Having finished Step 3, there are k partitions $RS_1, \ldots, RS_k$ stored on disk.
4. Merge the partitions $RS_1, \ldots, RS_k$ to obtain the join result *RS* in the order of $R.A$.

This algorithm is illustrated for $m = 2$ and $k = 2$ in Fig. 7. The important part of the plan – i.e., the combined sorting and partitioning phase and the subsequent re-merging of the fine-grained partitioning – is shaded in the figure. Here the same principle is utilized as in combining multiple OHJ operators in one plan: the fine-grained partitions constitute ordered runs which are merged to an ordered partition for the next phase. The remainder of the evaluation plan is the same as for the basic OHJ plans. Of course, these so-called SOHJ

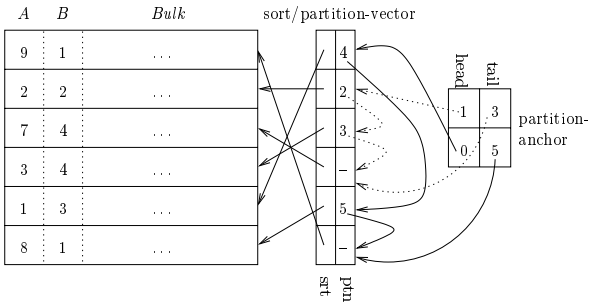


Fig. 8. Partitioning a sorted run

plans can, therefore, also be applied to multi-way join queries in the same way as described in the previous subsection. Tracing again the ordered relation (i.e., R), the following pattern of operators are applied (here, $S\&P$ denotes the combined sorting and partitioning step):

$$S\&P\ M\ J\ (P\ M\ J)^*\ M$$

Figure 8 illustrates the combined sorting/partitioning phase of the algorithm. A memory-sized chunk of the relation is loaded. Sorting is done via a vector that maintains pointers to the tuples being sorted; that is, only this vector is sorted, whereas the individual tuples need not be moved. Once the sorting is complete, we linearly scan this vector and determine the partition to which every tuple belongs. Hereby, we chain tuples that belong to the same partition together (i.e., we keep the index of the next tuple of the same partition in an additional field within the vector), and we keep a separate vector, called the *partition-anchors*, in order to keep the heads and the tails of every one of the k sorted "partition-lists" (in the example of Fig. 8, $k = 2$). Once this partitioning

is complete (i.e., the chaining is done and the heads and tails of the partition-anchors are set), the tuples can be written sequentially to disk: partition by partition following the heads of the partition-anchors one at a time and in the right sort-order. All partitions could, for example, be written into a single temporary file by inserting markers at partition boundaries, thereby avoiding overhead for allocating multiple temporary files. Note that Fig. 8 shows in fact the generation of the partitions R_{11} and R_{12} for run R_1 of Fig. 7.

With respect to run-time complexity, it would be cheaper to first partition each complete memory chunk and then sort the individual partitions: Assuming $m = |R|/M$ records fit into one memory chunk of size M, first sorting and then partitioning a memory chunk takes $m \cdot \log m + m$ abstract "operations". The reverse order, i.e., first partitioning a memory chunk and then sorting each of the N partitions requires only $m + N \cdot m/N \cdot \log(m/N) = m + m \cdot \log(m/N)$ operations. However, memory management for the partition/sort variant is more complex than for the sort/partition algorithm, because several sort vectors of unknown size have to be allocated. We have implemented both variants, and our performance experiments show that, in practice, the difference in run time is only marginal for the investigated configurations.

2.5 Early aggregation

So far, we have looked at SOHJ plans for simple **order by** queries. We will now see how SOHJ and OHJ plans can improve the performance of aggregate queries if sort-based aggregation with so-called *early aggregation*[5] is used [BD83, CS94,YL95,Lar97]. The idea of early aggregation is quite simple: As soon as a subgroup of tuples belonging to the same final group is identified, *collapse* them into a single tuple. Thus, the aggregation is folded such that it is already applied to the subgroups belonging to the same final group. During the final merge, the intermediate aggregation results are then combined. This is easily achieved for the aggregations *sum*, *min*, *max*, *count* which constitute commutative monoids [GKG+97] – i.e., operations that satisfy associativity and have an identity. For other aggregates more information has to be maintained to enable early aggregation. For example, in order to enable early *avg*-aggregation one has to store the *sum* and the *count* of each collapsed subgroup.

OHJ plans enable early aggregation very effectively if the *sort-ahead* is on the grouping attributes. Both variants (OHJ and SOHJ) produce sorted runs as a result of the join, and early aggregation can be implemented by merely collapsing all adjacent tuples with the same value of the grouping attributes into a single tuple *before* writing the join results to disk.

Let us again compare SOHJ plans with traditional hash join plans and look at the following query:

select R.A, **sum**(S.C)
from R **join** S **on** R.B = S.B
group by R.A
order by R.A.

Figure 9 shows an SOHJ plan and a traditional hash join plan for that query. The OHJ evaluation of this query – without the aggregation – was shown in Fig. 5 (page 194). Exploiting early aggregation, tuples with the same A-attribute value would be collapsed, i.e., the two tuples with $A = 0$ in RS_1 and the two tuples with $A = 4$ in RS_2 would be collapsed into one tuple.

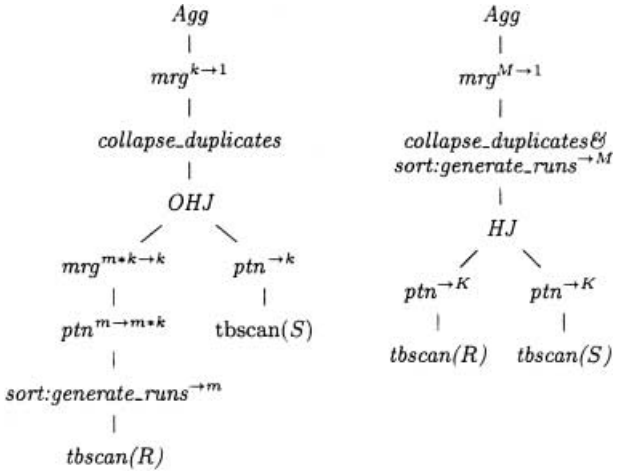


Fig. 9. Early aggregation ($M > k$)

Of course, the traditional hash join plan can benefit from early aggregation too. Here, early aggregation could be incorporated into the sorting of the runs: While loading the sort area, tuples belonging to the same group (called *duplicates* for brevity) are detected via a hash-vector and collapsed. Thereafter, the run is sorted and written to disk [Lar97].

The effectiveness of early aggregation depends on the number of collapsed duplicates. This number is in inverse proportion to the number of partitions (i.e., runs) being written after the join, because each of them is free of duplicates. In the notation of Fig. 9, the SOHJ plan writes k duplicate-free runs and the traditional hash join plan writes M runs. It turns out that in most cases $M > k$ holds, and therefore, the advantages of early aggregation are less pronounced in the traditional plan than in the SOHJ plan because less duplicates can be collapsed in the traditional plan. The number k is given as the number of partitions – denoted by $ptn^{\rightarrow k}$ in the query plan – that are needed to fit every individual partition of $\Pi_{B,C}(S)$ into a main-memory hash table. This number is usually quite small because of the projection on the relevant parts of S and because the hash join can make use of almost all the main memory. On the other hand, the number M is usually larger for two reasons:

- The sort area in which duplicates are collapsed cannot exploit all main memory because the collapse&sort operator runs in parallel with the hash join operator, which itself is memory intensive.
- The duplicates are detected after joining the tuples; therefore, these tuples are large.

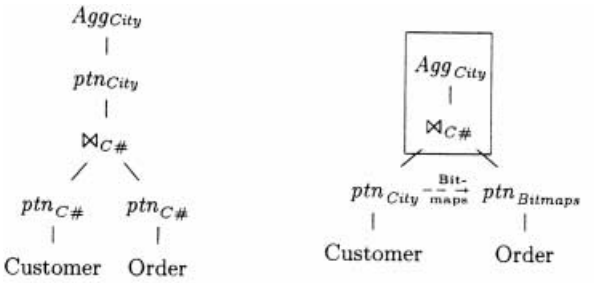
3 Early partitioning: generalized hash teams

To make early partitioning applicable for many important classes of decision support queries, we generalize the concept of hash teams proposed by Graefe et al. [GBC98]. The original hash teams are based on combining several hash-based

[5] Early aggregation must not be confused with early sorting or early partitioning, the techniques proposed in this paper.

select c.City, **sum**(o.Totalprice)
from Customer c, Order o
where c.C# = o.C#
group by c.City

Query 3: Join and aggregation
$(\mathcal{G}_{City}(\mathcal{C} \bowtie \mathcal{O}))$



a Traditional Hash Join and Hash Aggregation Plan **b** Generalized Hash Team

Fig. 10a,b. Execution plans for query 3

operations into a team. This allows all arguments of the team to be partitioned a priori without having to repartition intermediate results. In the original proposal, however, one can only form teams of hash-based operations that are all based on the same attribute.

3.1 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. As a running example, we will use Query 3, which asks for the total *Value* of all *Orders* grouped by the *Customer City*.

The traditional (state-of-the-art) plan to execute the example Query 3 is shown in Fig. 10a. 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 to group the results of the join by *City*. If there are more *Cities* than fit into the 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 I/O 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 a higher (CPU) cost than hashing; in any case, however, a traditional plan based on sorting would also involve I/O costs to write and read the *Customer* table, the *Order* table, and the result of the join.

Figure 10b shows a plan that makes use of generalized hash teams in order to execute our example query. Like the traditional plan shown in Fig. 10a, 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 each *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.

To make this clearer, let us look at Fig. 11, 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 (Passau) and M (Munich), the second partition contains all *Customers* located in B (Berlin) and HH (Hamburg), and the third partition contains all *Customers* located in NYC (New York) and LA (Los Angeles). Just as in a traditional (grace or hybrid) hash join, the goal is to generate partitions that fit into the 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 each of length ten. If a *Customer* tuple is inserted into a partition, the 1 + (*C#* `mod 10`)th bit of the corresponding bitmap is set. Thus, the fourth and sixth bits 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 bits 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 the 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 Fig. 11, for instance, *Order* 10 (highlighted in bold) is placed into the first and third *Order* partitions because this *Order* refers to *Customer* 3 and the fourth bit of 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[6], but false drops do impact the performance: the more false drops, the higher the I/O 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 Sect. 3.5. Furthermore, *Order* duplicates

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

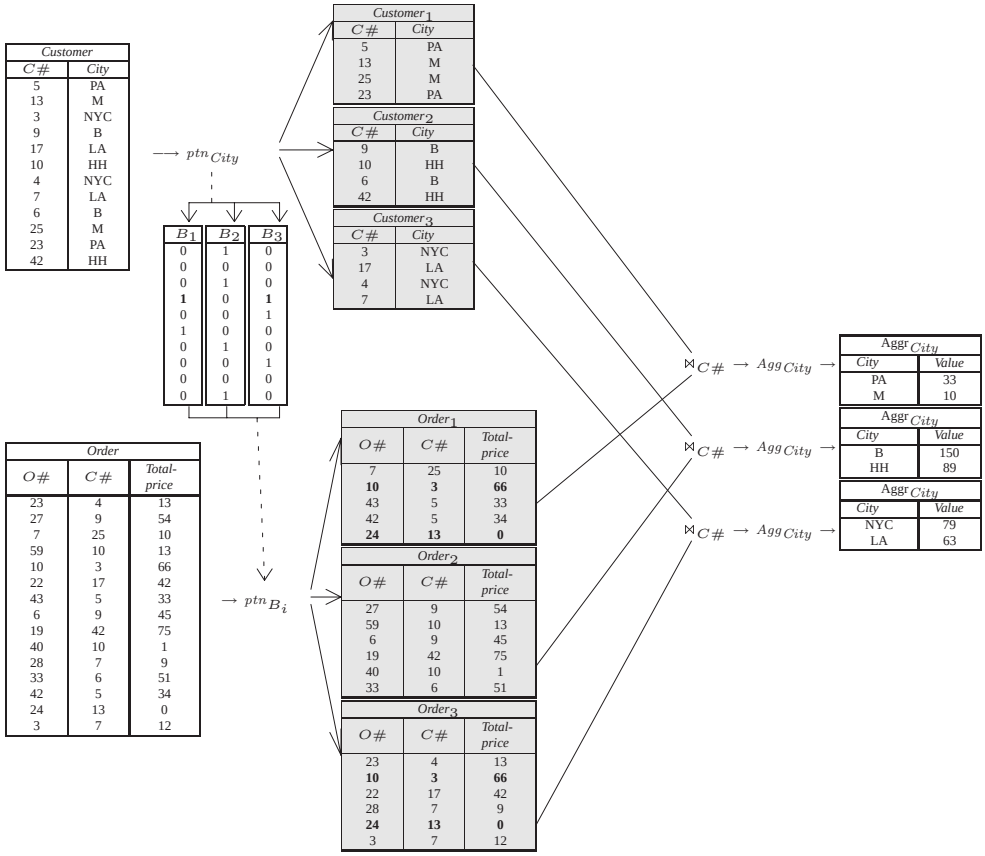


Fig. 11. Example execution of a generalized hash team

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 that generalized hash teams not be used 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*.

3.2 Multi-way joins with aggregation

Generalized hash teams can also be applied to multi-way joins. For illustration, let us look at Query 4 $(\mathcal{G}_{City}(\mathcal{C} \bowtie \mathcal{O} \bowtie \mathcal{L}))$.

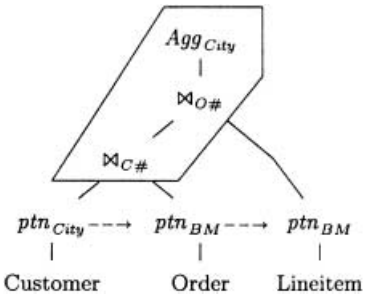
This is a three-way (functional) join of *Customer*, *Order*, and *Lineitem* followed by a grouping on the *City* attribute of *Customer*. Generalized hash teams are applicable by partitioning the *Customer* table by *City*, thereby constructing bitmaps in order to guide the partitioning of the *Order* table, as in the binary case described in Sect. 3.1. While partitioning the *Order* table, another set of bitmaps is constructed, and this set

select c.City, **sum**(l.Extendedprice)
from Customer c, Order o, Lineitem l
where c.C# = o.C# **and** l.O# = o.O#
group by c.City

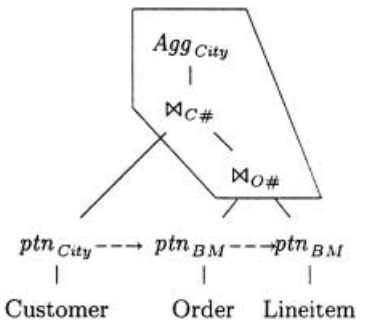
Query 4: Two joins and aggregation $(\mathcal{G}_{City}(\mathcal{C} \bowtie \mathcal{O} \bowtie \mathcal{L}))$

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 12 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 Fig. 12 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 Sect. 3.4, however, does help to reduce the memory requirements.) In the



a *Customer* or *Order* as Probe Input



b *Order* or *Lineitem* as Probe Input

Fig. 12a,b. Alternative query evaluation teams for the three-way join

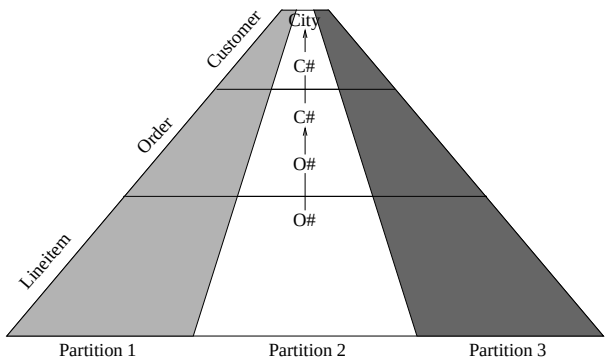


Fig. 13. Indirectly partitioning a hierarchical structure

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 *Lineitem*s, only the *Order* bitmaps are relevant (the *Customer* bitmaps can be discarded at that point).

Query 4 is a "classical" case in which to employ generalized hash teams, because the join/grouping columns form a hierarchy as can be derived from the functional dependencies $City \leftarrow C\# \leftarrow O\#$. This hierarchy of the relations is illustrated in Fig. 13. Indirect partitioning works particularly well for such hierarchical structures because, conceptually, the *cross-relation* partitions (denoted as *Partition 1*, *Partition 2*, and *Partition 3*, and indicated by 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-

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

a Partitioning of *Customer*

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

b Partitioning of *Order*

Fig. 14a,b. Partitioning the two input relations

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.

3.3 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 shown in Fig. 14a.

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 shown in Fig. 14b.

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.

3.3.1 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 a 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 will not 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*, to identify these objects.

The *used* bitmap can easily be computed by applying the component-wise *or* operation to the partitioning bitmaps $B_1, B_2, \ldots$. The *coll* bitmap is set at position k if at least two bitmaps, B_i and B_j, are set at position k. In our system, both bitmaps are actually computed during the partitioning of the *Customer* table.

3.3.2 Increasing locality on bitmaps

We can also tune the storage structure of the bitmaps in order to increase (processor) 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 $(n + 2) \cdot b$ by mapping the two-dimensional array in *row major sequence* into a one-dimensional vector. This way, the inner partitioning loop for the *Orders* can typically be carried out with a single processor cache miss.

3.4 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 straightforward 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] – so that they switch to the next partition synchronously.

We will now present 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 Fig. 15.

Let us first concentrate on the build phase, during which the hash table for the i-th partition C_i is constructed – shown in Fig. 15a. While loading the partition C_i, two hash tables are maintained: one hash table called *C.C#-HT* on the join column $C_i.C\#$ and a second, temporary hash table, called *C.City-HT*, on the grouping column $C_i.City$. Both hash tables contain pointers into the *hash area*, in which the group entries of the join/aggregation query are constructed. That is, the *hash area* will contain one entry for every *City* value of partition C_i. Let us 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 *C.C#-HT* hash table; second, the aggregation tuple for its *City* value, *PA*, is looked up via the *C.City-HT* hash table. If this was the first C_i tuple with *City* = *PA*, a new group entry is installed in the *hash area* and the corresponding pointer is inserted into the *C.City-HT*. Third, the pointer to this group entry of the *hash area* is installed in the *C.C#-HT* 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 Fig. 15b. The temporary hash table *C.City-HT* is not required for the further processing of this partition and can be deleted. Let us now trace the *Order* tuple $[C\# = 25, Totalprice = 10]$: The *C.C#-HT* hash table is inspected and the pointer to the group entry in the *hash area* is traversed. The *Totalprice* 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 *hash area* of the i-th partition). After the current probe partition is exhausted, the result tuples are retrieved ("flushed") from the *hash area* and the computation of the next *Customer/Order* partitions starts.

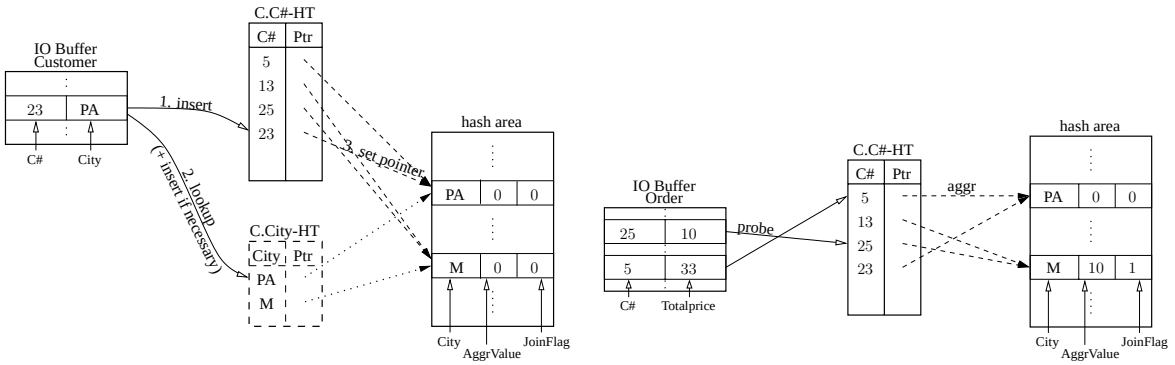
While this organization sounds complicated at first glance, it is easy to implement. The advantages are that a great deal of the main memory is saved because long strings with, say, *City* names need only be stored once in the *hash area* rather than for each *Customer* individually, and that a great 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*.

The "teaming up" of the aggregation with the preceding join can be extended to more than one join operator, as we will demonstrate in the example Query 4 ($\mathcal{G}_{City}(\mathcal{C} \bowtie \mathcal{O} \bowtie \mathcal{L})$) (page 198). This query consists of two joins followed by a grouping on the Customer's *City* (and aggregating the Lineitem's *Extendedprice*). The joins and grouping are along the functional dependencies

$$O\# \rightarrow C\# \rightarrow City.$$

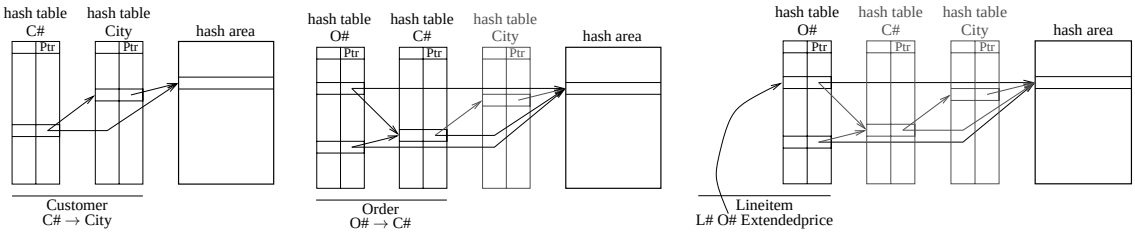
Once the bitmap-based partitioning of *Customer*, *Order*, and *Lineitem* (as described in Sect. 3.2) is finished, the first partition of *Customer* is read to build the hash area for the aggregation. As Fig. 16a demonstrates, there are two hash tables pointing to entries in the hash area: the *City* hash table and the $C\#$ hash table. These two hash tables are built as before (cf. Fig. 15). At the end of this stage (after the entire *Customer* partition is processed), the *City* hash table can be discarded. Then, the corresponding *Order* partition is read and the $O\#$ hash table is built by probing the $C\#$ hash table for every $[O\#, C\#, \ldots]$ tuple of the *Order* partition (illustrated in Fig. 16b). In this phase, (direct) pointers into the hash area are inserted into the $O\#$ hash table. After the entire *Order* partition is read, the $C\#$ hash table can be discarded and the corresponding *Lineitem* partition is read (cf. Fig. 16c). For every tuple $[L\#, O\#, Extendedprice]$ the $O\#$ hash table is probed to find the corresponding group tuple in the hash area. The *Extendedprice* is summed and the join flag is set – just as before in the binary join case of Fig. 15.

This simple scheme to team up several joins with a grouping is applicable if the aggregation only involves attribute(s) of the last relation in the chain – as is the case for Query 4. If



a Build Phase **b** Probe Phase

Fig. 15a,b. Implementation of the hash tables



a Build Phase **b** Probe Phase 1 **c** Probe Phase 2

Fig. 16a–c. Extending the teaming up concept

the attributes of intermediate relations are needed in the aggregation, they need to be stored in the hash tables. For example, if the example query involved an aggregation on *(o.Discount · l.Extendedprice)* the *Order*'s *Discount* would have to be stored in the $O\#$-hash table to "pick it up" in the probe phase with *Lineitem*s.

The main benefits of this multi-stage teaming up are again in reducing CPU costs by avoiding probing several hash tables for every *Lineitem* tuple and better main memory utilization because some hash tables can be discarded at intermediate stages of processing the hash team. As a consequence this allows larger partitions to be generated.

3.5 False drop analysis

In this section, we will summarize the formulae[7] 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 are more favorable. Using these formulae, the optimizer must be extended by formulae that estimate the overall cost of generalized hash teams and by enumeration rules that generate plans with generalized hash teams. These extensions are shown in Sect. 5 and/or are virtually the same as the extensions made in Microsoft's latest SQL Server product to integrate ordinary hash teams [GBC98].

[7] The detailed formula derivations are omitted and can be found in [KKW99].

3.5.1 Binary joins

We estimate the number of false drops for binary joins such as Query 3 ($\mathcal{G}_{City}(\mathcal{C} \bowtie \mathcal{O})$). 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.) We will use n for 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 one partition, and it is falsely 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:

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

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

$$o \cdot (n-1) \cdot \left(1 - \left(1 - \frac{1}{n \cdot b}\right)^{c-1}\right). \quad (1)$$

Unfortunately, this formula 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 the use of the following very simple approximation in order to estimate the number of false drops in a query optimizer:

$$o \cdot (n-1) \cdot \frac{c-1}{n \cdot b} \tag{2}$$

The approximation consists in assuming that no two *Customers* set the same bit in a bitmap. This formula is conservative: it can be shown that $\frac{c-1}{n \cdot b} > 1 - \left(1 - \frac{1}{n \cdot b}\right)^{c-1}$. Thus, a query optimizer using this formula will overestimate the number of false drops, and therefore it will use generalized hash teams cautiously.

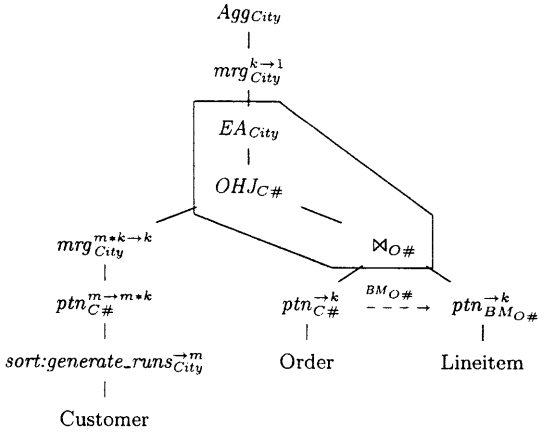


Fig. 17. Combining order-preserving hash joins and generalized hash teams

3.5.2 Multi-way joins

We will concentrate on Query 4 ($\mathcal{G}_{City}(\mathcal{C} \bowtie \mathcal{O} \bowtie \mathcal{L})$), as an example of multi-way join queries; it should be noted, however, 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 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. *Order*s placed into different *Order* partitions can have the same *O#* hash value; all *Lineitems* referring to such *Order*s produce false drops. So we get (according to Formula 2) as the number of false drops

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

   where b_o is the bitmap length for indirectly partitioning the *Lineitems*, i.e. the bitmaps generated during *Order* partitioning.
2. False drop propagation: If an *Order* produces a false drop, all the *Lineitems* that refer to that *Order* produce false drops as well. For this phenomenon we get as the number of false drops

   $$f_o \cdot \frac{l}{o}, \tag{4}$$

   where f_o is the number of *Order* false drops.

3.6 Combining order-preserving hash joins and generalized hash teams

In Fig. 17 we demonstrate how to combine the two techniques of *early sorting* and *early partitioning* on a modified version of our example Query 4 on page 198. We generate m sorted runs of the *Customer* relation on the *City* attribute. Each of the m *City*-sorted runs is k-way partitioned on $C\#$ to prepare for the subsequent order-preserving hash join. This results in $m \cdot k$ fine-grained partitions – each constituting a *City*-ordered run – of the *Customer* relation. In each phase, the OHJ merges m of the fine-grained partitions based on their *City* attribute into a single partition/run which is used as probe input. Using the OHJ algorithm the order of the *early sorted Customer* partitions will be preserved. The build input of the OHJ is created by a generalized hash team, which consists of *Order* and *Lineitem*. While partitioning *Order* by $C\#$ into k partitions, bitmaps for the resulting $O\#$ partitioning are generated. The bitmaps are used to k-way partition *Lineitem*. The result of this join, which retains the k-way partitioning by $C\#$, is kept in a main-memory hash table and the *Customer* tuples of the corresponding partition/run are probed against this hash table. The generated result tuples are ordered by *City*, so early aggregation on the *City* attribute can be performed very easily by collapsing identical *City* tuples. These duplicate-free partitions are written to disk and finally merged to do the full aggregation.

4 The bulk bypassing technique

4.1 Basic concept

Early sorting and also early partitioning have the effect that the sorting or partitioning of the base relations is often preserved throughout the entire query plan. That is, the result tuples are generated in exactly the same order as one of the argument relation's partitions. This enables us to strip off bulky attributes of this argument relation and re-merge them with the final result with very little cost. Stripping off bulky attributes saves main memory space and disk I/O if intermediate results need to be written to disk. We call this technique bulk bypassing.

4.2 Order-preserving hash joins with bulk bypassing

Let us again consider the schema of Sect. 2 with tables R and S.

If R is stored in $R.A$ order on disk (i.e., the OHJ plan case), bulk bypassing can be applied in a straightforward way: only attribute B, which is needed to compute the join, and a sequence number *Seq#*, which is used as surrogate for re-merging the *Bulk* data, are retained by R after the initial index scan of R. The join is then carried out (using attribute B) and the *Bulk* data is afterwards re-merged using the sequence

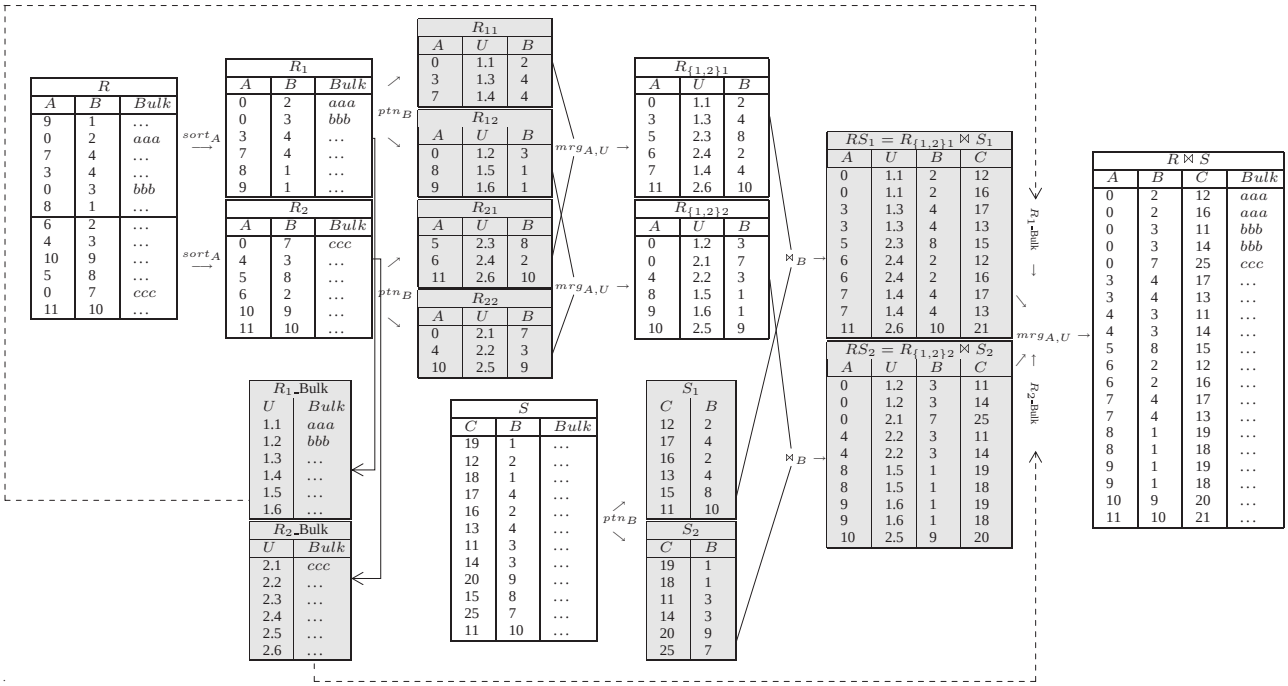


Fig. 18. Early sorting (SOHJ) and bulk bypassing

number. We will call this approach OHJ+BB in the remainder of the paper.

The reason why the OHJ+BB plan (and the TID join) only works well if R is already sorted according to A on disk is that the re-merge of the *Bulk* data using the sequence number gets prohibitively expensive due to random I/O if R is not sorted. After bringing R into $R.A$ order (i.e., as part of the $R.A$ index scan) the sequence numbers point randomly to tuples of A. Using our sorting-on-the-fly technique in conjunction with OHJs, however, we can achieve effective bypassing of bulk data with a cheap re-merge, even if R is not pre-sorted. For this purpose, we need to carry out an adjusted SOHJ plan. (This discussion assumes binary $R \bowtie S$ joins, but SOHJ plans for multi-way join queries can be adjusted just as easily.) Such a SOHJ+BB plan is illustrated in Fig. 18 for a binary join query. It involves the following three adjustments:

1. Adjust the sort&partition operator as follows: After sorting run R_i in the memory, write the *Bulk* data of the tuples of R_i in sort order (i.e., $R.A$) in a separate temporary file, *R_i-Bulk*, and assign every *Bulk* record a unique identifier $U := i.j$ consisting of the run number i followed by the position of the record j in the sorted run. Furthermore, isolate the A, U, and B columns of the run, partition the run and continue and carry out OHJs as proposed in the previous subsections. U, therefore, plays the same role in an SOHJ+BB plan as the sequence number in the OHJ+BB plan. B is needed to carry out the join, as in the OHJ+BB plan, and A is needed to re-merge intermediate results because two unique identifiers, say, $a.p$ and $b.q$ of two different runs a and b, are not comparable in terms of the sort criterion $R.A$. (In contrast, *a.Seq# < b.Seq#* always implies $a.A < b.A$ for tuples a and b of R if R is pre-sorted.)
2. The intermediate merges are performed by comparing the sort attribute A and the unique identifier $i.j$, in that precedence. This way we make sure that tuples with the same A values coming from the same run remain in the same order in which their *Bulk* data was written to disk, which is important to make the final merging of the *Bulk* efficient.
3. At the end, merge the *Bulk* with the join results using the unique identifiers; that is, we merge the partitions $RS_1, \ldots, RS_l$ (horizontally) and at the same time, we merge (vertically) the *Bulk* partitions $R_1\_Bulk, \ldots, R_m\_Bulk$. The final (vertical) merge is cheap and does not result in excessive random I/O because the *Bulk* runs are ordered in the same way as the join result, i.e., according to $R.A$.

select c.C#, c.City, **sum**(o.Totalprice)
from Customer c, Order o
where c.C# = o.C#
group by c.C#, c.City

Query 5: A bulky attribute (*City*) in the result $(\mathcal{G}_{\mathcal{C}\#,City}(\mathcal{C} \bowtie \mathcal{O}))$

4.3 Bypassing bulk around generalized hash teams

In the (S)OHJ query plans bulk bypassing was used to reduce the data volume that is written to disk at intermediate stages of the query evaluation process. In generalized hash teams this technique can also be beneficial in order to bypass bulky attributes around the join operations. This way, the individual partitions can be made larger, and therefore fewer partitions are needed. Bulk bypassing is possible if the result of the hash team is assembled in a hash area, i.e., if the hash team comprises a final grouping operation. If the grouping is on a single relation, then the bulky attributes of this relation can be bypassed. This only works if the bulk attribute(s) is/are

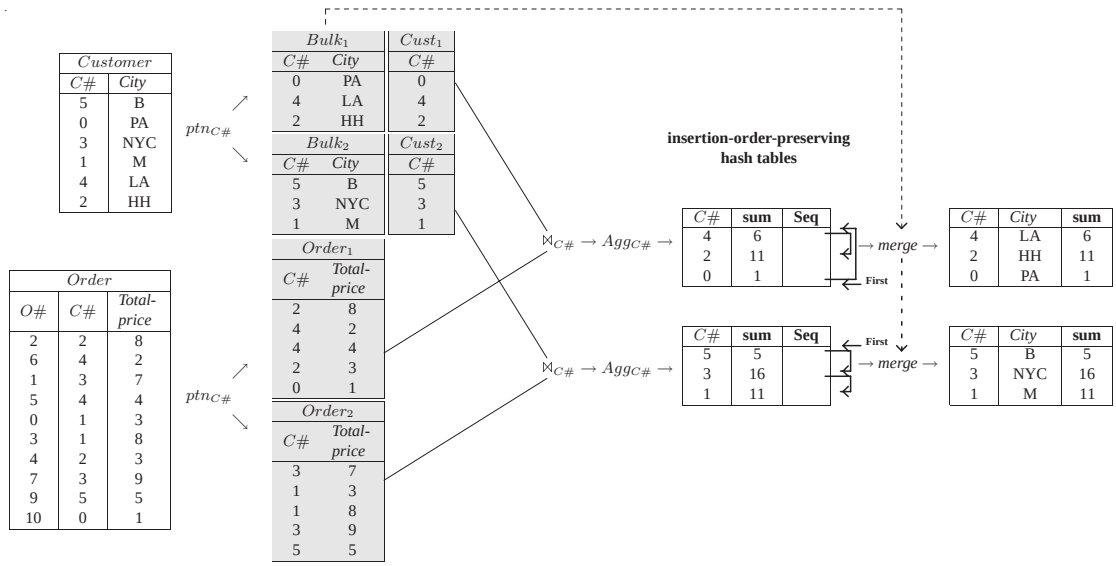


Fig. 19. Applying bulk bypassing to a team (Query 5)

functionally dependent on other grouping attributes. We will demonstrate bulk bypassing for hash teams on the example Query 5 defined on page 14. As before, we assume that *C#→ City* holds and that, therefore, the grouping can be performed on $C\#$ alone. For simplicity, we chose this example even though it is an ordinary hash team – for generalized hash teams it works analogously. Figure 19 shows the evaluation of this query.

During the partitioning phase, the bulky *City* attribute is stripped off the *Customer* partitions and stored on disk. It is important to store the [$C\#$, *City*] partitions in exactly the same order as the $C\#$ partitions. Having finished the partitioning of *Customer*s and *Order*, the actual hash teams are formed for each partition. Let us concentrate on the first partition: The $C\#$ partition, which constitutes the build input of the hash team, is read and inserted into a so-called *insertion-order-preserving hash table*. The insertion-order-preserving hash table is just a standard hash table except that the entries have an additional *Seq* pointer to chain them in insertion order. Once the join and the aggregation with the corresponding *Order* partition is completed, the hash table is read in insertion order (using the pointer chain, starting with the *First* pointer), and the corresponding bulk partition is merged. Note that the bulk partition is read in the exact order in which it was written to disk; therefore this merge is fairly cheap to perform. Our performance experiments indicate that this bulk bypassing, if applicable, can actually result in performance increase of up to a factor of two.

5 Query optimization

In this section, we show how an existing query optimizer can be extended in order to generate plans with early sorting (SOHJ), early partitioning (hash teams), and bulk bypassing. Many different query optimization architectures have been proposed in the literature and obviously there are many alternative ways to integrate SOHJ and hash teams into the query optimizer. In this section, we show how to integrate our proposed techniques into a traditional Selinger-style optimizer (i.e., bottom-up dynamic programming) [SAC+79, Loh88]. Furthermore, we show the optimizer extensions that are needed in order to exploit the full potential of our techniques, whereby we limit the generalized hash teams to hierarchies. Clearly, these extensions will significantly increase the size of the search space of the optimizer and, thus, the running time of the optimizer. We believe, however, that existing industrial-strength optimizers are capable of handling such a larger search space. For instance, early sorting (as needed for SOHJ plans) has already been implemented in the DB2 optimizer ([SSM96]), and (basic) hash teams have been implemented in Microsoft's SQL Server 7.0 ([GBC98]). In addition, heuristics such as those proposed in [KS00] can be used to speed up query optimization. Studying all the tradeoffs of optimization time vs. plan quality for supporting early sorting and partitioning is beyond the scope of this paper.

In the following, we will describe changes to the optimizer's cost model and the plan enumerator. These changes are essentially along the lines of previous work to extend bottom-up, dynamic-programming query optimizers, e.g., [Loh88, CS94,CS96,CK97].

Cost model The cost model extensions are straightforward. We only need to provide cost estimates for all new operators like OHJ, SOHJ, indirect partitioning, and BulkMerge. Cost formulae similar to those needed to estimate the cost of (S)OHJ operators have been devised in [BCK98, BCKK00] for the partition/merge algorithm. The cost of indirect partitioning and generalized hash teams strongly depends on the number of false drops. The number of false drops can be estimated as described in Sect. 3.5.

Search space Just as with "traditional optimizers," we use trees to represent plans. We consider all access paths (i.e., indices), all possible (bushy) join orders, and all common join methods (i.e., sort-merge, nested-loop, hash join). Furthermore, we consider all possible ways to apply generalized hash teams as well as (S)OHJ with and without bulk bypassing. That is, we

will consider sorting and partitioning a table by each interesting column (defined below). For ease of presentation, we will assume that group-by operators are applied after all joins, as is done by most commercial optimizers; in other words, we will ignore transformations as those proposed in [CS94,YL94] in the remainder of this section.

An interesting column of a table is any column that is used in a join or as part of an *order-by* or *group-by* clause. In modern database systems, a column used in a *rollup* or *cube* clause [GBLP96] is also interesting. In this sense, the concept of interesting column is identical to the concept of interesting order used in traditional optimizers [SAC+79]. To produce generalized hash teams, however, a column may be interesting for a table, even if it is not part of the table. In our examples of Sect. 3, for instance, *City* is an interesting column for the *Order* table and $C\#$ is interesting for the *Lineitem* table. A column which is part of Table A and not part of Table B may only be interesting for Table B, if Table A is "higher" in the hierarchy.

Properties In order to describe a (sub-)plan and simplify bottom-up plan enumeration, we annotate each (sub-)plan with properties.[8] Plan properties are also used for pruning plans during the bottom-up plan enumeration. (For brevity, we will not present the details of pruning here.) Annotating plans is also done in traditional query optimizers ([GD87, Loh88]). Typical plan properties include the *cost* of a plan, the *cardinality* of the output produced by the plan, and the *site* at which the plan is executed in a distributed system. To integrate early sorting and early partitioning, the following properties are relevant:

- *sorted by:* a set of columns indicating that the output of a plan is sorted by these columns.
- *partitioned by:* a set of columns indicating that the output of a plan is partitioned by these columns. When both *"sorted by"* and *"partitioned by"* property sets are non-empty, this indicates that the plan produces sorted runs which can be merged to produce a single unpartitioned and sorted output.
- *generated bitmaps:* a set of columns indicating that bitmaps were generated as a result of a partitioning step carried out in the plan. *Generated bitmaps* can only be non-empty if *partitioned by* is also non empty.
- *consumed bitmaps:* a set of columns indicating that bitmaps were consumed during indirect partitioning.
- *open/closed:* a Boolean value which indicates whether a plan is executable or requires bitmaps to be executable. For example, a plan that specifies that the *Lineitem* table should be partitioned by $C\#$ is not executable because the *Lineitem* table has no $C\#$ column. Such a plan would be annotated as *"open"* and it would be executed as part of a generalized hash team which involves a plan that generates bitmaps.
- *team break:* a Boolean value which indicates the border of a hash team if hash teams are split. Essentially, this property corresponds to the "pipelined/materialized" properties used in traditional optimizers and is *true* when intermediate results are written to disk.

[8] More precisely, properties are assigned to operators and the properties of a (sub-)plan are the properties of the root of the (sub-)plan.

select c.C#, c.City, **sum**(l.Extendedprice)
from Customer c, Order o, Lineitem l
where c.C# = o.C# **and** o.O# = l.O#
group by c.C#, c.City
order by c.C#

Query 6: Illustrating the optimization process
$(\mathcal{S}_{\mathcal{C}\#}(\mathcal{G}_{\mathcal{C}\#,City}(\mathcal{C} \bowtie \mathcal{O} \bowtie \mathcal{L})))$

- *bulk bypassed:* a set of (bulk) attributes which are bypassed. The full result can be computed by a vertical (merge) join with the bulk data as described in Sect. 4.

Enumerating access plans As mentioned at the beginning of this section, we consider a query optimizer that enumerates plans in a bottom-up way. Such an optimizer generates so-called access plans for all tables involved in a query in its first step. An access plan specifies how a table is read, i.e., using a full scan or an index. We propose to extend access plan generation and enumerate different access plans for all kinds of early sorting, early partitioning, and bulk bypassing. Specifically, we propose to generate the following access plans for a Table t with interesting columns c and d:

1. $sort_c(t)$: sort t by c.
2. $ptn_c(t)$, $ptn_c^{BB}(t)$: partition t by c, with and without bulk bypassing.
3. $sptn_{c,d}(t)$, $sptn_{c,d}^{BB}(t)$: partition t by c, at the same time sorting each partition by d, with or without bulk bypassing. (This is the combined sort&partition operator of SOHJ.)
4. $ptn_c(t) \rightarrow c_n$: partition t by c and generate a bitmap for c_n; such a plan is generated if c_n is a join column for a join with a table which is lower in the hierarchies than t.
5. $c_p \rightarrow ptn_{BM_{c_p}}(t)$: partition t using a bitmap; such a plan is generated if c_p is the join column for a join with a table which is higher in the hierarchy than t.
6. $c_p \rightarrow ptn_{BM_{c_p}}(t) \rightarrow c_n$: partition t using a bitmap and generating a bitmap.

Table 1 shows the access plans and their properties which are generated for Query 6. (Note that *City* is not an interesting column in this query, although it is used in the group-by clause. The reason is that *City* functionally depends on *C#*.)

Enumerating joins Just as for access plans, there is a fixed set of rules that tell the optimizer which join plans to enumerate. A join plan specifies how the results of two sub-plans are joined, depending on the properties of the sub-plans. These sub-plans may be access plans or other join plans. The following rules are applicable to enumerate (S)OHJ plans and plans with generalized hash teams. t_1 is the first sub-plan (as probe input), t_2 is the second sub-plan (as build input), and c is the join attribute.

1. $OHJ_c(t_1, t_2)$: join t_1 and t_2 using an OHJ. This plan is applicable if t_1 and t_2 are partitioned by c. If the partitioning of the result differs from the input partitioning, the fine-grained *partition–merge* operations are inserted after the join. The generated properties are:
   - *sorted by* $= t_1$.*sorted by*

Table 1. Access plans for Query 6

	C	$sort_{C\#}$	$ptn_{C\#}$	$sptn_{C\#,C\#}$	$ptn^{BB}_{C\#,C\#}$	$sptn^{BB}_{C\#,C\#}$
		C	C	C	C	C
sorted by	{}	{C#}	{}	{C#}	{}	{C#}
partitioned by	{}	{}	{C#}	{C#}	{C#}	{C#}
generated bitmaps	{}	{}	{}	{}	{}	{}
consumed bitmaps	{}	{}	{}	{}	{}	{}
open/closed	closed	closed	closed	closed	closed	closed
team break	yes	yes	yes	yes	yes	yes
bulk bypassed	{}	{}	{}	{}	{C.City}	{C.City}

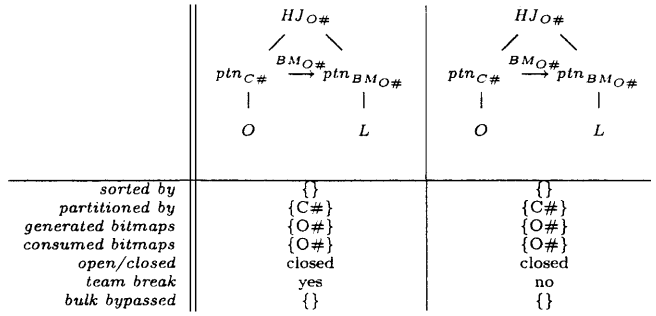
	O	$sort_{C\#}$	$sort_{O\#}$	$ptn_{C\#}$	$ptn_{O\#}$	$sptn_{C\#,C\#}$	$sptn_{C\#,O\#}$	$sptn_{O\#,C\#}$	$sptn_{O\#,O\#}$	$ptn_{C\#}^{\stackrel{BM_{O\#}}{\longrightarrow}}$
		C	C	C	C	C	C	C	C	C
sorted by	{}	{C#}	{O#}	{}	{}	{C#}	{C#}	{O#}	{O#}	{}
partitioned by	{}	{}	{}	{C#}	{O#}	{C#}	{O#}	{C#}	{O#}	{C#}
generated bitmaps	{}	{}	{}	{}	{}	{}	{}	{}	{}	{O#}
consumed bitmaps	{}	{}	{}	{}	{}	{}	{}	{}	{}	{}
open/closed	closed	closed	closed	closed	closed	closed	closed	closed	closed	closed
team break	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
bulk bypassed	{}	{}	{}	{}	{}	{}	{}	{}	{}	{}

	L	$sort_{O\#}$	$ptn_{O\#}$	$sptn_{O\#,O\#}$	$\stackrel{BM_{O\#}}{\longrightarrow} ptn_{BM_{O\#}}$
		L	L	L	L
sorted by	{}	{O#}	{}	{O#}	{}
partitioned by	{}	{}	{O#}	{O#}	{C#}
generated bitmaps	{}	{}	{}	{}	{}
consumed bitmaps	{}	{}	{}	{}	{O#}
open/closed	closed	closed	closed	closed	open
team break	yes	yes	yes	yes	yes
bulk bypassed	{}	{}	{}	{}	{}

Table 2. Selected plans for *Customer* $\bowtie_{C\#}$ *Order* (Query 6)

	$OHJ_{C\#}$ $sptn^{BB}_{C\#,C\#}$ — $ptn_{C\#}$ C — O	$mrg_{C\#}$ $ptn_{O\#}$ $OHJ_{C\#}$ $sptn^{BB}_{C\#,C\#}$ — $ptn_{C\#}$ C — O	$HJ_{C\#}$ $ptn_{C\#}$ — $ptn_{C\#}^{\stackrel{BM_{O\#}}{\longrightarrow}}$ C — O	$HJ_{C\#}$ $ptn_{C\#}$ — $ptn_{C\#}^{\stackrel{BM_{O\#}}{\longrightarrow}}$ C — O
sorted by	{C#}	{C#}	{}	{}
partitioned by	{C#}	{O#}	{C#}	{C#}
generated bitmaps	{}	{}	{O#}	{O#}
consumed bitmaps	{}	{}	{}	{}
open/closed	closed	closed	closed	closed
team break	yes	yes	yes	no
bulk bypassed	{C.City}	{C.City}	{}	{}

Table 3. Selected plans for *Order* $\bowtie_{O\#}$ *Lineitem* (Query 6)



sorted by	{}	{}
partitioned by	{C#}	{C#}
generated bitmaps	{O#}	{O#}
consumed bitmaps	{O#}	{O#}
open/closed	closed	closed
team break	yes	no
bulk bypassed	{}	{}

  - *partitioned by* $= c_n$, where c_n is an interesting column
  - *generated bitmaps* $= t_1$.*generated bitmaps* $\cup\ t_2$.*generated bitmaps*
  - *consumed bitmaps* $= t_1$.*consumed bitmaps* $\cup\ t_2$.*consumed bitmaps*
  - *open/closed* $= t_1$.*open/closed* $\wedge\ t_2$.*open/closed*
  - *bulk bypassed* $= t_1$.*bulk bypassed* $\cup\ t_2$.*bulk bypassed*
  - *team break* $=$ *yes*
2. $HJ_c(t_1, t_2)$: join t_1 and t_2 using a HJ. This plan is applicable if t_1 and t_2 are partitioned by c or $c \in t_1$.*generated bitmaps* and $c \in t_2$.*consumed bitmaps* or $c \in t_1$.*consumed bitmaps* and $c \in t_2$.*generated bitmaps*. If the generated partitioning of the result differs from the input partitioning, a *partition* operator is inserted after the join. The generated properties are:
  - *sorted by* $= \{\}$
  - *generated bitmaps* $= t_1$.*generated bitmaps* $\cup\ t_2$.*generated bitmaps*
  - *consumed bitmaps* $= t_1$.*consumed bitmaps* $\cup\ t_2$.*consumed bitmaps*
  - *open/closed* $=$ *if consumed bitmaps* $\subseteq$ *generated bitmaps then closed else open*
  - *bulk bypassed* $= t_1$.*bulk bypassed* $\cup\ t_2$.*bulk bypassed*
  - if *team break* $==$ *yes* then *partitioned by* $= c_n$, where c_n is an interesting column;
    if *team break* $==$ *no* then
    partitioned by $= t_2$.*partitioned by*

Tables 2, 3, and 4 show some sample two-way and three-way join plans enumerated for Query 6.

Postprocessing/code generation After a plan has been chosen by the optimizer, this plan is translated into an executable plan. The generation of an executable plan involves the generation of the *merge* operators for OHJ and *MergeBulk* operators for bulk bypassing. Also, *sptn* operators are replaced by a sequence of *sort–ptn–merge* operators. This postprocessing step is straightforward.

6 Performance evaluation

In this section we will present experimental results conducted using a prototypical implementation of OHJs, generalized and "ordinary" hash teams, and traditional (hash-based) algorithms to carry out joins and aggregation. We will present the running times of our example queries, using a synthetic TPC-H/R like database ([TPC99]). In contrast to the original database we used tuples and attributes of constant size. This was chosen to simplify the implementation – the comparative results of the different evaluation techniques, however, are not affected by this simplification. Our test database is characterized in Table 5.

6.1 Experimental environment

We integrated our implementation of (S)OHJ and generalized hash teams into an experimental query engine that is based on the iterator model [Gra93]. This query engine also provides iterators for traditional (hash-based) joins and aggregation. All code is written in C++. We installed the query engine on a Sun Ultra 10 with a 333 MHz processor and 128 MB of main memory. The operating system was Solaris 7. 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 are run concurrently and only a small amount of 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 and the intermediate query results were stored on a 18.2 GB IBM DNES-318350 disk drive.

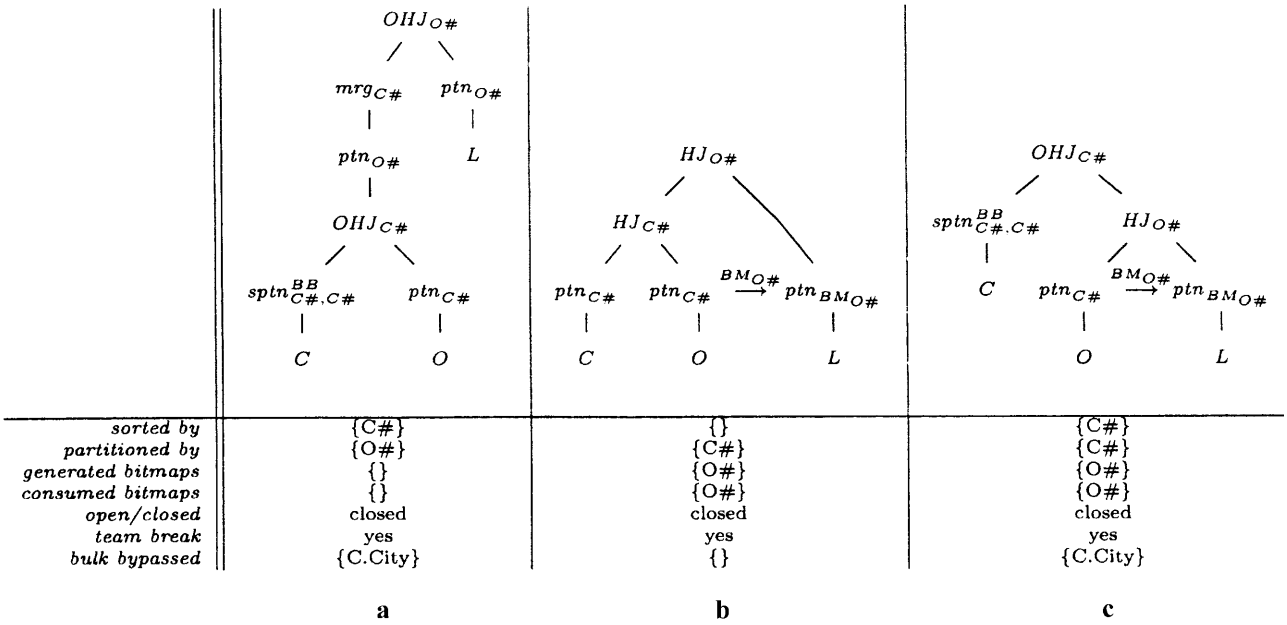
We adopted the idea presented in [DG94] to adjust the cluster size for writing and reading the partitions to the available memory. This way the number of disk seeks can be reduced enormously. In contrast to [DG94] we do not need to re-adjust the cluster size during the execution of the query because we assume constant memory size during query execution. The minimum cluster size is 4 kB, which is the physical page size, the maximum cluster size is 64 kB (16 pages $\grave{a}$ 4 kB).

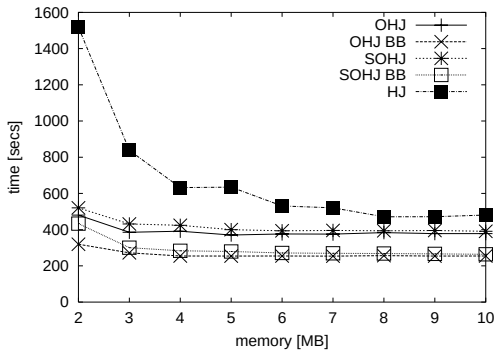
6.2 Order-preserving hash joins

Figure 20a shows the results of our implementation for the binary join plans for Query 1 ($\mathcal{S}_{Mkt,\mathcal{N}\#,\mathcal{C}\#}(\mathcal{C} \bowtie \mathcal{O})$) (page 193). The traditional hash join (HJ) plan with subsequent sorting shows the worst performance due to the expensive sort operation on the entire join result. For small memory configurations, the HJ plan shows particularly poor performance because in this case, there is not enough memory to satisfy the purposes of both the hash join and the sort at the same time – recall that these two operations run concurrently and share the available memory in this plan. The (S)OHJ plans, on the other hand, show high performance, even if memory is scarce, because no two memory-intensive operations run concurrently. Figure 20a also shows that the bulk bypassing (BB) variants of the OHJ plans yield an additional performance gain due to reduced disk I/O to write and re-read intermediate results. Evidently, the plots indicate that there is only a small difference between OHJ and SOHJ plans which proves the effectiveness of our sorting-on-the-fly approach. As a result, OHJ plans work well even in the absence of clustered indices.

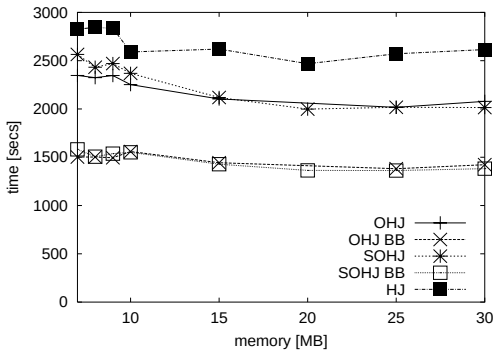
Figure 20b shows the performance results for the three-way join Query 2 ($\mathcal{S}_{Mkt,\mathcal{N}\#,\mathcal{C}\#}(\mathcal{C} \bowtie \mathcal{O} \bowtie \mathcal{L})$) (page 4). In addition to the second join operator, the OHJ plans contain the fine-grained partition/re-merge step. Evidently, the performance advantages observed for binary (S)OHJ plans are retained.

Table 4. Selected plans for *Customer* $\bowtie_{C\#}$ *Order* $\bowtie_{O\#}$ *Lineitem* (Query 6) **a** Pure SOHJ plan. **b** Pure GenTeam plan. **c** Combined SOHJ/GenTeam plan





a Response time [secs]
Query 1 ($\mathcal{S}_{Mkt,\mathcal{N}\#,\mathcal{C}\#}(\mathcal{C} \bowtie \mathcal{O})$)



b Response time [secs]
Query 2 ($\mathcal{S}_{Mkt,\mathcal{N}\#,\mathcal{C}\#}(\mathcal{C} \bowtie \mathcal{O} \bowtie \mathcal{L})$)

Fig. 20a,b. Order-preserving hash joins

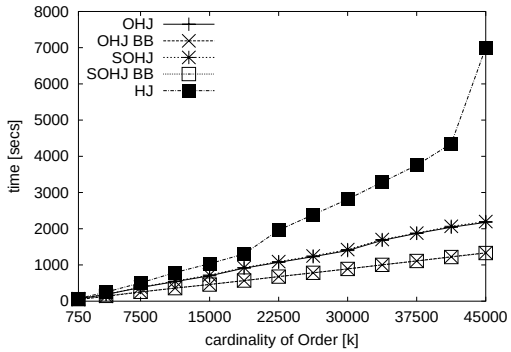


Fig. 21. Response time query 1 [secs] ($\mathcal{S}_{Mkt,\mathcal{N}\#,\mathcal{C}\#}(\mathcal{C} \bowtie \mathcal{O})$) (vary *#Order*, 10 MB memory)

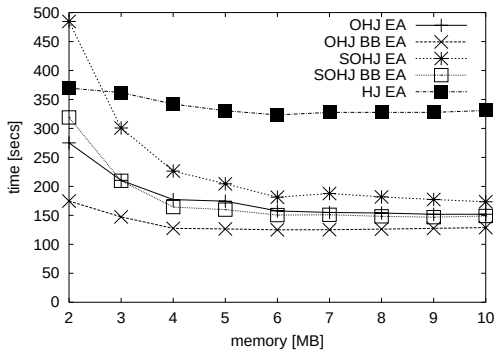
Figure 21 shows the impact of a large *Order* table on the performance of the five plans for Query 1. In this experiment the number of *Order*s per *Customer* is varied from 1 to 60, and the available main memory is fixed at 10 MB. Thus, the rightmost running times of Fig. 20a correspond to the 5 measurements at $x = 7500$, where *Customer*s have 10 *Order*s, on average. (Reported running times of the traditional HJ plans are always for the best possible join order. Note, that for large *Order* tables the HJ plans use *Order* as the outer, while *Customer* is always the outer in (S)OHJ plans.) We see that with increasing size of the *Order* table the advantages of our (S)OHJ plans

Table 5. Database characteristics

Table	*Tuple width*	*Cardinality*	*Size in MB*
Region	124 bytes	5	<4 kB
Nation	128 bytes	25	<4 kB
Supplier	160 bytes	50 000	8 MB
Customer	180 bytes	750 000	135 MB
Order	104 bytes	7 500 000	780 MB
Lineitem	112 bytes	30 000 000	3 360 MB

select c.City, **sum**(o.Totalprice)
from Customer c, Order o
where c.C# = o.C#
group by c.City
order by c.City

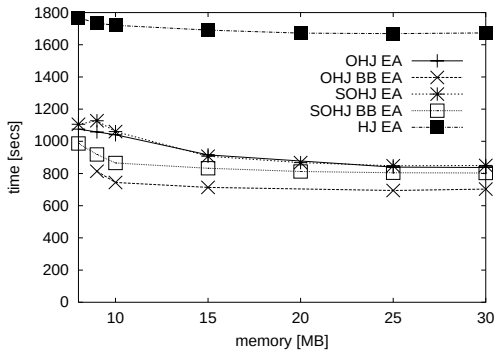
Query 7: Binary join with aggregation
$(\mathcal{S}_{City}(\mathcal{G}_{City}(\mathcal{C} \bowtie \mathcal{O})))$



a Response time Query 7 [secs]
$(\mathcal{S}_{\mathcal{C}\rangle\sqcup\dagger}(\mathcal{G}_{\mathcal{C}\rangle\sqcup\dagger}(\mathcal{C} \bowtie \mathcal{O})))$

select c.City, **sum**(l.Extendedprice)
from Customer c, Order o, Lineitem l
where c.C# = o.C# **and** o.O# = l.O#
group by c.City
order by c.City

Query 8: Multi-way join with aggregation
$(\mathcal{S}_{City}(\mathcal{G}_{City}(\mathcal{C} \bowtie \mathcal{O} \bowtie \mathcal{L})))$



b Response time Query 8 [secs]
$(\mathcal{S}_{City}(\mathcal{G}_{City}(\mathcal{C} \bowtie \mathcal{O} \bowtie \mathcal{L})))$

Fig. 22a,b. Order-preserving hash joins with aggregation

increase because the final sort of the traditional HJ plans becomes more and more expensive and dominates the cost of the whole query. The maximum performance gain of (S)OHJ plans to the HJ plans amounts up to a factor of six.

Figure 22a and b show the running times of the five plan alternatives for Query 7 $(\mathcal{S}_{City}(\mathcal{G}_{City}(\mathcal{C} \bowtie \mathcal{O})))$ and Query 8 $(\mathcal{S}_{City}(\mathcal{G}_{City}(\mathcal{C} \bowtie \mathcal{O} \bowtie \mathcal{L})))$, respectively. All plans benefit from early aggregation, but the benefits are most pronounced for those evaluation plans without bypassing bulk data, i.e., the OHJ, SOHJ and the HJ variants. These plans draw more benefit from collapsing duplicates because the size of the collapsed records is substantially larger than those of the BB variants. Therefore, the differences between the BB and standard (S)OHJ plans are less pronounced in these experiments.

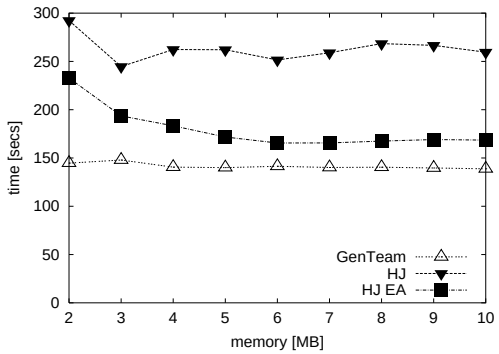
6.3 Generalized hash teams

Figure 23a shows the running time of Query 3 $(\mathcal{G}_{City}(\mathcal{C} \bowtie O))$ (page 197) using generalized hash teams and two traditional plans that use an OHJ 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[9]. We observe that, as expected, generalized hash teams 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 I/O 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 are significant. Note that for small memory size, the number of false drops is also particularly high for generalized hash teams, but the extra cost due to false drops is much lower than the cost of an extra partitioning step. With increasing memory size, the advantages of our new approaches get smaller. However, only for very large memory sizes, when the join and/or group-by can be carried out completely in the memory, do the traditional plans perform as well as our new approaches.

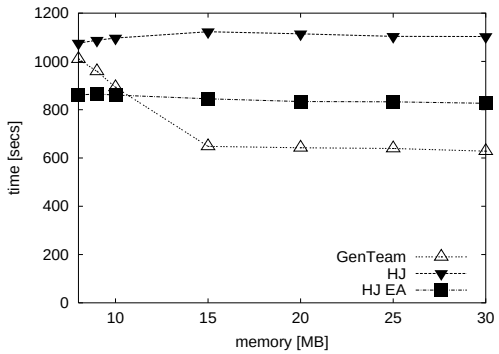
Figure 23b shows the running time of Query 4 $(\mathcal{G}_{City}(\mathcal{C} \bowtie O \bowtie L))$ (page 198) for various different plans. Again, generalized hash teams are the overall winner. In this case, however, generalized hash teams are only beneficial if a certain amount of memory is available. Recall from Sect. 3.2 that the memory requirements increase with the number of operations that participate in the team. Thus, the amount of false drops produced during partitioning the *Order* and *Lineitem* table impairs the running time at small memory configurations. 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 lose here because they require re-partitioning for the second join (i.e., the join with *Lineitem*).

We ran another experiment to show the benefits of bulk bypassing on generalized hash teams. Figure 24 shows the profit of bulk bypassing applied to Query 9 $(\mathcal{G}_{\mathcal{C}\#,City}(\mathcal{C} \bowtie \mathcal{O} \bowtie \mathcal{L}))$. At very small memory configurations the execution time of GenTeam BB is about a factor of two better than that of GenTeam because the described effect of a reduced number of partitions leads to less false drops. The more memory is allocated for the query, the more the performance gain of BB diminishes.

[9] Remember not to confuse early aggregation with early sorting or early partitioning.



a Response time Query 3 [secs] ($\mathcal{G}_{City}(\mathcal{C} \bowtie \mathcal{O})$)



b Response time Query 4 [secs] ($\mathcal{G}_{City}(\mathcal{C} \bowtie \mathcal{O} \bowtie \mathcal{L})$)

Fig. 23a,b. Generalized hash teams

6.4 Combining order-preserving hash joins and generalized hash teams

Figure 25a and b show the results of combining OHJs and generalized hash teams. We show the running times of the three-way join query *Customer* $\bowtie$ *Order* $\bowtie$ *Lineitem* with grouping and sorting on *City* (Query 8 ($\mathcal{S}_{City}(\mathcal{G}_{City}(\mathcal{C} \bowtie \mathcal{O} \bowtie \mathcal{L})))$, page 209) and a grouping and sorting on $C\#$ (Query 6 $\{\mathcal{S}_{\mathcal{C}\#}[\mathcal{G}_{\mathcal{C}\#,City}(\mathcal{C} \bowtie \mathcal{O} \bowtie \mathcal{L})]\}$, page 205) to demonstrate the impact of the size of the aggregation result. A simplified version of the SOHJ GenTeam plan of Query 8 is shown in Fig. 17. (Here we also apply early aggregation on the hash team part.) The SOHJ GenTeam plan for Query 6 is similar.

Looking at Fig. 25a for the results of Query 8, where we group on *City*, we can recognize that the SOHJ BB EA plan shows a relative constant running time, as already shown in the previous experiments of Sect. 6.2. For very small memory sizes the running times of the GenTeam variants are quite high due to many false drops. For large main memory allocations, however, the GenTeam plans turn out to be slightly better than the SOHJ plans and much better than the traditional HJ plan. The pure GenTeamSort plan (with sorting of the aggregation result) performs better than the combined SOHJ GenTeam plan for this query, because of the fewer tuples that have to be sorted: there are 75 000 cities, as opposed to to 750 000 *Customer* tuples, which are early sorted by the SOHJ.

Looking at Fig. 25b for the results of Query 6, where the grouping is done by $C\#$, the SOHJ BB EA plan shows almost the same running times as in Fig. 25a. However, now the combined SOHJ GenTeam plan performs better than the pure GenTeamSort plan, due to the bigger aggregation result. So both query evaluation plans have to sort the same number of tuples, but due to the more tuples in the aggregation result, more partitions have to be created for the pure team and so more false drops occur, which leads to longer running times. The extremely high running times (up to 6 400 secs for 5 MB) of the traditional HJ EA plan are not shown for small memory configurations.

These results show that – in order to choose the most efficient plan – it is fundamental for the application of both algorithms and the combination of them to determine the size of the aggregation result.

select c.C#, c.City, **sum**(l.Extendedprice)
from Customer c, Order o, Lineitem l
where c.C# = o.C# **and** o.O# = l.O#
group by c.C#, c.City

Query 9: Grouping by a non-bulk attribute ($\mathcal{G}_{\mathcal{C}\#,City}(\mathcal{C} \bowtie \mathcal{O} \bowtie \mathcal{L})$)

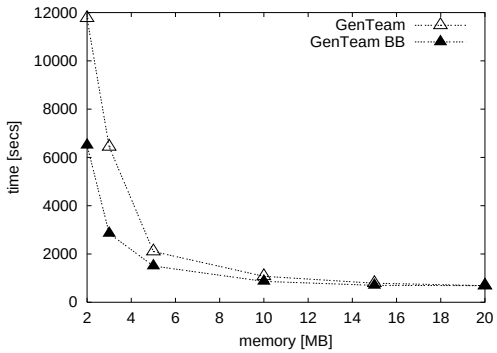
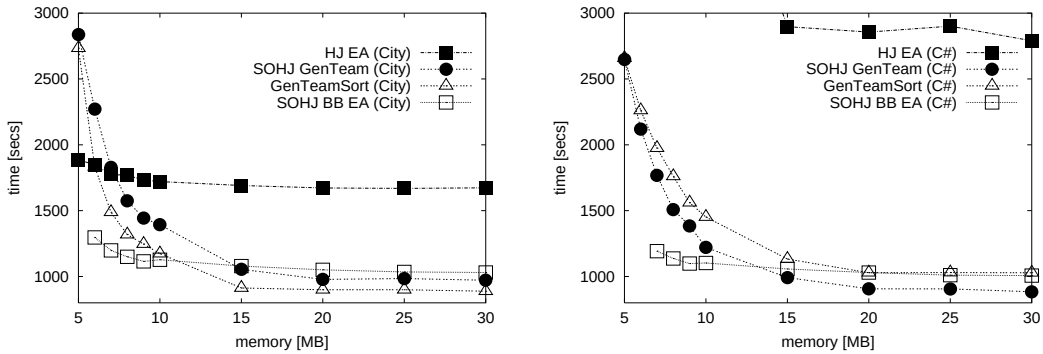


Fig. 24. Bulk bypassing in generalized hash teams: response time of Query 9 [secs] ($\mathcal{G}_{\mathcal{C}\#,City}(\mathcal{C} \bowtie \mathcal{O} \bowtie \mathcal{L})$)

6.5 Applying generalized hash teams to TPC-H/R queries

Finally, we show an application of generalized hash teams to a "real-world query" of the TPC-H/R benchmark suite. Let us consider in detail Query Q5 of the TPC-H/R suite (our concept is also applicable to other queries of TPC-H/R, e.g., Query Q10 and Query Q18):

select n.Name, **sum**(l.Extendedprice$*$(1$-$l.Discount))
as Revenue
from Customer c, Order o, Lineitem l, Supplier s,
Nation n, Region r
where c.C# = o.C# **and** o.O# = l.O#
and l.S# = S.S# **and** c.N# = s.N#
and s.N# = n.N# **and** n.R# = r.R#
and r.Name = '[region]'
and o.Orderdate $>=$ **DATE** '[date]'
and o.Orderdate $<$ **DATE** '[date]' +
INTERVAL 1 **YEAR**
group by n.Name
order by Revenue **desc**



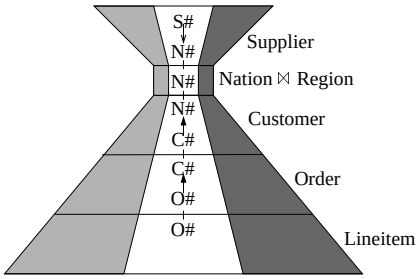
a Response time Query 8 [secs] $(\mathcal{S}_{City}(\mathcal{G}_{City}(\mathcal{C} \bowtie \mathcal{O} \bowtie \mathcal{L})))$

b Response time Query 6 [secs] $(\mathcal{S}_{C\#}(\mathcal{G}_{C\#,City}(\mathcal{C} \bowtie \mathcal{O} \bowtie \mathcal{L})))$

Fig. 25a,b. Combining OHJs and generalized hash teams

Query	Memory	Time	SF
GenTeam	8 MB	312 secs	5.0
trad. HJ	8 MB	870 secs	5.0
trad. HJ	16 MB	867 secs	5.0
RDBMS $\binom{\text{highest}}{\text{opt. level}}$	>8 MB	1021 secs	5.0
GenTeam	1.5 MB	59 secs	1.0
trad. HJ	1.5 MB	164 secs	1.0
trad. HJ	3 MB	163 secs	1.0
RDBMS $\binom{\text{highest}}{\text{opt. level}}$	>2 MB	209 secs	1.0

a Running times of different scale factors (SF)[10]



b Underlying hierarchy and three-way partitioning

Fig. 26a,b. TPC-H/R Query Q5

This query has multiple chains of functional dependencies:

$$O\# \rightarrow C\# \rightarrow N\# \rightarrow R\# \quad \text{and} \quad S\# \rightarrow N\# \rightarrow R\#.$$

These functional dependencies constitute two overlapping hierarchies as shown in Fig. 26b. So, a possible query evaluation plan using a generalized hash team could be as in Fig. 27b. The resulting partitioning of the entire hierarchy is illustrated in Fig. 26b by the three different shadings, each of which represents one partition of the entire hierarchy which is processed by a single hash team. Here we first join *Customer*, *Nation*, and *Region* to filter out the relevant *Customer* tuples. During partitioning the known set of bitmaps for indirectly partitioning the *Order* table is created. Additionally a Bloom filter ([Blo70]) for the join with *Supplier* is calculated. So the I/O volume can be reduced enormously on all participating base relations. The bitmaps created for indirect partitioning contain the described *used* bitvector, which acts as an implicit Bloom filter.

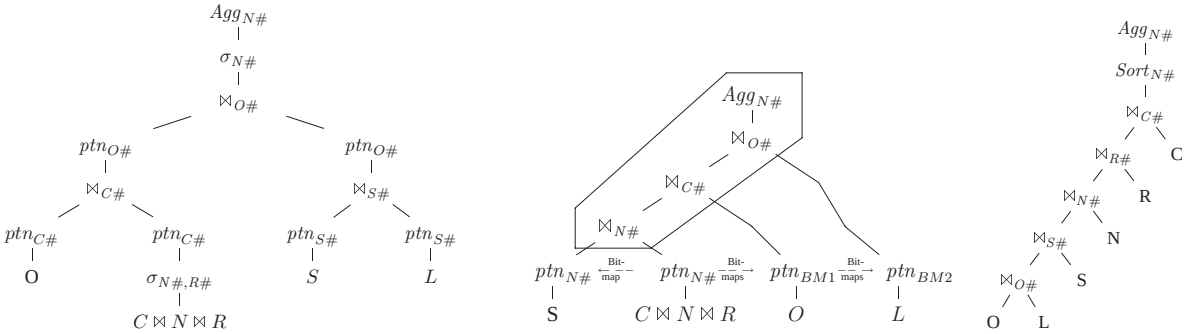
The traditional hash join plan shown in Fig. 27a has to write large volumes of intermediate results while joining the big tables *Order* and *Lineitem*. The only optimization that can be applied to the traditional plan is the final in-memory aggregation, which does not improve the performance significantly. The results of the running times are shown in Fig. 26a. In order to show the advantages of the generalized hash teams we used two scale factors of the database. The teams show in both cases superior performance. It should be noted that due to the optimized memory usage the hash teams do not require more memory than shown in the table.

We also compared our query engine with a commercial RDBMS using both scale factors 1.0 and 5.0. In the commercial RDBMS it was not possible to limit the memory for the whole plan, so each operator got 2 MB for the scale factor 1.0 and 8 MB for the scale factor 5.0. Thus, even though we could not determine the total memory exactly, the RDBMS plans consumed much more memory than we allocated for the GenTeam plans, which were limited to 1.5 MB (and 8 MB respectively) in total. The statistics for all tables and all useful indexes were generated for both database sizes. The built-in optimizer of the RDBMS has chosen the plan shown in Fig. 27c. The performance of this RDBMS plan was comparable with our chosen traditional hash join plans on our experimental query engine. The performance gain of using generalized hash teams amounted in both cases to a factor of 3.

7 Conclusions

Many queries – in particular, in OLAP and decision support applications – involve joins and grouping with aggregation and/or sorting of the result. In this paper we devised two complementary query evaluation techniques to push sorting and partitioning to the leaves of query evaluation plans. *Early sorting* and *early partitioning* are effective in many decision support queries because they allow to perform costly operations

[10] It could not be determined how much memory the RDBMS allocated for the whole plan. Each operator was limited to the denoted memory.



a Traditional hash join with in-memory aggregation **b** Generalized hash team **c** Plan generated by RDBMS

Fig. 27a–c. Possible QEPs for TPC-H/R Query Q5

(involving disk I/O) on relatively small base relations instead of on intermediate results which become very large in some OLAP queries.

Order-preserving hash joins allow an existing order of one of the base relations to be preserved or the costly run generation phase of the sorting to be pushed to a base relation. This early sorting is applicable if the desired sort order of the query is based on attributes of a single base relation only. One particular advantage of order-preserving hash join plans is that they reduce the main memory requirements of queries because memory-intensive operations such as the probe phase of a join and the make-run phase of a sort are not carried out concurrently. A great additional performance gain can be achieved for many group-by queries by exploiting early aggregation.

The early partitioning technique is based on Graefe et al.'s [GBC98] *hash teams*, which were integrated into Microsoft's SQL Server product. This technique allows one to "team up" several join (and grouping) operators; however, in Microsoft's hash teams, all operators must involve the *same* attribute. This, of course, restricts the applicability of their approach to a very special class of queries. We proposed generalized hash teams which allow one to "team up" join and grouping operators even if they are based on different columns. This 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. We presented details of such generalized hash teams and showed 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. We also showed how to combine early sorting and early partitioning.

In order to further reduce the size of intermediate results, we investigated the bulk-bypassing technique for order-preserving hash joins and hash teams. This technique allows one to bypass large (bulky) attributes of one relation around expensive operations such as joins. We showed how an optimizer could be extended to enumerate all proposed algorithms. Lastly we carried out experiments demonstrating the usefulness of early sorting and early partitioning for many classes of decision support queries. We compared the individual algorithms and combinations of them. We also investigated the applicability of our early sorting and partitioning plans for "real-world" queries of the TPC-H/R benchmark suite. Several of the queries can be optimized using our approach, we studied one query (Q5) in detail and found out that we could achieve an improvement of a factor 3 compared to a traditional plan of a commercial RDBMS product.

Acknowledgements. We would like to thank Christoph Pesch for his help on the estimation of the false drops and Konrad Stocker for his comments on the integration of *early sorting* and *early partitioning* into an optimizer. We acknowledge the helpful comments of the anonymous reviewers.

References

[BCK98] Braumandl R., Claussen J., Kemper A. Evaluating functional joins along nested reference sets in object-relational and object-oriented databases. In: Proc. of the Conf. on Very Large Data Bases (VLDB), New York, USA, August 1998, pp. 110–121

[BCKK00] Braumandl R., Claussen J., Kemper A., Kossmann D. Functional join processing. The VLDB Journal 8(3-4): 156–177, 2000.

[BD83] Bitton D., DeWitt D.J. Duplicate record elimination in large data files. ACM Trans. on Database Systems 8(2): 255–265, 1983

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

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

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

[CK97] Carey M., Kossmann D. On saying "enough already!" in SQL. In: Proc. of the ACM SIGMOD Conf. on Management of Data, pp. 219–230, Tucson, Ariz., USA, May 1997

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

[CS96] Chaudhuri S., Shim K. Optimization of queries with user-defined predicates. In: Proc. of the Conf. on Very

Large Data Bases (VLDB), pp. 87–98, Bombay, India, September 1996

[DG94] Davison D.L., Graefe G. Memory-contention responsive hash joins. In: Proc. of the Conf. on Very Large Data Bases (VLDB), pp. 379–390, Santiago, Chile, September 1994

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

[GBLP96] Gray J., Bosworth A., Layman A., Pirahesh H. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In: Proc. IEEE Conf. on Data Engineering, pp. 152–159, New Orleans, La., USA, February 1996

[GD87] Graefe G., DeWitt D. The EXODUS optimizer generator. In: Proc. of the ACM SIGMOD Conf. on Management of Data, pp. 160–172, San Francisco, Calif., USA, May 1987

[GKG+97] Grust T., Krüger J., Gluche D., Heuer A., Scholl M.H. Query evaluation in CROQUE – calculus and algebra coincide. In: Proc. British National Conference on Databases (BNCOD), pp. 84–100, London, UK, July 1997

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

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

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

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

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

[KKW99] Kemper A., Kossmann D., Wiesner C. Generalized hash teams for join and group-by. In: Proc. of the Conf. on Very Large Data Bases (VLDB), pp. 30–41, Edinburgh, UK, September 1999

[KS00] Kossmann D., Stocker K. Iterative dynamic programming: A new class of query optimization algorithms. ACM Trans. on Database Systems 25(1), March 2000, in press

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

[Loh88] Lohman G. Grammar-like functional rules for representing query optimization alternatives. In: Proc. of the ACM SIGMOD Conf. on Management of Data, pp. 18–27, Chicago, Ill., USA, May 1988

[LR99] Li Z., Ross K.A. Fast joins using join indices. The VLDB Journal 8(1): 1–24, May 1999

[ME92] Mishra P., Eich M. Join processing in relational databases. ACM Computing Surveys 24(1): 63–113, March 1992

[MR94] Marek R., Rahm E. TID hash joins. In: International Conference on Information and Knowledge Management (CIKM), pp. 42–49, Gaithersburg, Md., USA, 1994

[SAC+79] Selinger P., Astrahan M., Chamberlin D., Lorie R., Price T. Access path selection in a relational database management system. In: Proc. of the ACM SIGMOD Conf. on Management of Data, pp. 23–34, Boston, USA, May 1979

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

[SSM96] Simmen D., Shekita E., Malkemus T. Fundamental techniques for order optimization. In: Proc. of the ACM SIGMOD Conf. on Management of Data, pp. 57–67, Montreal, Canada, June 1996

[TPC99] Transaction Processing Performance Council TPC. TPC benchmarks H and R (decision support). Standard Specification, Transaction Processing Performance Council (TPC), October 1999. http://www.tpc.org/

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

[YL95] Yan W.P., Larson P.A. Eager aggregation and lazy aggregation. In: Proc. of the Conf. on Very Large Data Bases (VLDB), pp. 345–357, Zürich, Switzerland, September 1995

[ZDNS98] Zhao Y., Deshpande P., Naughton J., Shukla A. Simultaneous optimization and evaluation of multiple dimensional queries. In: Proc. of the ACM SIGMOD Conf. on Management of Data, pp. 271–282, Seattle, Wash., USA, June 1998