Universität Passau
Technical Report MIP-9810

Order-Preserving Hash Joins: Sorting (Almost) For Free*

Jens Claussen   Alfons Kemper   Donald Kossmann

August 19, 1998

Universität Passau
Lehrstuhl für Informatik
94030 Passau, Germany
⟨lastname⟩*@db.fmi.uni-passau.de*
http://www.db.fmi.uni-passau.de/

Abstract

Database systems must be able to produce ordered query results; either to pass them to application programs or end users or to compute standard or modern database operations for decision support such as aggregation, moving sums, moving averages, top N or bottom N. In this paper, we present a new approach that significantly speeds up the ability of database systems to produce ordered query results for join queries. We call the key element used in this approach *order-preserving hash joins* (or *OHJ* for short). Just like the well-known (index) nested-loop join method, OHJ preserves the order of one of its input tables, and thus, OHJ makes it possible to use indices or *early sort operations* in order to carry out the *ordering* part of a query very cheaply. Other than nested-loop joins, however, OHJs show good performance even if the tables involved in the query are very large, and thus, OHJs are also able to carry out the *joining* part of a query efficiently. We discuss a series of examples for which order-preserving hash joins are applicable and present the results of performance analyses which demonstrate that order-preserving hash joins significantly speed up the execution of many important classes of decision support queries.

1 Introduction

The ability to produce ordered results is one of the most important features of a query processor. First of all, this feature is important because users often like to have the final results of their queries ordered (i.e., they make use of SQL's **order by** clause). Secondly, this feature is important in order to achieve good performance because several algorithms for, say, join or group-by processing require ordered inputs (i.e., ordered intermediate results). We believe that this feature is going to become even more important with the current trend to extend database systems with new operations and to construct decision

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

support systems that ask for ordered results on top of relational database systems. Consider, for example, the *rollup* operator which can be used to compute *moving sums* and *moving averages* and which is implemented in, e.g., Microsoft's SQL Server 6.5 product and in several middleware products for decision support (e.g., SAP's business information warehouse [DHKK97, KKM98]). To work right, the *rollup* operator requires its input to be ordered according to the columns specified in the **group by ... with rollup** clause of the query (e.g., *mktsement, nationkey,* and *custkey* in a query that sums up the customers' order values and computes moving sums by nation and market segment). The *cube* operator is another example that benefits from ordered query results. If the input of the *cube* operator is ordered, then the *PipeSort* algorithm of [AAD+96] can be applied, thereby saving the cost for the first and most expensive sort. Further examples include operators for ranking and *top N* or *bottom N* queries; these operators also work best if their input is already in the right order [CK97], and such operators have also been implemented as part of relational database products and middleware products for decision support [KS95].

There are many different strategies to obtain ordered (final or intermediate) query results, and it is the job of the query optimizer of a database system to determine the best overall strategy (i.e., plan). How the optimizer can find the best plan has been the subject of previous work on query optimization (e.g., [SAC+79, SSM96]), and one of the toughest decision the optimizer faces is where to place *sort* operators in a query plan. Given the current set of available join methods, the optimizer often faces 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; this is called *sort-ahead* in [SSM96]. On the other hand, *sort-ahead* limits the options for join processing so that *sort-ahead* might be the cause for very high costs for join processing. To solve this dilemma, we propose a new approach which makes it possible to do *sort-ahead* and have cheap joins at the same time. Our approach is based on a new technique that we call *order-preserving hash joins* (or just OHJ) and that can be used instead of nested-loop joins in order to preserve the order generated by *sort-ahead*.

We will clarify these observations and the advantages of our new approach in more detail using simple example queries in Section 2. The remainder of this paper is then organized as follows: Section 3 describes our new approach and how it can be used to improve the performance of several classes of queries, including the simple example queries of Section 2 and aggregate queries which can efficiently be computed using OHJs and sort-based grouping with early aggregation [BD83]. Section 3 shows, furthermore, how our approach can be extended with "TID join"-like techniques in order to achieve further performance improvements. Section 4 presents the results of performance experiments that confirm the usefulness of our approach. Section 5 contains our conclusions and suggestions for future work.

2 Running Examples

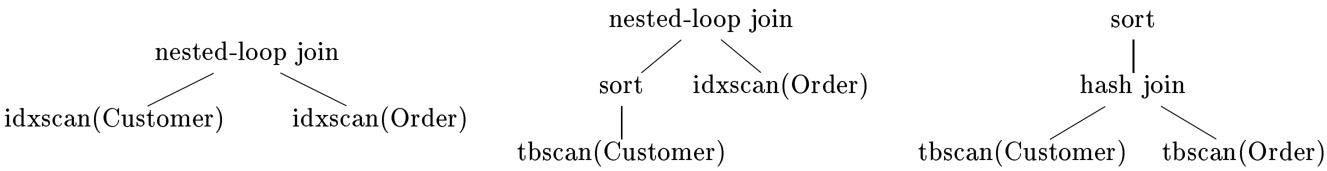
To demonstrate the mechanisms and the benefits of order-preserving hash joins, we will use the two example queries shown in Figures 1 and 2. Both queries involve a database with *Customer*, *Order*, and *Lineitem* tables and the usual (e.g., TPC-D style) information (e.g., *Customer.name*, *Customer.address*, *Lineitem.extendedprice*, etc.). We assume, as

```
select c.*, o.totalprice
from Customer c, Order o
where c.custkey = o.custkey
order by c.mktsegment, c.nationkey, c.custkey
```

Figure 1: Query 1

```
select c.*, l.extendedprice
from Customer c, Order o, Lineitem l
where c.custkey = o.custkey and
      o.orderkey = l.orderkey
order by c.mktsegment, c.nationkey, c.custkey
```

Figure 2: Query 2



Index Scan + NLJ | Sort + NLJ | HHJ + Sort

Figure 3: Traditional Plans for Query 1

in reality, that the *Customer* table contains significantly less tuples than the *Order* and *Lineitem* tables. The first query involves a join between the *Customer* and *Order* tables and requires the results to be produced in *Customer.mktsegment, Customer.nationkey, Customer.custkey* order. The second query involves, in addition, a join with the *Lineitem* table. As described in the introduction, both queries 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, or these queries could be seen as a query block that produces the input of a *rollup* operator implemented as part of an extended relational database system.

Figure 3 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 nested-loop join, which is the only known order-preserving join applicable in this case, will be very high because nested-loop joins (indexed as well as non-indexed) are not appropriate to join the *Customer* and *Order* tables. 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 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 this paper is to break this dilemma and allow the optimizer to generate plans like the ones shown in Figures 4 and 5. The key idea is to split the different phases of external sorting and carry out join operations in between the *make_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 order-preserving hash join exhibits the same good performance as standard hash joins. As

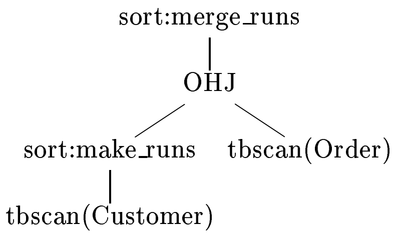


Figure 4: SOHJ Plan for Query 1

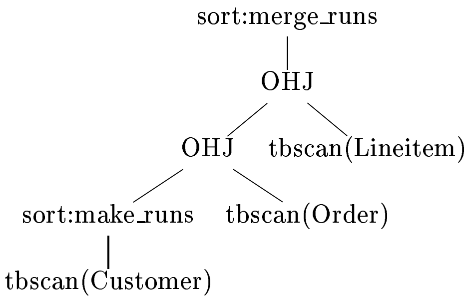


Figure 5: SOHJ Plan for Query 2

shown in Figure 5, it is possible to have any number of OHJ operators in a query and as we will see in the next section, 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.

Our performance experiments demonstrate that SOHJ and OHJ query plans show significantly better performance than traditional query plans in many situations. There is one aspect, however, that somewhat limits the applicability and usefulness of our approach: OHJ operators make only sense if the outer table (i.e., the probe input) is in the right order. That is, the joins with the *Customer* table in the plans of Figures 4 and 5 cannot freely be reordered for our example queries: *Customer* must always be the outer. Traditional hash join plans, on the other hand, can freely be reordered. The query processor could, for example, reorder the join of the third plan of Figure 3 so that *Order* is the outer table, and this reordering improves the plan if the size of the *Order* table after projecting out the *o.custkey* and *o.totalprice* columns is larger than the size of the *Customer* table. A query optimizer should, therefore, consider the SOHJ plan shown in Figure 4 in addition to the traditional plans of, say, Figure 3 and choose the overall cheapest plan depending on the database configuration and characteristics of the query (selectivity of predicates, used columns, etc.). Likewise, the optimizer should consider the plan of Figure 5 in addition to traditional plans for Query 2. For Query 2, the optimizer should, furthermore, consider a plan that uses, say, a hash join to join the *Order* and *Lineitem* tables and then OHJ to join the result of that join with the (sorted) *Customer* table.

In terms of related work, there has, of course, been a great deal of work on join techniques, sorting, and query processing in general. A good overview of all join techniques used in practice today is given in [ME92], and [Gra94] describes 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 our OHJ approach into an existing query processor that is based on that architecture as part of our experimental work, and just as well, our OHJ approach can be integrated with very modest 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 seventies; see, e.g., [SAC+79] or [SSM96] for a more recent paper which specifically addresses *sort-ahead*. As described above, our work builds on top of that work, and the purpose of our work is to provide the optimizer with new options to construct *sort-ahead* plans. Related query optimization work also includes work on "group-bys before joins" [YL94, CS94]; again our work complements that work as we propose ways to further improve the performance of the "eager" group-by plans proposed in that work. In our own previous work, we proposed the $P(PM)^*M$-algorithm which is based on a similar idea to partition nested reference sets before a functional join and merge the partitions after the join [BCK98]. The $P(PM)^*M$ algorithm, however, was 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, OHJs are order-preserving (not just "nested-set" preserving) and applicable in pure relational as well as object-oriented and object-relational database systems.

3 The Order-Preserving Hash Join Plans

In this section, we present the full range of order-preserving hash join plans. We begin with simple (binary) and multi-way OHJ plans that work if there is a clustered, ordered index which can be used to get the right ordering. We then present the details of SOHJ plans which are applied in the absence of such an index. After that, we present a technique that we call "bypassing bulk data" which can be used to improve the performance of OHJ as well as SOHJ plans, and we show how OHJ and SOHJ plans can be used to improve the performance of aggregate queries.

3.1 Simple (Binary) OHJ Plans

Order-preserving hash joins are based on Grace hash joins as described in [HCLS97]. That is, both input relations are partitioned using hashing in such a way that all partitions of the inner (build) relation fit in 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 Figure 6 that 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. 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 those simple order-preserving hash joins. 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 with small main memory) or no partitioning at all is required.

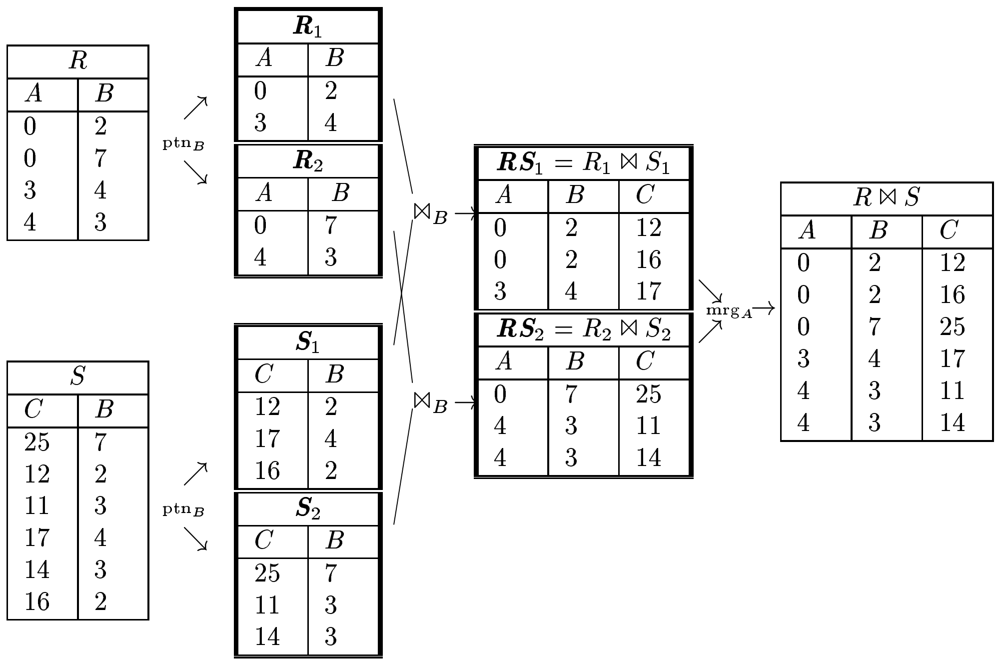


Figure 6: Order-Preserving Binary Hash Join Plan: Build Input S and Probe Input R

3.2 Multi-Way OHJ 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 example Query 2 of Figure 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 Figure 5. Here, we must be careful, however, because we cannot afford using two simple OHJs. Such a naive implementation would involve a full-fledged *merge* step as part of the R *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 $k * l$ partitions on disk. These $k * l$ 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 Figure 7 for $k = 2$ and $l = 2$, and step-by-step the algorithm works as follows:

R	
A	B
0	2
0	7
0	3
3	4
4	3
5	8
6	2

ptn_B

$\boldsymbol{R_1}$	
A	B
0	2
3	4
5	8
6	2

$\boldsymbol{R_2}$	
0	7
0	3
4	3

S	
C	B
12	2
17	4
14	3
19	8
16	2
13	4
11	3
25	7

ptn_B

$\boldsymbol{S_1}$	
C	B
12	2
17	4
19	8
16	2
13	4

$\boldsymbol{S_2}$	
14	3
11	3
25	7

$\bowtie_B \rightarrow \text{ptn}_C$

$\bowtie_B \rightarrow \text{ptn}_C$

$\boldsymbol{RS_{11}}$		
A	B	C
0	2	12
0	2	16
6	2	12
6	2	16

$\boldsymbol{RS_{12}}$		
A	B	C
3	4	17
3	4	13
5	8	19

$\boldsymbol{RS_{21}}$		
A	B	C
0	3	14
4	3	14

$\boldsymbol{RS_{22}}$		
A	B	C
0	7	25
0	3	11
4	3	11

T	
C	D
12	38
17	34
16	35
12	48
13	33
14	37
11	36
19	40
25	30

ptn_C

$\boldsymbol{T_1}$	
C	D
12	38
16	35
12	48
14	37

$\boldsymbol{T_2}$	
17	34
13	33
11	36
19	40
25	30

mrg_A

mrg_A

$\bowtie_C \rightarrow$

$\bowtie_C \rightarrow$

$\boldsymbol{RST_1}$			
A	B	C	D
0	2	12	38
0	2	12	48
0	2	16	35
0	3	14	37
4	3	14	37
6	2	12	38
6	2	12	48
6	2	16	35

$\boldsymbol{RST_2}$			
A	B	C	D
0	7	25	30
0	3	11	36
3	4	17	34
3	4	13	33
4	3	11	36
5	8	19	40

$\text{mrg}_A \rightarrow$

$R \bowtie_B S \bowtie_C T$			
A	B	C	D
0	2	12	38
0	2	12	48
0	2	16	35
0	3	14	37
0	7	25	30
0	3	11	36
3	4	17	34
3	4	13	33
4	3	11	36
4	3	14	37
5	8	19	40
6	2	12	38
6	2	12	48
6	2	16	35

Figure 7: Order-Preserving Three-Way Hash Join Plan: $(R \bowtie_B S) \bowtie_C T$

The probe input R is scanned in sort order (via a clustered index on $R.A$—not shown) and partitioned (ptn_B) for the first join. Disk partitions are marked with thick rules; the first join pipes its result tuples into the "fine-grained partitioner" ptn_C which writes the partitions RS_{11} and RS_{12} of $R_1 \bowtie_B S_1$ and the partitions RS_{21} and RS_{22} of $R_2 \bowtie_B S_2$ onto disk; the subsequent merge (mrg_A) operation pipes its tuples directly into the (probing of the) final hash join $\bowtie_C$; this join writes the partitions RST_1 and RST_2 onto disk; the final merge restores the order of the entire join result.

1. Scan S and partition S into k main memory-sized[1] 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[2] 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 $k * l$ 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 using order-preserving hash joins: 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:

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

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

[1] 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.

[2] 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.

3.3 SOHJ Plans: Sorting on The Fly and (Almost) For Free

One might argue that our order-preserving hash join technique is only efficient if there is a clustered index on 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 the performance section, 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[3] partitions $S_1, \ldots, S_k$ using a hash function h_k on $S.B$
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 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, there are $m * k$ partitions $R_{11}, \ldots, R_{1k}, \ldots, R_{mk}$—each constituting a valid sort run—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 Figure 8. 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. The remainder of the

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

R

A	B	*Bulk*
9	1	...
2	2	...
7	4	...
3	4	...
1	3	...
8	1	...
6	2	...
4	3	...
10	9	...
5	8	...
0	7	...
11	10	...

$\xrightarrow[\text{sort}]{\text{load\&}}$

R_1

A	B	*Bulk*
1	3	...
2	2	...
3	4	...
7	4	...
8	1	...
9	1	...

R_2

A	B	*Bulk*
0	7	...
4	3	...
5	8	...
6	2	...
10	9	...
11	10	...

ptn_B

$\boldsymbol{R}_{11}$

A	B	*Bulk*
2	2	...
3	4	...
7	4	...

$\boldsymbol{R}_{12}$

A	B	*Bulk*
1	3	...
8	1	...
9	1	...

$\boldsymbol{R}_{21}$

A	B	*Bulk*
5	8	...
6	2	...
11	10	...

$\boldsymbol{R}_{22}$

A	B	*Bulk*
0	7	...
4	3	...
10	9	...

$\text{mrg}_A \rightarrow$

$R_{\{1,2\}1}$

A	B	*Bulk*
2	2	...
3	4	...
5	8	...
6	2	...
7	4	...
11	10	...

$R_{\{1,2\}2}$

A	B	*Bulk*
0	7	...
1	3	...
4	3	...
8	1	...
9	1	...
10	9	...

S

C	B
19	1
12	2
18	1
17	4
16	2
13	4
11	3
14	3
20	9
15	8
25	7
21	10

ptn_B

$\boldsymbol{S}_1$

C	B
12	2
17	4
16	2
13	4
15	8
21	10

$\boldsymbol{S}_2$

C	B
19	1
18	1
11	3
14	3
20	9
25	7

$\bowtie_B \rightarrow$

$\boldsymbol{RS}_1 = \boldsymbol{R}_{\{1,2\}1} \bowtie \boldsymbol{S}_1$

A	B	C	*Bulk*
2	2	12	...
2	2	16	...
3	4	17	...
3	4	13	...
5	8	15	...
6	2	12	...
6	2	16	...
7	4	17	...
7	4	13	...
11	10	21	...

$\boldsymbol{RS}_2 = \boldsymbol{R}_{\{1,2\}2} \bowtie \boldsymbol{S}_2$

A	B	C	*Bulk*
0	7	25	...
1	3	11	...
1	3	14	...
4	3	11	...
4	3	14	...
8	1	19	...
8	1	18	...
9	1	19	...
9	1	18	...
10	9	20	...

mrg_A

$R \bowtie S$

A	B	C	*Bulk*
0	7	25	...
1	3	11	...
1	3	14	...
2	2	12	...
2	2	16	...
3	4	17	...
3	4	13	...
4	3	11	...
4	3	14	...
5	8	15	...
6	2	12	...
6	2	16	...
7	4	17	...
7	4	13	...
8	1	19	...
8	1	18	...
9	1	19	...
9	1	18	...
10	9	20	...
11	10	21	...

Figure 8: Sorting On the Fly (Almost) For Free

Disk partitions are marked with thick rules; the initial sorting and the partitioning is combined such that the run R_1 is directly written into its fine-grained partitions R_{11}, R_{12} on disk and the run R_2 into its fine-grained partitions R_{21}, R_{22}; the subsequent merge operation pipes its tuples directly into the join, i.e., the runs $R_{\{1,2\}1}$ and $R_{\{1,2\}2}$ are not buffered but rather directly piped into the (probing of the) hash join as soon as the tuples emanate from the merge.

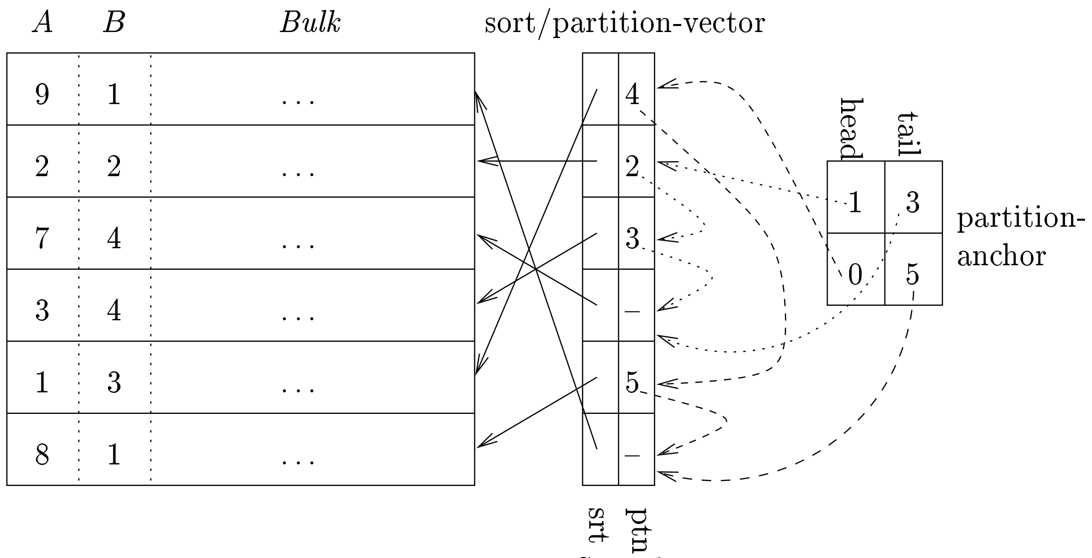


Figure 9: Partitioning a Sorted Run

evaluation plan is the same as for the basic order-preserving hash join plans. Of course, these so-called SOHJ 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 9 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 of the k sorted "partition-lists" (in the example of Figure 9, $k = 2$). 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 Figure 9 shows in fact the generation of the partitions R_{11} and R_{12} for run R_1 of Figure 8.

At this point, we want to take a brief look at how, after all this effort, the performance of our SOHJ plans compares to the performance of traditional hash join plans. Figure 10 shows the SOHJ plan and traditional hash join plan for the example Query 2 of Figure 2: the plan is essentially the same as the plan in Figure 5, but this plan shows significantly more detail. The *buf* operator denotes places where intermediate results are written to disk. The number of partitions consumed or produced are depicted as subscripts of the operators. For example, the $merge_{m*k \rightarrow k}$ merges m partitions at a time, thereby reducing its input of $m * k$ partitions to k partitions. We can see that both plans write and re-read intermediate results five times to disk; the volume written and read is also the same for both plans so that SOHJ plans neither incur additional disk IO nor save any disk IO

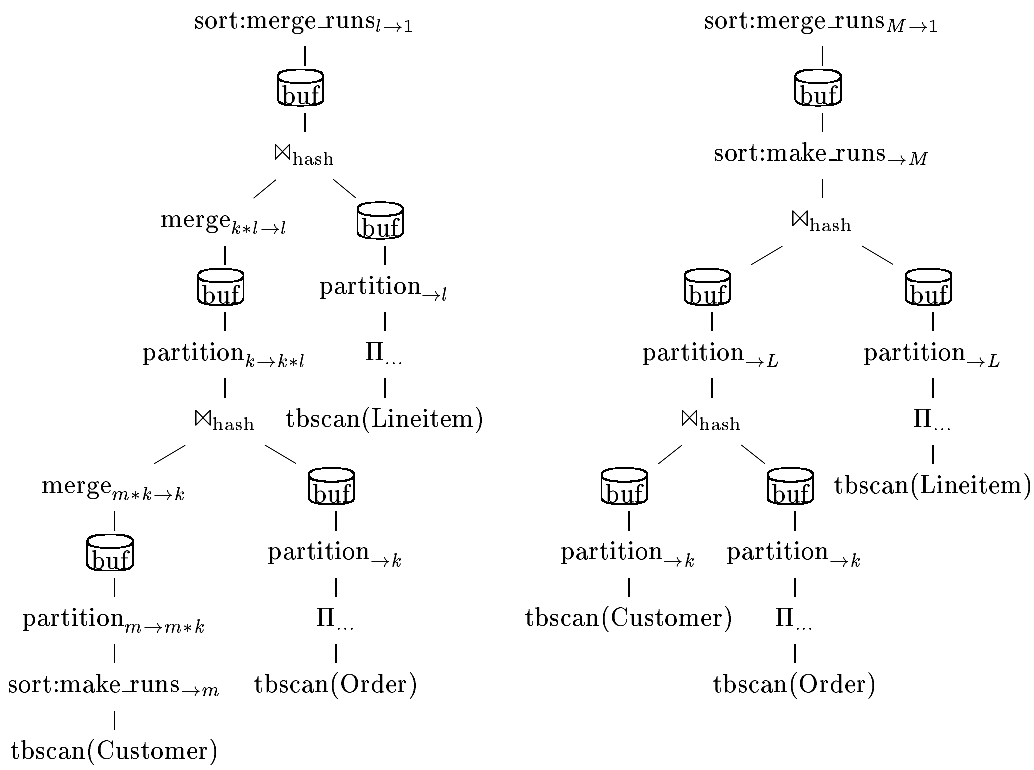


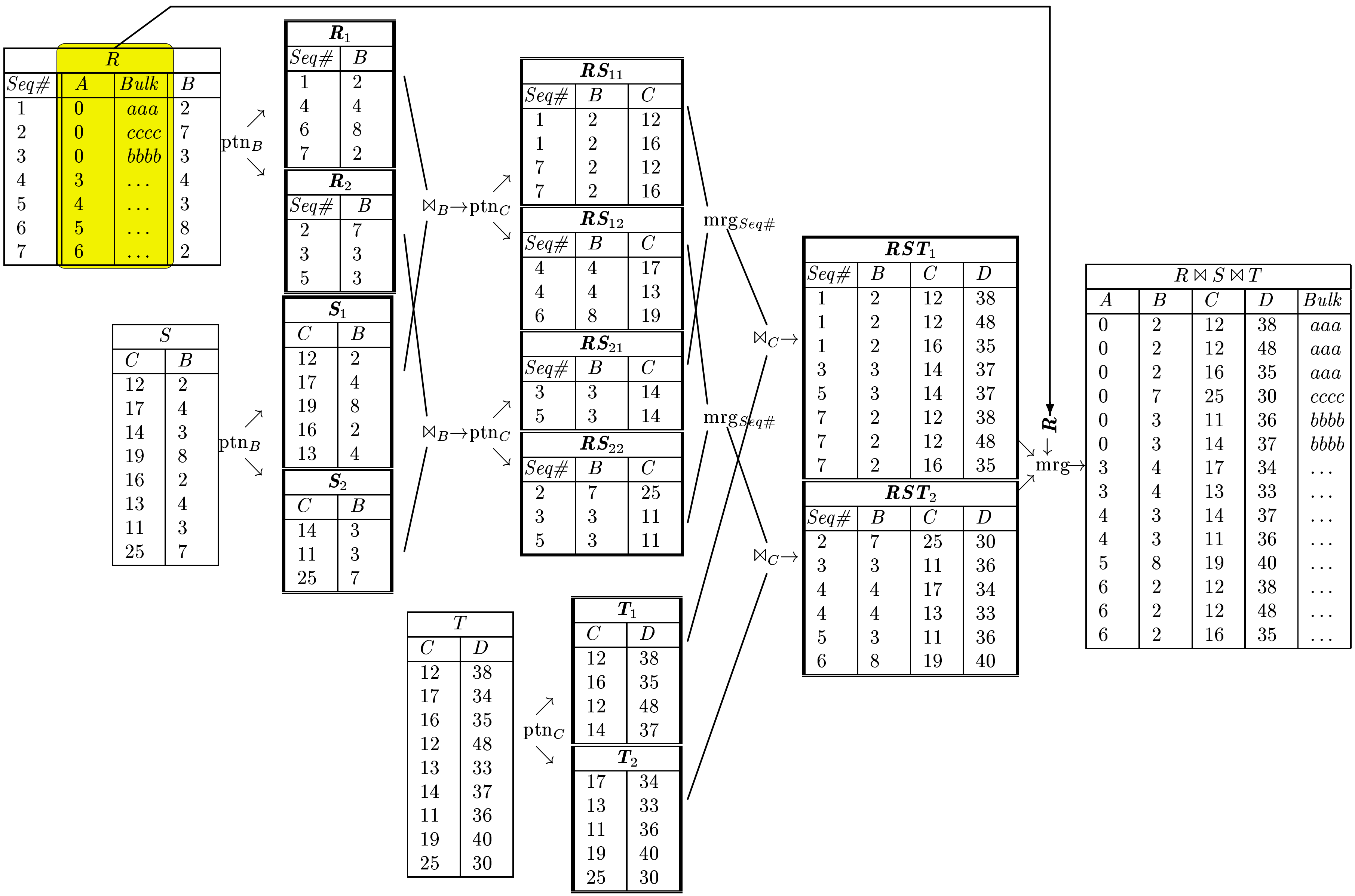
Figure 10: SOHJ and Traditional Hash Join Plan for Query 2

activity. The advantages of SOHJ plans (at this point) stem from reduced CPU costs due to sort-ahead and from cheaper IO. Consider, for example, the last join (i.e., the join with T) in the traditional hash join plan; the probe phase of this join must share the available main-memory buffers with the *make_runs* phase of the succeeding *sort* operator. In contrast, the (initial) *make_runs* phase and the probe phases of all joins can use (almost) all main-memory buffers (modulo IO buffers used to write and merge partitions) in the SOHJ plan. As a result, l will be significantly smaller than L and m will be significantly smaller than M and the partitioning and merging operations will, overall, be (slightly) cheaper in the SOHJ plan than in the traditional hash join plan due to less random IO.

3.4 Bypassing R's Bulk Data Around Joins

We will now show how the performance of OHJ and SOHJ plans can be improved by bypassing R's bulk data around the join. The idea is to project out R's *Bulk* attributes, only keep the columns that are necessary for the join(s) (and maybe sort) and *re-merge* the *Bulk* attributes at the end in order to get the right query result. This way, a great deal of IO can be saved during join processing. A similar approach is known as TID join processing [MR94]. (Note, we use *vertical* merges for *Bulk* re-merging; that is, we merge columns of the same tuples together. This is in contrast to *horizontal* merging which is used for, say, sorting throughout this paper.)

If R is stored in $R.A$ order on disk (i.e., the OHJ plan case), this trick can be applied in a straightforward way: only attribute B which is needed to compute the join and a sequence number *Seq#* which is needed for re-merging the *Bulk* data are retained of R after the initial index scan of R. The join is then carried out (using attribute B) and the *Bulk* data is then re-merged using the sequence number. This approach is shown for a three-way join query in Figure 11 – we will call it OHJ+BB in the remainder of the paper.



R

Seq#	*A*	*Bulk*	*B*
1	0	*aaa*	2
2	0	*cccc*	7
3	0	*bbbb*	3
4	3	...	4
5	4	...	3
6	5	...	8
7	6	...	2

S

C	*B*
12	2
17	4
14	3
19	8
16	2
13	4
11	3
25	7

T

C	*D*
12	38
17	34
16	35
12	48
13	33
14	37
11	36
19	40
25	30

R_1

Seq#	*B*
1	2
4	4
6	8
7	2

R_2

Seq#	*B*
2	7
3	3
5	3

S_1

C	*B*
12	2
17	4
19	8
16	2
13	4

S_2

C	*B*
14	3
11	3
25	7

T_1

C	*D*
12	38
16	35
12	48
14	37

T_2

C	*D*
17	34
13	33
11	36
19	40
25	30

RS_{11}

Seq#	*B*	*C*
1	2	12
1	2	16
7	2	12
7	2	16

RS_{12}

Seq#	*B*	*C*
4	4	17
4	4	13
6	8	19

RS_{21}

Seq#	*B*	*C*
3	3	14
5	3	14

RS_{22}

Seq#	*B*	*C*
2	7	25
3	3	11
5	3	11

RST_1

Seq#	*B*	*C*	*D*
1	2	12	38
1	2	12	48
1	2	16	35
3	3	14	37
5	3	14	37
7	2	12	38
7	2	12	48
7	2	16	35

RST_2

Seq#	*B*	*C*	*D*
2	7	25	30
3	3	11	36
4	4	17	34
4	4	13	33
5	3	11	36
6	8	19	40

$R \bowtie S \bowtie T$

A	*B*	*C*	*D*	*Bulk*
0	2	12	38	*aaa*
0	2	12	48	*aaa*
0	2	16	35	*aaa*
0	7	25	30	*cccc*
0	3	11	36	*bbbb*
0	3	14	37	*bbbb*
3	4	17	34	...
3	4	13	33	...
4	3	14	37	...
4	3	11	36	...
5	8	19	40	...
6	2	12	38	...
6	2	12	48	...
6	2	16	35	...

Figure 11: Order-Preserving Three-Way Hash Join With Bypassing Bulk Around the Join Processing

R is scanned in sort order (via a clustered index on $R.A$). Only the *Seq#* and the join-relevant attribute B are propagated. The final merge (*mrg*) orders tuples from RST_1 and RST_2 according to their *Seq#* values and then replaces the *Seq#* column by the *Bulk* and A attributes obtained from *Seq#-th* tuple of R. Thus, during the merge R is scanned again in sort order (via the cluster index).

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 IO; i.e., 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 order-preserving hash joins, however, we can achieve effective bypassing of bulk data with a cheap re-merge, even if R is not pre-sorted. We simply need to adjust the SOHJ plan by the following three points. (This discussion assumes binary $R \bowtie S$ joins, but SOHJ plans for multi-way join queries can be adjusted just as easily):

1. Adjust the Sort&Partition operator as follows: After sorting run R_i in 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 order-preserving hash joins 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 IO because the *Bulk* runs are ordered in the same way as the join result; i.e., according to $R.A$.

This SOHJ+BB plan is illustrated in Figure 12 for a binary join query.

3.5 Early Aggregation

So far, we looked at SOHJ and OHJ 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* is used [BD83, 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

R

A	B	*Bulk*
9	1	...
0	2	*aaa*
7	4	...
3	4	...
0	3	*bbbb*
8	1	...
6	2	...
4	3	...
10	9	...
5	8	...
0	7	*cccc*
11	10	...

R_1

0	2	*aaa*
0	3	*bbbb*
3	4	...
7	4	...
8	1	...
9	1	...

R_2

0	7	*cccc*
4	3	...
5	8	...
6	2	...
10	9	...
11	10	...

R_{11}

A	U	B
0	1.1	2
3	1.3	4
7	1.4	4

R_{12}

0	1.2	3
8	1.5	1
9	1.6	1

R_{21}

5	2.3	8
6	2.4	2
11	2.6	10

R_{22}

0	2.1	7
4	2.2	3
10	2.5	9

$R_{\{1,2\}1}$

A	U	B
0	1.1	2
3	1.3	4
5	2.3	8
6	2.4	2
7	1.4	4
11	2.6	10

$R_{\{1,2\}2}$

A	U	B
0	1.2	3
0	2.1	7
4	2.2	3
8	1.5	1
9	1.6	1
10	2.5	9

R_1_Bulk

U	*Bulk*
1.1	*aaa*
1.2	*bbbb*
1.3	...
1.4	...
1.5	...
1.6	...

R_2_Bulk

U	*Bulk*
2.1	*cccc*
2.2	...
2.3	...
2.4	...
2.5	...
2.6	...

S

C	B
19	1
12	2
18	1
17	4
16	2
13	4
11	3
14	3
20	9
15	8
25	7
21	10

S_1

C	B
12	2
17	4
16	2
13	4
15	8
21	10

S_2

C	B
19	1
18	1
11	3
14	3
20	9
25	7

$R_{\{1,2\}1} \bowtie S_1$

A	U	B	C
0	1.1	2	12
0	1.1	2	16
3	1.3	4	17
3	1.3	4	13
5	2.3	8	15
6	2.4	2	12
6	2.4	2	16
7	1.4	4	17
7	1.4	4	13
11	2.6	10	21

$R_{\{1,2\}2} \bowtie S_2$

A	U	B	C
0	1.2	3	11
0	1.2	3	14
0	2.1	7	25
4	2.2	3	11
4	2.2	3	14
8	1.5	1	19
8	1.5	1	18
9	1.6	1	19
9	1.6	1	18
10	2.5	9	20

$R \bowtie S$

A	B	C	*Bulk*
0	2	12	*aaa*
0	2	16	*aaa*
0	3	11	*bbbb*
0	3	14	*bbbb*
0	7	25	*cccc*
3	4	17	...
3	4	13	...
4	3	11	...
4	3	14	...
5	8	15	...
6	2	12	...
6	2	16	...
7	4	17	...
7	4	13	...
8	1	19	...
8	1	18	...
9	1	19	...
9	1	18	...
10	9	20	...
11	10	21	...

Figure 12: Projecting Bulk Data From the Runs

For every run, the *Bulk* data is written out to a file in sort order (according to attribute A). The *Bulk* records are identified by $U = i.j$ denoting the run number i and the position within the run j. During the final merge the *Bulk* data is re-inserted. By writing out the *Bulk* files in run order we ensure that each *Bulk* file is read sequentially, that is *Bulk* record $i.k$ is re-inserted (possibly several times) before $i.l$ if $k < l$.

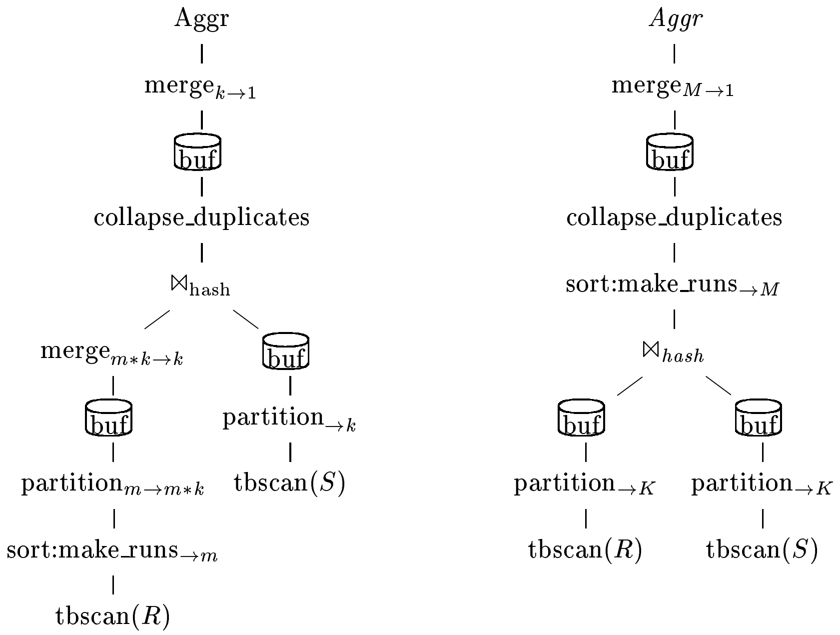


Figure 13: Early Aggregation ($M \gg k$)

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 *cardinality* of each collapsed subgroup.

The order-preserving hash join 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 tuples with the same value of the grouping attributes into a single tuple *before* writing the join results to disk. This way, OHJ and SOHJ plans save a great deal of disk IO. As an example, look at Figure 7; using early aggregation, only three tuples of RST_1 and four tuples of RST_2 need to be written to disk if $R.A$ is the grouping attribute.

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 13 shows an SOHJ and a traditional hash join plan for that query. Of course, the traditional hash join plan can benefit from early aggregation, too. Here, early aggregation is incorporated into the sorting of the runs: After a run is sorted, all duplicates are collapsed. Unfortunately, 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. To see why, we need to consider the number of partitions/runs produced by the two plans as part of the sort-based aggregation. The SOHJ plan generates k runs where k is determined by the number of partitions generated for the (order-preserving) hash join; that is,

$$k \approx \lceil (\textit{size of}\, \Pi_{B,C}(S))/(\textit{main memory size}) \rceil$$

Note, here, that (almost) all the main-memory buffers can be allocated to the hash join, as no other memory-intensive operations run concurrently.

The traditional hash join plan generates M runs as part of the sort-based aggregation. More precisely,

$$M \approx \lceil (\textit{size of } \Pi_{R.*,S.C}(R \bowtie S))/((\textit{main memory size}) - (\textit{space for join hash table}))\rceil$$

as the available main memory buffers must be shared by the hash join and the *make_run* phase of the sort-based aggregation and the sorting is carried out on the (large) result of the join. Here, we also assume that sorting is carried out using a so-called "load&sort" routine (as in many systems today) rather than replacement selection [Lar97]. From the two equations it should be clear that M is significantly larger than k, and therefore, more duplicates will be collapsed in the SOHJ plan as a result of early aggregation: Assuming a uniform distribution of duplicates, the number of duplicates that can be collapsed is in inverse proportion to the number of runs generated. Thus, sort-based aggregation with early aggregation is particularly attractive using our SOHJ (and OHJ) techniques.

3.6 Wrap Up

We described order-preserving hash joins for simple and multi-way join queries as well as for cases in which clustered indices are and are not available. The early sorting and the multi-way OHJ joins are based on fine-grained partitioning. One might suspect that this fine-grained partitioning is especially vulnerable to partitioning skew: Fortunately, this is not the case because we are not dependent on particular hash partitioning functions. We use just the same partitioning functions as a traditional Grace hash would use. In case of correlations between two successive partitionings we may generate corresponding fine-grained partitions (i.e., runs of the same re-merge) that are different in size. This, however, does not incur a (severe) performance penalty because, after the re-merge we obtain the same partitioning (i.e., partition cardinalities) as the normal Grace hash join.

Furthermore, we described a technique called *bypassing bulk data* which can be used to enhance order-preserving hash join plans (SOHJ and OHJ) and which is typically not attractive to enhance traditional query plans. We also showed how order-preserving hash joins and early aggregation complement each other well for aggregate queries. As another example for which our techniques might be useful, consider a four-way join $(R \bowtie_{R.B=S.B} S) \bowtie_{R.D=T.D} (T \bowtie_{T.C=P.C} P)$. Here, we could use SOHJ plans for $R \bowtie_B S$ and $T \bowtie_C P$ in order to generate the order for a final merge join. This evaluation plan is shown in more detail in Figure 14.

An interesting question is: Can other join methods be made order-preserving. Of course, merge joins are order-preserving; but, they only preserve orders on join attributes. The block nested-loop join method would, in fact, generate ordered runs for an existing ordering of the inner relation. However, looking closer into the details, there does not seem to be an efficient way to make block nested loop joins order-preserving for multi-way join queries or for sorting ahead: Recall that both, the low cost early sorting and the multi-way join order preservation of the S/OHJ plans was achieved by a fine-grained partitioning step which is not applicable when using block nested-loop joins. (An order-preserving variant of the block nested-loop join method might, however, be attractive for the *first* join with a preordered inner relation, followed by *regular* order-preserving hash joins.)

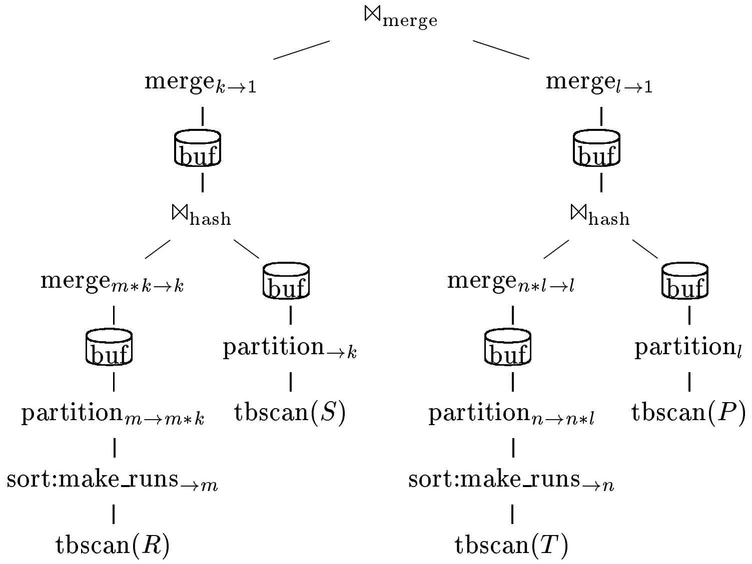


Figure 14: Preparing for the Final Merge Join

4 Quantitative Evaluation

We implemented all operators used in the order-preserving hash join plans in an iterator-based query engine. We used the query engine for an initial performance overview and to validate a cost model that we also used to study the tradeoffs of OHJ and SOHJ plans. We need a cost model in order to facilitate a broader analysis; in particular for experiments with many different database sizes which could not be generated with our query engine with reasonable effort.

4.1 Benchmark Environment

All experiments were performed on a Sun UltraSPARC1/170E system with two internal disks running Solaris 2.6. One disk carries the base relations, the second one is used for temporary segments, e.g., partitions and runs. A constant amount of memory for buffering, sort areas and hash tables is available to the query at any time. The optimal allocation of memory to iterators that run concurrently (i.e., as part of a pipeline in the iterator model) was determined by our optimizer with the help of a detailed cost model.

The cost model, which we also used to generate performance numbers, accurately grasps the major operations of a query engine. I/O costs are modelled according to [HCLS97], i.e., the difference between random and sequential I/O operations and the block transfer size are taken into account. CPU costs and corresponding parameter settings for hash-based operations are taken from [PCV94] and [HR96].

The cardinalities and record sizes of the base tables *Customer*, *Order*, and *Lineitem* (cf. Table 1) are chosen according to the TPC-D benchmark at scale factor 1.0. In some experiments, we varied the size of the *Order* table in order to show how our concepts would scale for large database sizes. We ran the two example queries from Figures 1 and 2 and the group-by query of Section 3.5. In all, we studied five different kinds of plans for these queries: (1) traditional hash join plans – here, we made sure at all times that we used the best join order for every query, memory configuration, and *Order* table size; (2) OHJ plans; (3) SOHJ plans; (4) OHJ+BB plans; and (5) SOHJ+BB plans. We also

table	number of records	avg. record size
Customer	150,000	168
Order	1,500,000	112
Lineitem	6,000,000	126

Table 1: Benchmark Database

studied index-nested-loop join plans such as those of Figure 3, but these plans always had costs of 2000 secs or even more so that we do not show the results here.

4.2 Customer-Order Queries

Figure 15 shows the cost model results for the binary join plans for Query 1 of Figure 1. The traditional hash join plan (HJ) with subsequent sorting shows by far the worst performance due to the expensive sort operation on the entire join result. For small memory configurations, the HJ run 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 in the traditional hash join plan. The S/OHJ plans, on the other hand show good performance even if memory becomes scarce because no two memory-intensive operations run concurrently in those plans. The figure also shows that the bulk bypassing (BB) variants of the OHJ plans yield an additional performance gain due to reduced disk IO to write and re-read intermediate results. Surprisingly, 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, order-preserving hash join plans work well even in the absence of clustered indices and that is the reason why we said "sorting almost for free" in the subtitle of this paper.

Figure 16 shows the running times of the same plans obtained using our query engine. All in all, we observe the same effects. The absolute running times are a little higher with the query engine than predicted by the cost model because the query engine currently has some (unnecessary) I/O overhead to write and read temporary results to/from disk. (We did not want to model this artifact of our query engine in the cost model.) Another potential source of difference between the cost model estimates and the running times of the query engine, which may become apparent for the HJ plans, is that the query engine implements hybrid-hash joins [DKO+84] while the cost model models Grace hash joins; however, we found out in a separate analysis that the differences between hybrid and Grace hash joins were marginal for the queries and main memory sizes studied in all our experiments.

Figure 17 shows the impact of a large *Order* table on the performance of the five plans. (Keep in mind that for large *Order* tables the HJ plans use *Order* as the outer, while *Customer* is always the outer in the S/OHJ plans.) We see that with increasing size of the *Order* table the advantages of our S/OHJ plans increase because the final sort of the traditional HJ plans becomes more and more expensive and dominates the cost of the whole query.

Figure 18 shows the running times of the five plan alternatives for the aggregation query of Section 3.5. 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 profit from collapsing duplicates because

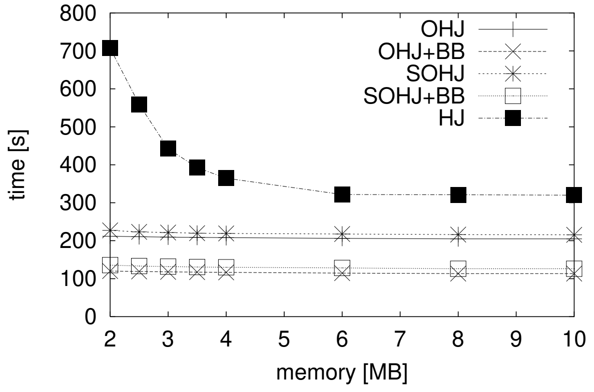


Figure 15: Customer/Order, vary Buffer
150k Customer, 1500k Order, Cost Model

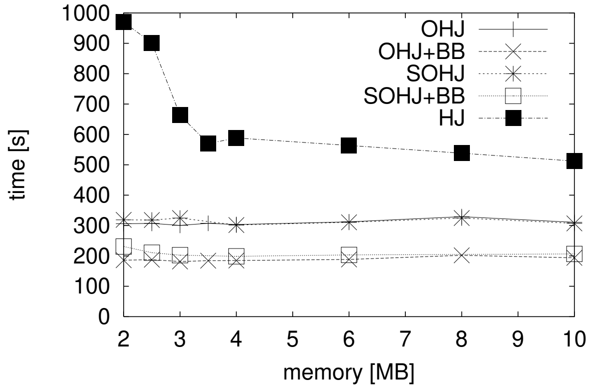


Figure 16: Customer/Order, vary Buffer
150k Customer, 1500k Order, Query Engine

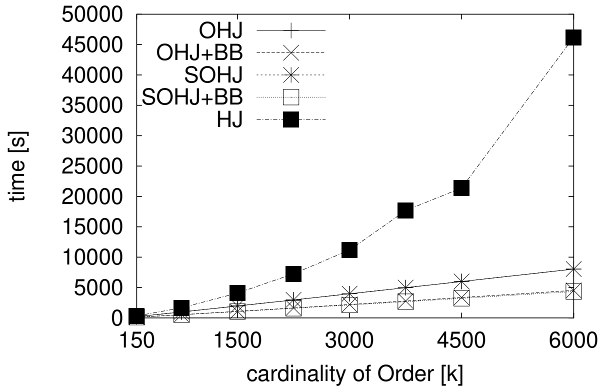


Figure 17: Customer/Order, vary #Order
150k Customer, 10MB Memory

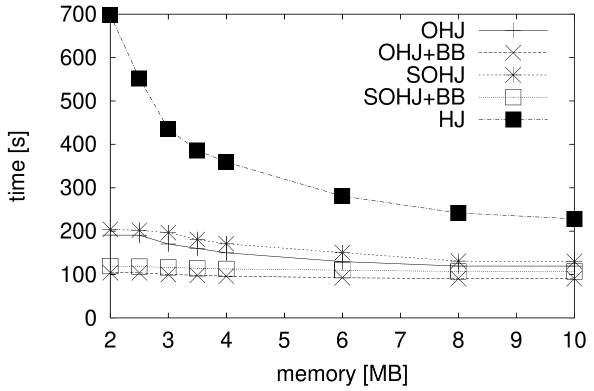


Figure 18: Customer/Order, Early Aggr.
150k Customer/1500k Order

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 this experiment.

4.3 Customer-Order-Lineitem Queries

Figure 19 shows the performance results for the three-way join Query 2 of Figure 2. In addition to the second join operator, the OHJ plans contain the fine-grained partition/remerge step—see Figure 10. Evidently, the performance advantages observed for binary OHJ are retained. Likewise, Figure 20 shows that with larger *Order* and *Lineitem* tables (double and quadruple cardinality), the gap between the HJ plan and the S/OHJ plans becomes bigger; just as for binary join queries (cf. Figure 17).

5 Conclusion

Many join queries—in particular, in OLAP and decision support applications—require ordered results. Very often, the sorting is based on attributes of one small relation (or a few small relations) only. In this case, our new order-preserving hash join evaluation

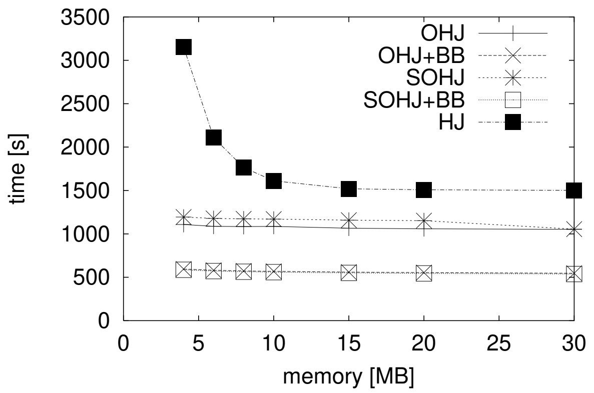


Figure 19: Customer/Order/Lineitem
150k Customer/1500k Order/6000k Lineitem

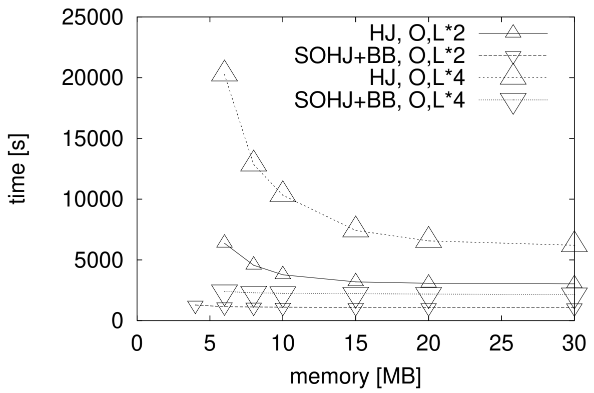


Figure 20: Customer/Order/Lineitem
150k Customer/vary #Order and #Lineitem

allows to exploit an existing ordering of the relation (OHJ) or to push-down the sorting in the evaluation plan (SOHJ). This proves to be a very effective optimization if the join result's cardinality is larger than the sort relation's cardinality—as it is the case in many decision support queries. Further enhancements of the order-preserving hash join plans aim at reducing the size of intermediate results: The bulk bypassing (BB) technique allows to bypass the large attributes of the sort relation around the join processing and the early aggregation is applicable in (frequently encountered) group-by queries. Both enhancements achieved substantial additional performance gains—as our analytical and experimental evaluations showed.

Acknowledgements

We thank Reinhard Braumandl for letting us use his implementation as a basis for our cost model and Christian Wiesner for implementing the SOHJ operators in our query engine.

References

[AAD+96] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 506–521, Bombay, India, September 1996.

[BCK98] R. Braumandl, J. Claussen, and A. Kemper. 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.

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

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

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

[DHKK97] J. Doppelhammer, T. Höppler, A. Kemper, and D. Kossmann. Database performance in the real world: TPC-D and SAP R/3. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 123–134, Tucson, AZ, USA, May 1997.

[DKO⁺84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 1–8, Boston, 1984.

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

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

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

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

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

[KKM98] A. Kemper, D. Kossmann, and F. Matthes. SAP R/3: a database application system. Tutorial handouts for the ACM SIGMOD Conference, Seattle, WA, USA, June 1998. `http://www.db.fmi.uni-passau.de/`.

[KS95] R. Kimball and K. Strehlo. Why decision support fails and how to fix it. *ACM SIGMOD Record*, 24(3):92–97, September 1995.

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

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

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

[PCV94] J. M. Patel, M. J. Carey, and M. K. Vernon. Accurate modeling of the hybrid hash join algorithm. In *Proc. of the ACM SIGMETRICS*, pages 56–66, Santa Clara, CA, May 1994.

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

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

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