DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Efficient Parallel Implementations of the GroupJoin Operator

Aly Kamel

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Efficient Parallel Implementations of the GroupJoin Operator

Effiziente parallele Implementierungen des GroupJoin-Operators

Author:	Aly Kamel
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	Philipp Fent
Submission Date:	15.02.2021

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15.02.2021 Aly Kamel

Abstract

Merging the relational join operator and a grouping together into one operator (GroupJoin) has been proposed for some time. Under a few scenarios, in particular if the grouping is performed on the join attribute, a GroupJoin can provide more efficient computation, which would lead to significant speedups to aggregation queries.

In this paper, efficient implementations of the GroupJoin are provided, both for serial as well as for concurrent execution on parallel architectures.

Contents

1 Introduction

Most aggregation queries require joining multiple relations together to extract meaningful information. In typical databases, the joining and aggregating steps are executed separately in sequence. Usually, each of the operations requires a hash table to be built, meaning two hash tables get constructed and deconstructed.

It is not uncommon that tuples are grouped by the same attributes used in the join:

```
select p.id, p.name, sum(s.profit)
from Products p left outer join Sales s on p.id = s.productId
group by p.id, p.name
```

In those cases, the same hash table can be used for both the joining and the grouping process, sparing the building costs of an extra hash table.

Joins and aggregations are probably the most costly of the traditional relational database operators [25]. Figuring out a way to optimize them will always be a topic of interest. Pipelining aggregations is not possible, since the results are only produced after the entire input has been processed [11], limiting potential parallel optimization techniques. Both operations are typically implemented in one of two ways: hashing or sorting [2, 4, 16, 19]. Building the hash table or sorting the inputs are usually the bottlenecks of those implementations, respectively. Combining the join and the group-by operators removes a repetition of the most expensive part. Depending on the input, it has been suggested to reorder hashing and grouping to minimize the size of the intermediate result built [31], leading to better performance. A GroupJoin can achieve the same effect without increasing the amount of work on the query optimizer by making use of the intermediate results immediately.

This idea to unite the join and group-by operators is not new, as it has been suggested throughout the years under various names. The concept dates back to the 1980s as a way to simplify aggregation queries. The procedure was called *aggregate join* [20] and *outer aggregation* [7]. Steenhagen et al. proposed the *nest join* as a way to optimize nested queries and provide correct aggregation results [30]. Chatziantoniou et al. used the GroupJoin operator under the name *MD-join* to evaluate complex OLAP queries containing for instance cube-by expressions [9]. May and Moerkotte outlined several strategies for serial GroupJoins, which they referred to as *binary groupings* [22].

Moerkotte and Neumann focused on algebraic equivalences that make it possible to transform joins followed by a grouping into GroupJoins [24].

In this paper, strategies to execute the GroupJoin operator in-memory are described and compared with the previously proposed ideas. Parallel implementations especially have not been discussed thoroughly before. The source code for all implementations described in this paper is available online [1].

The paper is structured as follows. Section 2 introduces a few definitions, notations and brief explanations of concepts used in this paper, including the definition of the GroupJoin. In Section 3 commonly used techniques to compute joins and aggregations are discussed. Serial GroupJoin strategies are provided in Section 4. Following that, parallel strategies are introduced in Section 5. In Section 6 the different implementations are evaluated based on their performance. Finally, Section 7 concludes the paper.

2 Preliminaries

2.1 Algebra

For the purposes of this paper, a relation is defined as an unordered multiset of tuples. A tuple consists of a *key value*, which is used for the join and the group-by operators, and a *rest value* containing any remaining attributes. Note that through our definition of a relation, the key does not necessarily uniquely identify a tuple.

Keys and rest values can be of almost any arbitrary type. Some basic requirements exist for custom types, e.g. implementations relying on a hash table require keys to be hashable and comparable for equality. (Multi)sets are denoted by $\{.\}$ and the cardinality of a relation by $|.|$. We use $\circ$ to refer to the concatenation of tuples.

For a given expression e that evaluates to a relation, $\sigma_p(e)$ denotes a selection, which extracts all tuples from e that satisfy the predicate p. A θ-join pairs tuples from two relations that satisfy the binary predicate θ. An equi-join refers to a θ-join using the equality predicate. Additionally, the map operator is defined as $\chi_{a:f}(e) := \{x \circ [a : f(x)] | x \in e\}$, where all rows x are extended by a new attribute a with its value calculated by applying the function f on x. The identity function is denoted by id.

2.2 Predicates

We differentiate between three types of comparison predicates used by the join operator: Equality (=), non-equality ($\neq$) as its negation, and inequalities ($<$, $\leq$, $>$, $\geq$). Inequalities are represented solely by the <-predicate throughout this paper to avoid repetition, since all logic presented can be easily adapted to suit the other predicates. Distinguishing between the predicate types allows for optimized GroupJoin strategies.

In this paper, we focus on predicates containing a single clause only. We refer to [22] for handling predicates with more than one clause.

2.3 Aggregate Functions

A *scalar* aggregate function $agg : S \times \cdots \times S \to \varrho$ produces a single *aggregate value* for a given relation [12]. An aggregate function detects the different groups in the input based on a grouping parameter and calculates an aggregate value for each one of them. Aggregate functions are a cornerstone of data analysis, as they enable the extraction of valuable information out of a large set of data. For example, the total revenue a company has earned can be retrieved for each year individually by using the sum aggregate function.

We now highlight a few properties of aggregate functions, that enable efficient computation [10].

Definition 1. A scalar aggregate function is *decomposable* if there exists functions

$$agg' : S \times \cdots \times S \to \tau$$

$$c : \tau \times \cdots \times \tau \to \tau$$

$$g : \tau \to \varrho$$

with

$$agg(X \cup Y) = g(c(agg'(X), agg'(Y)))$$

for $X, Y \subseteq S \times \cdots \times S$.

The *combination function c* is commutative and can be used to merge the results of multiple agg′ applications, so that the aggregation can be computed incrementally. The *finalization function g* performs a final step on the intermediate result to convert it into an aggregate value of the required type. Each aggregate function defines an identity element ϵ for τ which represents the base state of the accumulated value. Combining the result of agg′ with ϵ does not alter the result.

Definition 2. An aggregate function is *reversible* if a *subtract function*

$$h : \tau \times \cdots \times \tau \to \tau$$

exists with

$$agg(X) = g(h(agg'(X \cup Y), agg'(Y))).$$

The sub-functions of the common aggregate functions are provided in Table 2.1. For example, the computation of the average uses a pair containing the sum and the number of elements included in the sum, both initialized to 0. Combining and subtracting subtotals is achieved through simple addition and subtraction, respectively. During the final aggregation step, the acquired sum is divided by the count to obtain the average.

Note that most aggregate functions are self-decomposable and out of the common ones, only min and max are not reversible. Sum, min, max, count and avg all return NULL if they receive the empty set as input.

	ϵ	agg′	finalize	combine	subtract
sum	0	+	id	+	-
min	∞	min	id	min	
max	$-\infty$	max	id	max	
count	0	count	id	+	-
avg	(0,0)	(+, count)	/	(+, +)	(-, -)

Table 2.1: Sub-functions of common aggregate functions

2.4 GroupJoin

Definition 3. The GroupJoin operator is defined as

$$e_1 \bowtie_{a_1 \theta a_2; q:agg} e_2 := \chi_{q:agg(\sigma_{a_1 \theta a_2}(e_2))}(e_1)$$

with θ being a binary predicate and a_1 and a_2 being attributes of expressions e_1 and e_2, respectively [22, 21]. a_1 is also the grouping parameter used for the calculation of the aggregate function.

The GroupJoin can be expressed equivalently by an outerjoin and a grouping as well [24]. Based on the relationships between the attributes, more complex parameters may be used for the join and grouping. Our introductory example can be conveyed with a GroupJoin because a product is uniquely identified by its id, meaning the product name will not influence the grouping procedure.

Extending the GroupJoin to allow the usage of aggregation vectors, so that multiple aggregate functions can be computed simultaneously, has been discussed in [24] and [9].

The left hand side has been called the *base-values* or *grouping* input and the right hand side the *detail* or *aggregation* input [9, 22]. From this point on, we will simply refer to the left join operand as L and to the right operand, similarly, as R, in order to reduce verbosity. With that, the GroupJoin notation can be abbreviated to

$$L \bowtie_{\theta; q:agg} R$$

with the join parameters being the key values of L and R as described in Subsection 2.1. Figure 2.1 contains example relations, which are used to demonstrate the evaluation of a GroupJoin in Figure 2.2.

Any of R's attributes may be used for the evaluation of the aggregate function. Note that while only the key value of L is needed for the calculation of the GroupJoin, the remaining attributes of L are still part of the output and therefore cannot be discarded during the computation.

A

key	a
1	4
2	3
1	8
3	2

B

key	b
1	6
2	4
4	1
2	3

Figure 2.1: Example relations

$A \bowtie_{=;c:avg} B$

key	a	$\sigma_{A.key=B.key}(B)$
1	4	{(1,6)}
2	3	{(2,4),(2,3)}
1	8	{(1,6)}
3	2	{}

(a) Result after selection

$A \bowtie_{=;c:avg} B$

key	a	c
1	4	6
2	3	3.5
1	8	6
3	2	NULL

(b) Final result

Figure 2.2: Example of a GroupJoin evaluation

3 Techniques

3.1 Joins and Aggregations

Due to their similarity, GroupJoin implementations are heavily inspired by regular join and aggregation implementations. For this reason, we first delve into the three most common techniques used to compute joins and aggregations [23, 16], before we discuss strategies for the GroupJoin.

3.1.1 Nesting

The most basic approach to manage sets of data in a database is through a nested loop. Nested loop implementations have the benefit of being extremely simple and straight-forward to implement, but they are usually dominated in terms of performance by other strategies. Their major advantage over other implementations is that any type of predicate can be used without any modifications to the algorithm.

To evaluate a join query using a nested loop, one of the inputs is designated as the *inner relation* and the other as the *outer relation*. Tuples in the outer relation are compared with every tuple in the inner relation. Whenever the join predicate is satisfied, the matching tuple pair is added to the result set. The flexibility of nested joins makes them the optimal choice for atypical join predicates [26].

Aggregations iterate over each item in their input and search for a matching group in the output. If a group has been found, the corresponding aggregate value is updated. Otherwise, a new group is initialized with the base value.

3.1.2 Sorting

Sorting-based algorithms are based around the fact that processing tuples can be accelerated if it is known that they are ordered by their key. Despite the availability of efficient sorting methods, the sorting phase still accounts for most of the computation time.

One of the most commonly used joins is the *sort-merge join*. Both inputs are first sorted on their respective join attributes. Unlike the nested approach, both relations are scanned together, reducing the amount of tuples that need to be compared. In total, for inputs of sizes N and M, only $N + M$ comparisons are needed.

The calculation of the aggregate value is similar to the nested strategy, except that finding the equivalent group is achieved in constant time. After sorting the input on the grouping parameter, if a group has already been created, it would be the last entry in the output. As a result, only a single value has to be checked.

3.1.3 Hashing

Hashing strategies utilize the fast lookup of hash tables to reduce the amount of comparisons needed, which is why they are used in a wide range of applications. Algorithms are separated into a *build phase* and a *probe phase*.

Hash joins are based on building the hash table with one input and probing it with tuples from the other input. Since each hash bucket only contains a small subset of the relation, the amount of join candidates that have to be checked is decreased. Good hash functions are able to lower the number of hash collisions leading to better performance. Hash joins are widely believed to be currently superior to sort-merge joins [19, 4]. However, hash joins are limited to the evaluation of equi-joins.

Hash tables provide an easy way to group a relation. Tuples are inserted based on the hash of the grouping parameter or merged with preexisting groups. The end result is a hash table containing all groups.

3.2 GroupJoin

In this section a few of the techniques used in the upcoming implementations are explained.

Entries are added into the hash table *HT* with the *insert* function and are searched out with *lookup*. The symbol $\perp$ is used to represent that a lookup has failed to return a value or an iterator has reached the end of a relation. The *append* function receives a tuple and its associated aggregate value and attaches them to the result.

Repeatedly splitting the input of a decomposable aggregate function allows us to operate on single rows independently. This makes it possible to calculate the aggregate value cumulatively while iterating over a relation by updating an accumulator value [20]. The functions *advance* and *produce* take advantage of that fact. Their semantics can be derived from our previous definitions:

$$advance(v, t) := combine(v, agg'(t))$$

$$produce(t, v) := append(t, finalize(v)).$$

Advance merges a tuple $t \in S$ into the intermediate aggregate result $v \in \tau$. Produce performs the final aggregation step on v before adding the pair to the result set. These specifications reduce the amount of function calls performed and provide an abstraction layer. All of our implementations require the aggregate function to be decomposable and provide advance and finalize functions, as well as an ϵ value, similar to how user-defined aggregate functions are specified in Oracle's Database [13].

The result of a GroupJoin is always of the same size as its left input. That knowledge allows memory to be allocated in advance, avoiding any costly potential reallocations. This applies to the result set as well as to the hash table.

To accommodate for rows with no join partners, the aggregate value can be wrapped in an Optional type. When advance is applied, a flag is set to mark the aggregate value as valid. Invalid results are treated as NULL values.

4 Serial Implementations

4.1 Nesting

NestedGJ

The most straightforward strategy to implement a GroupJoin is through a nested loop. R is fully scanned for each tuple in L to find tuples satisfying the join predicate. Each match causes the aggregate value to be updated. Chatziantoniou et al. use R in the outer loop and L in the inner one instead [9].

The nested GroupJoin makes it possible to use any of the predicate types unlike the next algorithms. Its simplicity comes at the expense of having quadratic time complexity, which limits its usage to small input sizes or queries with a high join selectivity [23].

4.2 Hashing

4.2.1 =-Predicate

HashUEqGJ

In most cases where a GroupJoin is used, the attribute used in the join predicate functionally determines L [24]. In other words, the key value uniquely identifies the tuples. The algorithm presented in this section exploits this fact and is based on ideas described in [22, 24].

At the start, a hash table is populated. Each tuple in L gets mapped to ϵ. Then the hash table is probed with R. Whenever an entry has been found, the aggregate value is advanced. After the probing phase has concluded, a final iteration over the hash table is performed in order to finalize the calculation of the aggregate function.

HashEqGJ

An adjustment to the previous algorithm to account for the general case, where key values are not necessarily unique in L, is described in [22]. During the probing phase, all groups with the same key are located and the aggregate value of each of them is advanced. In the end, tuples of the same group are paired with an identical summary value.

A problem with this implementation is that all duplicate values need to get updated for each join partner found. This can be troublesome for inputs with a high duplicate count. Together with a high enough join selectivity, the performance would degrade heavily due to the repeated traversals required.

GroupLEqGJ

An alternate algorithm that takes duplication into consideration is illustrated in Algorithm 1. Instead of storing entire rows or row pointers in the hash table, only the key value is stored. During the insertion step, duplicate keys are ignored. As a result, each group has a single representative in the hash table. To put it in other words, L is effectively grouped by the key value, which is what this algorithm's name is based on. As a result, tuples in R need to update one value at most. In the last step, each tuple in L has to be paired with the aggregate value associated with its group. These summary values can be easily retrieved from the hash table. We trade the cost of the repeated iteration with the, in comparison, fairly cheap cost of hash table lookups.

GroupREqGJ

A different approach is to switch the relations used to build and probe the hash table, as demonstrated in Algorithm 2. In this scenario, R is grouped instead of L. The aggregate value of each group is calculated during the build phase. Either the aggregate value of an existing group is updated or a new entry is created in the hash table and initialized with the appropriate value. The lookup and insertion steps can be combined together, so that the hash table is accessed once per row. L is then used to probe the hash table and perform the final aggregation step. Due to R being already grouped, each tuple can find at most one join partner in the hash table.

Noteworthy is that since the aggregate value calculation takes fully place during the first phase, tuples from L can be appended to the result right after probing, eliminating the need for an additional iteration. In fact, this approach requires only one iteration of R and L each and $|R|+|L|$ hash table accesses. In comparison, GroupLEqGJ requires two iterations over L, one iteration over R and $|L|+|R|+|L|$ hash table accesses.

Algorithm 1: GroupLEqGJ

```
Build:
foreach $T \in L$ do
    if $HT.lookup(T) = \perp$ then
        HT.insert(T.key, $\epsilon$);
    end
end
Probe:
foreach $T \in R$ do
    $val \leftarrow HT.lookup(T)$;
    if $val \neq \perp$ then
        advance(val, T);
    end
end
Output:
foreach $T \in L$ do
    $val \leftarrow HT.lookup(T)$;
    produce(T, val);
end
```

Algorithm 2: GroupREqGJ

```
Build:
foreach $T \in R$ do
    $val \leftarrow HT.lookup(T)$;
    if $val = \perp$ then
        $val \leftarrow \epsilon$;
        HT.insert(T.key, val);
    end
    advance(val, T);
end
Probe/Output:
foreach $T \in L$ do
    $val \leftarrow HT.lookup(T)$;
    if $val = \perp$ then
        $val \leftarrow \epsilon$;
    end
    produce(T, val);
end
```

GroupLREqGJ

To reduce the space cost, hash joins ideally build the hash table on the relation with the fewer distinct values, which typically is the smaller one [23, 6]. A smaller hash table also means the cache can be utilized more effectively. Since the input sizes can be quickly checked at runtime, we introduce an algorithm GroupLREqGJ that chooses one of the previous two algorithms to run depending on the input sizes. According to our testing in Section 6, GroupLEqGJ is the dominant strategy if L is at least 10 times smaller than R. Unlike other optimizations based on the given inputs, determining the smaller relation would barely impact the performance.

Algorithm 3: GroupLUneqGJ

```
Probe:
$total \leftarrow \epsilon$;
foreach $T \in R$ do
    advance(total, T);
    $val \leftarrow HT.lookup(T)$;
    if $val \neq \perp$ then
        advance(val, T);
    end
end
Output:
foreach $T \in L$ do
    $val \leftarrow HT.lookup(T)$;
    $val \leftarrow subtract(total, val)$;
    produce(T, val);
end
```

Algorithm 4: MinUneqGJ

```
Find Min:
$min1.x \leftarrow \infty$; $min2.x \leftarrow \infty$;
foreach $T \in R$ do
    if $T.x < min1.x$ then
        $min2 \leftarrow min1$;
        $min1 \leftarrow T$;
    else if $T.x < min2.x$ and $T.key \neq min1.key$ then
        $min2 \leftarrow T$;
    end
end
Output:
foreach $T \in L$ do
    if $T.key \neq min1.key$ then
        $val \leftarrow min1.x$;
    else
        $val \leftarrow min2.x$;
    end
    produce(T, val);
end
```

4.2.2 $\neq$-Predicate

GroupJoins that couple tuples with unequal keys require us to slightly adjust our approach. If the aggregate function is reversible, all of the previous implementations can be reused by adding an extra step: Calculating a total summary value [22]. This value is generated by applying the scalar aggregate function on all rows. The aggregate value calculated for each group is exactly the complement of the needed value. By deducting this intermediate value from the total, the final aggregate value of a row can be obtained. For that we use the subtraction function defined in Subsection 2.3.

The pseudo code for GroupLUneqGJ is shown in Algorithm 3. The build phase is identical to the one of GroupLEqGJ and has therefore been omitted. We can similarly define GroupRUneqGJ and GroupUneqGJ analogous to the algorithms in Subsection 4.2.1.

MinUneqGJ

As previously mentioned, min and max are the only functions used in typical aggregation queries that are not reversible. Instead of falling back to the naive nested approach of Subsection 4.1, we discuss a more efficient strategy for min in Algorithm 4, with x being the attribute that the aggregate function minimizes.

Through an iteration over R, the two global minimal elements belonging to two different groups are located. Tuples in the same group as the minimal element are then coupled with the second minimal value. All other tuples have unequal keys and are therefore combined with the minimal value.

Adapting the same approach for max is trivial. It might also be possible to develop similar strategies for custom aggregate functions.

4.2.3 <-Predicate

Even though regular hash joins cannot deal with inequality predicates, GroupJoins can make use of hashing more effectively due to R getting grouped. For inequality joins the key values must have a total order and be comparable, clearly. L is sorted and used to build the hash table. Tuples in R should be matched with tuples in L that have a smaller key value. Instead of updating the aggregate value for all the join partners, the aggregate value is only updated for the partner belonging to the largest group. Since L is sorted, binary search can be employed to find the requested group as fast as possible. In the last step we iterate over L in reverse order. That way, the aggregate value can be calculated cumulatively. A group's final aggregate value is calculated by combining its intermediate aggregate value with that of the preceding groups.

4.3 Sorting

4.3.1 =-Predicate

SortMergeEqGJ

Sorting the inputs and then merging them is an alternative to the hash-based procedure. The only difference to a classic sort-merge join, is that the aggregate value is advanced whenever the join keys match and finalized in the last step. The merging phase is shown in Algorithm 5. The aggregate value of the tuple currently being processed is stored, so that succeeding rows with identical keys are spared the calculation.

MergeEqGJ

The majority of the work of the sort-merge join is spent during the sorting phase [23, 19]. If both inputs are already sorted, the merge join is the obvious strategy of choice. Presorted inputs may stem from a clustered index or a preceding operation that has sorted the input on the join parameter, for example a series of GroupJoins using the same L relation.

4.3.2 $\neq$-Predicate

The same algorithms can be adapted for the $\neq$-predicate. The total aggregate value described in Subsection 4.2.2 however would have to be calculated beforehand in an additional iteration. This reduces the viability of a sorting-based approach.

4.3.3 $<$-Predicate

SortMergeLessGJ

In the first step, the inputs are sorted on the key value in descending order. Similar to the hash-based approach, an accumulator is used to calculate the aggregate values successively. Pseudocode for the implementation is provided in Algorithm 6. Since a tuple has a smaller key than the tuples preceding it, it also matches to the same rows in R. For this reason, the accumulator can simply be advanced once to avoid recalculation.

Algorithm 5: MergeEqGJ

```
RIt ← R.begin();
foreach T ∈ L do
    if T.key ≠ prevKey then
        val ← ϵ;
        while RIt ≠ ⊥ and RIt.key < T.key do
            RIt ← RIt.next();
        end
        while RIt ≠ ⊥ and RIt.key = T.key do
            advance(val, RIt);
            RIt ← RIt.next();
        end
        prevKey ← T.key;
    end
    produce(T, val);
end
```

Algorithm 6: SortMergeLessGJ

```
Sort:
sortDesc(L);
sortDesc(R);

Merge/Output:
val ← ϵ;
RIt ← R.begin();
foreach T ∈ L do
    while RIt ≠ ⊥ and T.key < RIt.key do
        advance(val, RIt);
        RIt ← RIt.next();
    end
    produce(T, val);
end
```

5 Parallel Implementations

One of the most common methods used to parallelize various database operators, including hash and sort-merge joins, has been through partitioning the provided input [3, 5, 11]. By splitting the data into chunks and operating on each of them in parallel, significant speedups can be reached [11]. Strategies that do not benefit from explicit partitioning rely on a shared data structure and have the responsibility of organizing its access to avoid race conditions. While inserting tuples or updating values requires syncing, comparisons only need read-access, so they can be performed fully in parallel. Managing concurrent operations through reader-writer locks is a possibility.

Out of the parallel join concepts, partitioned hash joins are the most suited for GroupJoins. The no-partitioning approach is less feasible for GroupJoins, since the aggregate value of a tuple is calculated gradually and only finalized after all tuples have been processed. This step-wise computation means that the aggregate value of a group can be updated multiple times. As updates require write-access, they would have to be synchronized. While coming up with strategies to minimize the wait duration is possible, more synchronization is needed than for regular joins. No-partitioning approaches for hash joins are already outperformed by partitioned ones [3], which would likely be the case for GroupJoins. For that reason, in this paper we only focus on implementations that make use of partitioning.

5.1 Partitioning

The serial GroupJoin algorithms previously discussed are optimized for small input sizes. We can benefit from that fact by splitting the relations into smaller ones and applying our algorithms to them.

Theorem 1. For $L = \bigcup_{i=0}^{n} L_i$ with all L_i being pairwise disjoint and $R = \bigcup_{i=0}^{n} R_i$, if all tuples in R that match tuples in L_i based on the predicate θ appear in R_i, then

$$L \Join_{\theta;q:agg} R = \bigcup_{i=0}^{n} L_i \Join_{\theta;q:agg} R_i.$$

The theorem enables intra-operator parallelism by forbidding overlaps between the subdivisions of L. Otherwise aggregate values may be calculated incorrectly or the same tuples could be processed multiple times. Based on this equivalence, Chatziantoniou et al. proposed speeding up GroupJoins by partitioning L into ranges and distributing the chunks among processors [9]. If R has a clustering index on the key value, then the individual R partitions can be easily identified. Otherwise, R has to be fully scanned for each L partition.

Our implementation differs in a few aspects. It is agnostic about the underlying database and any existing indexes. To avoid a full relation scan, both L and R are partitioned, similar to existing partitioned join strategies [5]. In the end, each thread is responsible for a disjunct set of keys. A serial GroupJoin is performed on each of those partitions with a single thread inserting into the hash table and calculating the aggregate value. Since each thread operates on a different hash table, the need for synchronization is circumvented.

Partitioning Function

The inputs can be partitioned in any way, as long as tuples with the same key are partitioned to the same group. Chatziantoniou et al. divide the workload into ranges. The overall performance depends on the uniformity of the distribution. In the worst case, a majority of tuples occupy a small number of partitions, so that a few threads end up overloaded with work while others are idle. Any metadata at hand can be used to achieve better distribution.

Tuning

Instead of choosing the amount of partitions based on the number of available processors [9], we let the partition count depend on the size of the inputs and a fixed partition size. By doing so, partitions are smaller and consequently the chance that they fit in-memory increases. Erring on the side of too many partitions helps avoid partitions being overflowed [23, 14].

Balkesen et al. has shown that hardware conscious join implementations are better performing than hardware oblivious ones [3]. For that reason, the partition size can be modified to suit the specifications of the machine. The partition size dictates the amount of chunks the relations should be split to. Optimally, it should be set to be smaller than the cache size, so that hash tables can reside entirely inside the cache. This significantly reduces the amount of cache misses [29], resulting into higher lookup performance.

Since the number of partitions usually exceeds the number of available processors on the machine, pre-assigning a thread to each partition could lead to many context switches, which would impact the performance. Instead, a limited amount of threads are each assigned a partition at start and whenever a thread has completed its work on a partition, another partition is dynamically assigned to it. This also helps balance out the work load, as the partitioning function cannot guarantee equal partition sizes. Threads assigned to smaller partitions can finish early and receive more work instead of remaining idle.

5.1.1 =-Predicate

Instead of splitting the relations into ranges, chunks are obtained through hash partitioning. Hash partitioning has the advantage of being more general and applicable with custom types, as well as not requiring any prior information about the data distribution. In addition, it is faster to compute than a range partitioning function, which would run in $\mathcal{O}(\log n)$ time [27].

After the partitioning has been completed, any of the serial GroupJoin algorithms described previously may be used to calculate the result of the partial GroupJoin GroupLREqGJ in particular is advantageous due to the flexibility it provides.

5.1.2 $\neq$-Predicate

GroupJoins with the $\neq$-predicate and reversible aggregate functions require the total aggregate value in the last step of the calculation, as described in Subsection 4.2.2. It can be obtained during the partitioning of the R in the same way that it is acquired in the serial algorithms. L contains no values relevant for the calculation of the total aggregate value and can therefore be partitioned as usual. During the join phase, the calculation of the total value is skipped and the previously calculated value is used instead.

5.1.3 <-Predicate

Partitioning is more complicated for inequality predicates. Partitions cannot be determined using a hash function like in the previous algorithms, since the range each row falls in is important for the calculation. For that reason, range partitioning is necessary. The goal is to divide the range of L into intervals with roughly the same amount of rows. One of the ways to estimate the partition dividers is through random sampling of the inputs.

The total value also has to be cumulatively calculated, as described in Subsection 4.3.3. For the calculation of the aggregate value of a group, all rows in R with a smaller key value are needed. The rows needed for a group with the key k in partition L_i are limited to

1. all tuples $T \in R_i$ with T.key > k, and
2. all tuples $T \in R_j$ with j > i.

The former statement is valid just like in the serial approach and the later one holds due to our use of range partitioning. The total aggregate value is obtained by applying the aggregate function on each set of rows and then using the predefined combine function to merge the results together.

The summary value of the first set is acquired as usual during the GroupJoin process. The other subtotal is calculated during the partitioning stage, alike the strategy in the previous section. The total value is computed for each partition first. Then the total of each partition is combined with the totals of partitions with larger keys. In the end, the total of partition i summarizes the minimum set of rows needed for calculating the aggregate value of partition $i - 1$. The total used in the GroupJoin is instantiated with this value instead of ϵ.

5.2 Parallel Partitioning

The biggest downside of the previous algorithms is the added cost of partitioning. Even if both inputs are partitioned at the same time and hashes are computed in constant time, the extra serial iteration of the inputs is costly. This leads to the partitioning phase being a bottleneck of our implementations. Since our algorithm is already assumed to be ran on a parallel machine, we can introduce parallelism into the partitioning process itself as well. The partitioning work is distributed evenly among a number of threads by utilizing a shared partition table. Our work is heavily inspired by strategies used for parallel partitioned joins [19, 5, 3].

5.2.1 =-Predicate

The process of the parallel partitioned GroupJoin is demonstrated in Figure 5.1. During the initial run, each thread i builds a histogram hst_i to count the amount of rows belonging to a partition (1). Afterwards, the histograms are combined to prefix sums (2). For threads $1, ..., n$ and partition k, $hst_{i,k}$ represents the number of elements in that partition that got assigned to thread i. Thread i can insert tuples belonging to partition k into the shared table starting at position $pos_{i,k}$. It can be deduced from the histograms as follows:

$$pos_{1,0} := 0$$

$$pos_{1,j} := \sum_{l=1}^{n} hst_{l,j-1}$$

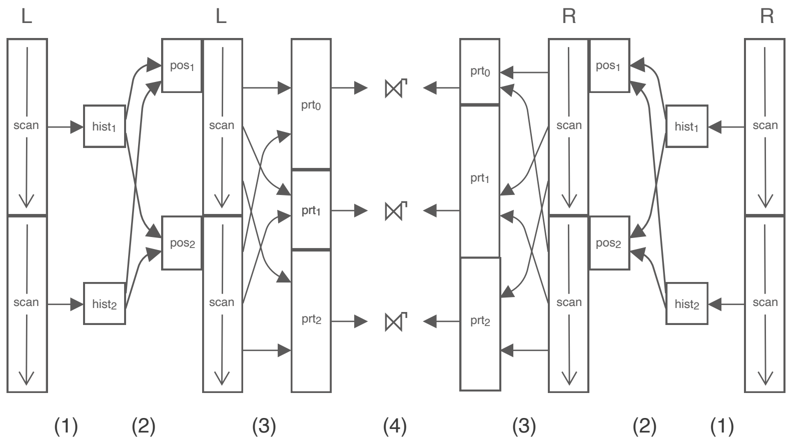


Figure 5.1: Execution of the parallel partitioned GroupJoin

$$pos_{i,j} := pos_{i-1,j} + hst_{i-1,0}$$

Each thread is aware of the space allocated to it so that the partition table can be built in parallel without the need for synchronization (3). After both relations have been partitioned, threads are assigned partitions to perform the GroupJoin on (4). The indexes used for building the partition table are also used to build the result set concurrently.

5.2.2 $\neq$-Predicate

Parallel partitioning allows for a faster calculation of the total aggregate value. Each of the working threads can calculate the total value of the rows it is responsible for. The subtotals are then merged together using the combine function to form the total value. Both operations can be easily integrated into the partitioning process of R.

5.2.3 <-Predicate

The same idea applies for the <-predicate as well. After a worker has determined the partition a tuple belongs to, it also updates the total aggregate value of that partition. The main thread then combines all these subtotals together to form a single total for each partition.

6 Evaluation

In this section, we compare the performance of the previously described algorithms. The code was written in C++11 and compiled with `gcc -O3` on an Intel Xeon with 2.2 GHz. 20 processor cores were used for the execution of our parallel algorithms. All queries were executed five times and the average was plotted.

The C++ Standard Library specifies that hash tables chain collisions [18]. Robin hood hashing minimizes the amount of checks needed before a given key is found [8] and is consistently among the top performing collision handling strategies [28]. Tessil's hash table implementation [15] with linear probing and robin hood hashing showed significant speedups over the standard library implementation, which is why it was the hash table of choice in our algorithms.

Division hashing was used as the hashing function in our tests due to its simplicity and applicability. Algorithms from Intel's TBB library [17] are used to schedule tasks to available threads.

6.1 Datasets

Tuples in the relations used in our tests consist of an integer key and an integer rest value, like in Figure 2.1. Both attributes were picked out of a pool of randomly generated values. By varying the size of the pool, the join selectivity can be controlled. In our test queries, tuples in L had an average of one join partner in R. Unless specified otherwise, both inputs are of the same size, which is notated as *input size* in our figures. The standard integer sum as shown in Table 2.1 is used as the aggregate function.

6.2 Results

GroupLEqGJ vs. GroupREqGJ

The benefit of building the hash table with the smaller relation can be seen in Figure 6.1. As previously theorized, GroupREqGJ is the dominant GroupJoin strategy for equally sized inputs. However, if L is significantly smaller than R, then GroupLEqGJ outperforms GroupREqGJ. Since GroupLREqGJ chooses dynamically which of the algorithms to run, the optimal strategy is always performed with unnoticeable overhead. The

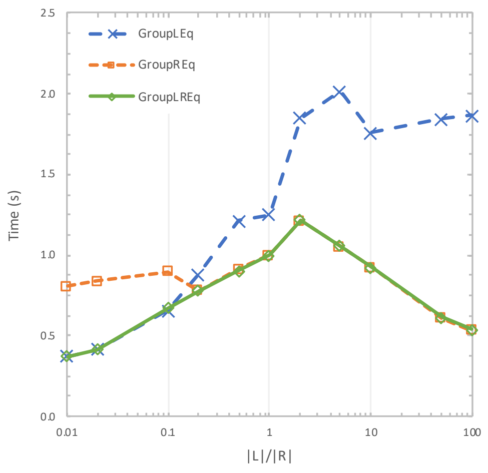
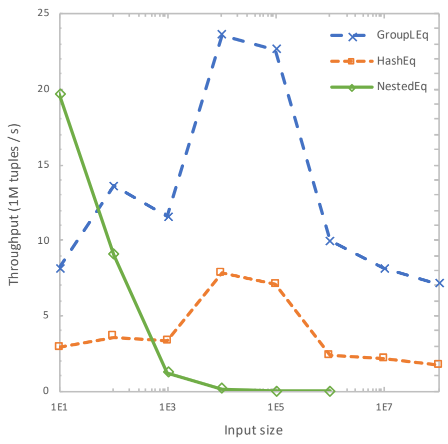


Figure 6.1: Impact of switching the build relation

shift where GroupLEqGJ becomes the superior strategy is when R is more than five times larger than L. Unless one relation is known beforehand to always be significantly smaller than the other, there is no reason to favor GroupLEqGJ or GroupREqGJ over the general GroupLREqGJ.

Hashing-based approaches



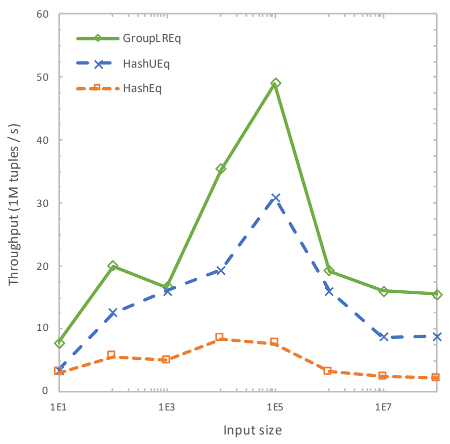
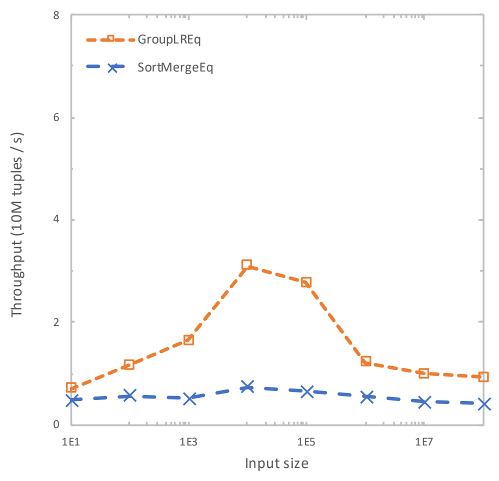
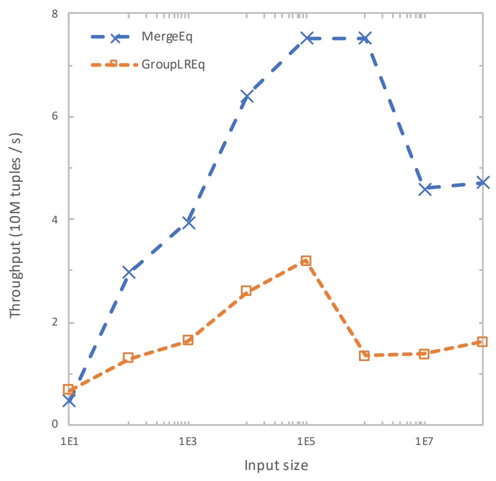


Figure 6.2: Nesting and hashing-based GroupJoins

Figure 6.3: Keys unique in L



(a) Input unsorted



(b) Input presorted

Figure 6.4: Comparison of the sorting-based and the hashing-based approaches

In Subsection 4.2.1, GroupLEqGJ was described as an alternative to the proposed HashEqGJ. Both algorithms rely on building a hash table with L and probing it with R. In Figure 6.2, we compare their performances with each other. It is evident that the nested loop GroupJoin does not compete with the other algorithms. For relations containing more than 1000 rows, the nested approach is extremely inefficient and should not be used. Between the other two approaches, GroupLEqGJ is the clear winner. Its success is owed to how duplicates are dealt with.

Despite HashUEqGJ performing better than HashEqGJ when operating on a duplicate-free left input, GroupLREqGJ still edges out both, as seen in Figure 6.3. Determining if HashUEqGJ can be applied on the given input cannot be achieved at runtime in an efficient manner. Therefore, unless it is guaranteed that the join parameter is unique in L, the old hash-based implementation is not worth consideration.

Sorting vs. Hashing

As expected, the sort-merge GroupJoin is clearly outclassed by the hash GroupJoin, as seen in Figure 6.4a. The high cost of the sorting is exposed once it is separated from the GroupJoin. When sorting is taken out of the equation, the merge algorithm is superior to the hash-based one, more than doubling its throughput, as seen in Figure 6.4b. Noteworthy is that the sortedness of the inputs has no recognizable impact on the performance of GroupLREqGJ.

The takeaway is that the usage of the merge implementation should be limited to presorted relations. This result was anticipated, as the same applies for regular hash joins and sort-merge joins [19]. More modern research shows that sort-merge joins still have potential and are likely to overtake hash joins once the technology is advanced enough [2]. It would be reasonable to assume that GroupJoins will follow the same path.

Serial vs. Parallel Partitioning

For large enough inputs, the divide-and-conquer strategy is beneficial. Even though both partitioning approaches offer an improvement, parallel partitioning performs better, as apparent in Figure 6.5. The speed up gained from dividing the partitioning process among multiple workers outweighs the overhead caused.

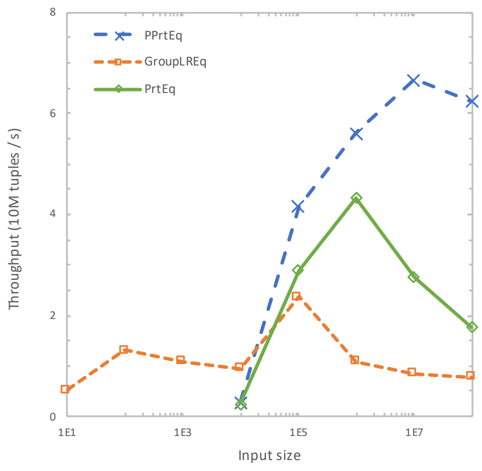


Figure 6.5: =-Predicate

Other predicates

Figure 6.6 and Figure 6.7 compare the performance of our different GroupJoin implementations for the other predicate types. Like with the equi-GroupJoin, our hashing-based algorithm outperforms the sorting-based one. Since tuples need to be ordered during the execution of a GroupJoin with the <-predicate anyway, the sort-based approach performs better.

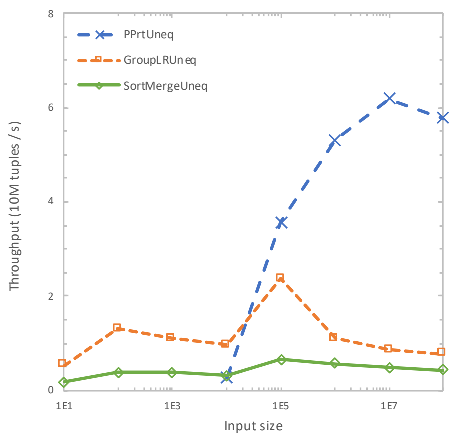


Figure 6.6: $\neq$-Predicate

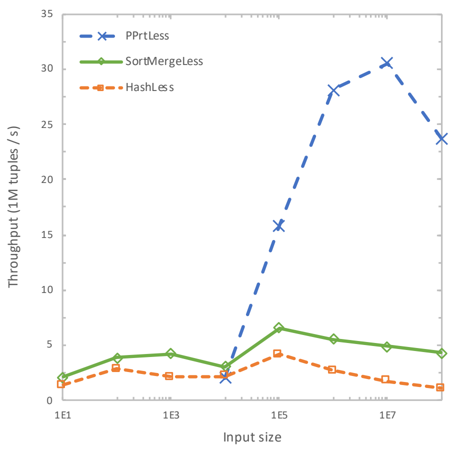


Figure 6.7: <-Predicate

7 Conclusion

As seen in our evaluation, our GroupJoin algorithms achieve a significant speedup compared to the available implementations. For each different scenario there is a different optimized strategy presented.

Nonetheless, there is still a lot of room for improvement. Partitioned hash joins have reached higher speedups than our GroupJoin implementations [3], which shows that our work can still be refined. Our goal was to provide implementations that are as general as possible, oblivious to any preexisting conditions or relationships between relations. No assumptions were made concerning the inputs. However in real world applications, some amount of information is known beforehand. Any information about the key values would open up possibilities for improvement. Input dependant hashing algorithms would provide better performance, as they can reduce hash collisions and distribute rows across partitions more equally.

Moerkotte and Neumann have shown that replacing sequences of joins and groupings with GroupJoins helps accelerate the execution of TPC-H benchmarks [24]. In this paper, we have discussed several strategies on how GroupJoins can be implemented in a database system. Integrating the GroupJoin operator into database systems especially those with heavy aggregation load is definitely advantageous and should be expected in the near future.

Bibliography

[1] https://gitlab.lrz.de/ga53teq/groupjoin.

[2] M.-C. Albutiu, A. Kemper, and T. Neumann. "Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems." In: *Proceedings of the VLDB Endowment* 5 (June 2012). DOI: 10.14778/2336664.2336678.

[3] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware." In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 2013, pp. 362–373. DOI: 10.1109/ICDE.2013.6544839.

[4] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. "Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited." In: 7.1 (Sept. 2013), pp. 85–96. ISSN: 2150-8097. DOI: 10.14778/2732219.2732227.

[5] S. Blanas, Y. Li, and J. Patel. "Design and evaluation of main memory hash join algorithms for multi-core CPUs." In: Jan. 2011, pp. 37–48. DOI: 10.1145/1989323.1989328.

[6] K. Bratbergsengen. "Hashing Methods and Relational Algebra Operations." In: *Proceedings of the 10th International Conference on Very Large Data Bases*. VLDB '84. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1984, pp. 323–333. ISBN: 0934613168.

[7] G. von Bultzingsloewen. "Optimizing SQL Queries for Parallel Execution." In: *SIGMOD Rec.* 18.4 (Dec. 1989), pp. 17–22. ISSN: 0163-5808. DOI: 10.1145/74120.74123.

[8] P. Celis. "Robin Hood Hashing." PhD thesis. CAN, 1986.

[9] D. Chatziantoniou, M. Akinde, T. Johnson, and S. Kim. "The MD-join: an operator for complex OLAP." In: Jan. 2001, pp. 524–533. DOI: 10.1109/ICDE.2001.914866.

[10] S. Cluet and G. Moerkotte. "Efficient Evaluation of Aggregates on Bulk Types." In: Sept. 1995, p. 8. DOI: 10.14236/ewic/DBPL1995.6.

[11] D. DeWitt and J. Gray. "Parallel database systems: The future of high performance database systems." In: *Communications of the ACM* 35.6 (1992), pp. 85–98.

[12] R. Epstein. *Techniques for Processing of Aggregates in Relational Database Systems*. Tech. rep. UCB/ERL M79/8. EECS Department, University of California, Berkeley, Feb. 1979.

[13] J. Gennick. "Build Custom Aggregate Functions." In: *Oracle Magazine* (2006).

[14] R. H. Gerber. "Data-flow query processing using multiprocessor hash-partitioned algorithms." In: (Jan. 1986).

[15] T. Goetghebuer-Planchon. *robin-map*. `https://tessil.github.io/robin-map/`. 2020.

[16] G. Graefe. "Query Evaluation Techniques for Large Databases." In: *ACM Comput. Surv.* 25.2 (June 1993), pp. 73–169. ISSN: 0360-0300. DOI: `10.1145/152610.152611`.

[17] *Intel® Threading Building Blocks*. `https://software.intel.com/content/www/us/en/develop/tools/threading-building-blocks.html`. 2019.

[18] ISO. *Working Draft, Standard for Programming Language C++*. Feb. 2011.

[19] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. "Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs." In: *Proceedings of the VLDB Endowment* 2.2 (2009), pp. 1378–1389.

[20] A. Klug. "Access Paths in the "Abe" Statistical Query Facility." In: *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*. SIGMOD '82. Orlando, Florida: Association for Computing Machinery, 1982, pp. 161–173. ISBN: 0897910737. DOI: `10.1145/582353.582382`.

[21] N. May, S. Helmer, and G. Moerkotte. "Nested queries and quantifiers in an ordered context." In: vol. 20. Feb. 2004, pp. 239–250. ISBN: 0-7695-2065-0. DOI: `10.1109/ICDE.2004.1320001`.

[22] N. May and G. Moerkotte. "Main memory implementations for binary grouping." In: *International XML Database Symposium*. Springer. 2005, pp. 162–176.

[23] P. Mishra and M. H. Eich. "Join processing in relational databases." In: *ACM Computing Surveys (CSUR)* 24.1 (1992), pp. 63–113.

[24] G. Moerkotte and T. Neumann. "Accelerating Queries with Group-By and Join by Groupjoin." In: *PVLDB* 4 (Aug. 2011), pp. 843–851. DOI: `10.14778/3402707.3402723`.

[25] I. Müller, P. Sanders, A. Lacurie, W. Lehner, and F. Färber. "Cache-Efficient Aggregation: Hashing Is Sorting." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 1123–1136. ISBN: 9781450327589. DOI: 10.1145/2723372.2747644.

[26] T. Neumann, V. Leis, and A. Kemper. "The Complete Story of Joins (in HyPer)." In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*. Gesellschaft für Informatik, Bonn, 2017, pp. 31–50.

[27] O. Polychroniou and K. A. Ross. "A Comprehensive Study of Main-Memory Partitioning and Its Application to Large-Scale Comparison- and Radix-Sort." In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 755–766. ISBN: 9781450323765. DOI: 10.1145/2588555.2610522.

[28] S. Richter, V. Alvarez, and J. Dittrich. "A Seven-Dimensional Analysis of Hashing Methods and Its Implications on Query Processing." In: *Proc. VLDB Endow.* 9.3 (Nov. 2015), pp. 96–107. ISSN: 2150-8097. DOI: 10.14778/2850583.2850585.

[29] A. Shatdal, C. Kant, and J. F. Naughton. "Cache Conscious Algorithms for Relational Query Processing." In: *Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 510–521. ISBN: 1558601538.

[30] H. J. Steenhagen, P. M. G. Apers, and H. M. Blanken. "Optimization of nested queries in a complex object model." In: *Advances in Database Technology — EDBT '94*. Ed. by M. Jarke, J. Bubenko, and K. Jeffery. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 337–350. ISBN: 978-3-540-48342-7.

[31] W. P. Yan and P.-Å. Larson. "Eager Aggregation and Lazy Aggregation." In: *Proceedings of the 21th International Conference on Very Large Data Bases*. VLDB '95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 345–357. ISBN: 1558603794.