B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 31

The Complete Story of Joins (in HyPer)

Thomas Neumann[1], Viktor Leis[2], Alfons Kemper[3]

Abstract: SQL has evolved into an (almost) fully orthogonal query language that allows (arbitrarily deeply) nested subqueries in nearly all parts of the query. In order to avoid recursive evaluation strategies which incur unbearable $O(n^2)$ runtime we need an extended relational algebra to translate such subqueries into non-standard join operators. This paper concentrates on the non-standard join operators beyond the classical textbook inner joins, outer joins and (anti) semi joins. Their implementations in HyPer were covered in previous publications which we refer to. In this paper we cover the new join operators *mark-join* and *single-join* at both levels: At the logical level we show the translation and reordering possibilities in order to effectively optimize the resulting query plans. At the physical level we describe hash-based and block-nested loop implementations of these new joins. Based on our database system HyPer, we describe a blue print for the complete query translation and optimization pipeline. The practical need for the advanced join operators is proven by an analysis of the two well known TPC-H and TPC-DS benchmarks which revealed that all variants are actually used in these query sets.

1 Introduction

Joins are arguably the most important relational operators and come in a number of variants. The FROM clause of any SQL query may contain inner joins, as well as left, right, and full outer joins. In addition, many systems support the (anti) semi join operators in order to be able to express (NOT) EXISTS subqueries as joins.

Besides these specific constructs, SQL has become fully orthogonal, i.e., subqueries can occur almost everywhere in a query, including the SELECT, FROM, and WHERE clauses. A query may thus contain an expression which itself contains a subquery and so on. The easiest way to model this is as mutual recursion, i.e., expressions as well as queries can refer to and evaluate each other. Indeed, this is what some systems, for example, PostgreSQL do. The disadvantage of this simple, non-relational approach is that it makes many important optimizations nearly impossible. In effect, it pre-determines the execution plan of common query patterns to nested-loop-style execution with $O(n^2)$ runtime.

We argue that the well-known inner, outer, and semi joins are not enough. To efficiently support full SQL, two additional join types, which in HyPer we call *single join* and *mark join*, are needed. Both variants are introduced in an early stage of the query optimizer in order to translate certain subquery constructs to relational algebra. As a result, the hard-to-optimize mutual recursion of expressions and subqueries is broken up, i.e., expressions do not refer to subqueries any more. Subqueries are translated into re-orderable joins.

[1] Technische Universität München, neumann@in.tum.de
[2] Technische Universität München, leis@in.tum.de
[3] Technische Universität München, kemper@in.tum.de

Our algebra-based, orthogonal approach

- enables additional join-reordering [MN08] opportunities and
- is the foundation of our unnesting [NK15] technique.

This work, therefore, ties together several strands of prior work and gives the full picture of the join optimization pipeline in HyPer. Note that the algorithms of the traditional types have been described in prior work [AKN13, La13, Le14].

The structure of this paper closely mirrors HyPer's query optimizer. After discussing some preliminaries in Section 2, we show how to translate SQL to our extended relational algebra in Section 3. This translation focuses only on correctness, not performance. How this algebra is optimized is described in the following Section 4. Section 5 focuses on the implementation of the non-standard join algorithms.

2 Preliminaries

Before we introduce the non-standard join operators in the next section, let us repeat the definitions of the well-known [Mo14] join variants first.

First, we have the regular *(inner) join*, which is simply defined as cross product followed by a selection:

$$T_1 \bowtie_p T_2 \quad := \quad \sigma_p(T_1 \times T_2).$$

It computes the combination of all matching entries from relation T_1 and relation T_2 using predicate p. It is used in most SQL queries, but its definition is not sufficient in the presence of correlated subqueries. The subquery has to be evaluated for every tuple of the outer query, therefore we define the *dependent join* as

$$T_1 \Join\!\!\!\blacktriangleright_p T_2 \quad := \quad \{t_1 \circ t_2 | t_1 \in T_1 \land t_2 \in T_2(t_1) \land p(t_1 \circ t_2)\}.$$

Here, the right hand side is evaluated for every tuple of the left hand side. (By convention, only the right hand side may depend not the left hand side.) We denote the attributes produced by an expression T by $\mathcal{A}(T)$, and free variables occurring in an expression T by $\mathcal{F}(T)$. To evaluate the dependent join, $\mathcal{F}(T_2) \subseteq \mathcal{A}(T_1)$ must hold, i.e., the attributes required by T_2 must be produced by T_1. The dependent join and its transformation rules form the basis for HyPer's unnesting, which was described in a prior BTW paper [NK15].

Furthermore, we have the semi, anti semi, the left outer, and full outer joins:

$$T_1 \ltimes_p T_2 \quad := \quad \{t_1 | t_1 \in T_1 \land \exists t_2 \in T_2 : p(t_1 \circ t_2)\}$$

$$T_1 \triangleright_p T_2 \quad := \quad \{t_1 | t_1 \in T_1 \wedge \nexists t_2 \in T_2 : p(t_1 \circ t_2)\}$$

$$T_1 ⟕_p T_2 \quad := \quad (T_1 \bowtie_p T_2) \cup \{t_1 \circ_{a \in \mathcal{A}(T_2)} (a : null) | t_1 \in (T_1 \triangleright_p T_2)\}$$

$$T_1 ⟗_p T_2 \quad := \quad (T_1 ⟕_p T_2) \cup \{t_2 \circ_{a \in \mathcal{A}(T_1)} (a : null) | t_2 \in (T_2 \triangleright_p T_1)\}$$

All of these join variants also have a corresponding dependent join variant, which is analogous to the dependent inner join.

Besides the join operators, there is *group by* as additional important operator:

$$\Gamma_{A;a:f}(e) \quad := \quad \{x \circ (a : f(y)) | x \in \Pi_A(e) \wedge y = \{z | z \in e \wedge \forall a \in A : x.a = z.a\}\}$$

It groups its input e (i.e., a base relation or a relation computed from another algebra expression) by A, and evaluates one (or more comma separated) aggregation function(s) to compute aggregated attributes. If A is empty, just one aggregation tuple is produced—as in SQL with a missing group by-clause.

3 Translating Complex SQL to Extended Relational Algebra

The standard operators from the previous section are well known, but they are not sufficient to translate all constructs of modern SQL. Note that, unsurprisingly, some of the examples are a bit unusual. This is to be expected because simple SQL queries can indeed be translated using the well known operations from Section 2. But these somewhat unusual queries are valid SQL, too, and the database needs efficient means to evaluate them. We will illustrate our approach by showing example queries and their translation into relational algebra. We use the following schema:

- Professors: {[Name, <u>PersID</u>, Sabbatical, . . .]}
- Courses: {[Title, ECTS, Lecturer references Professors, . . .]}
- Assistants: {[Name, Boss references Professors, JobTitle, . . .]}

Before discussing the complex cases we briefly cover the so called *canonical translation*, i.e., the textbook mapping of SQL into relational algebra. It simply takes the cross product of the input relations, applies the WHERE predicate as filter, and outputs the result. For example

```
select Title, Name
from Courses, Professors
where PersID = Lecturer
```

is translated into

$$\Pi_{Title,Name}(\sigma_{PersID=Lecturer}(Courses \times Professors)).$$

Later optimization stages will then translate this *canonical translation* into a more efficient plan, for example combining a selection and a cross product into a join. Unfortunately this simple textbook translation is not sufficient for real-world queries which we will now demonstrate. Along the way, we will introduce additional join operators that are needed to handle the various constructs efficiently. Note that for readability reasons we will not include the final projection in the subsequent examples but concentrate on the real operators.

3.1 Dependent Join

The most obvious limitation of the canonical translation is that it ignores correlated subqueries. That is, it just takes the input relations (or input subqueries) from the *from* clause, and forms a cross product from them. This is not always possible in reality. Consider, for example, the following query:

```
select Name, Total
from Professors, (select sum(ECTS) as Total
                  from Courses
                  where PersId = Lecturer) C
```

Here, the subquery depends upon the outer join[4] and a cross product is not applicable. Instead, the correlated subquery has to be added using a *dependent join* like this:

$$Professors \Join (\Gamma_{\emptyset,total:sum(ECTS)}(\sigma_{PersId=Lecturer}(Courses))).$$

Of course the query optimizer will try to get rid of the dependent join as quickly as possible, for example using the techniques from [NK15]. But the initial translation step into relation algebra requires a dependent join—using a regular cross product would be incorrect. Some systems use a nested loop join here, without explicitly designating it as *dependent*, but nested loop joins are highly undesirable for performance reasons. It is usually much better to first introduce dependent joins and then convert them into more efficient regular joins using unnesting techniques.

3.2 Single Join

Correlated subqueries are one class of problems during canonical translation, the second class are scalar subqueries. In SQL, subqueries can be used to compute scalar values, as long as they produce exactly one column and at most one row. Consider for example the following query:

[4] Some DBMSs require extra syntax to indicate this correlation, for example `LATERAL` in PostgreSQL, but others like HyPer accept the query as it is.

```
select PersId, p.Name, (select a.Name
                        from Assistants a
                        where a.Boss = p.PersId
                        and JobTitle = 'personal assistant')
from Professors p
```

This query selects the name of the personal assistant for each professor. Remember that we do not want to fall back to mutual recursion, i.e., we do not want to evaluate the subquery for each professor, as that would lead to $O(n^2)$ runtime. Instead, we want to *join* the Professors relation with the subquery, but we have to take the SQL semantics into account: If the subquery produces a single result we use it as scalar value. If the subquery produces no result the scalar value is NULL. And if there is more than one result we have to report an error.

To express this in relational algebra we introduce a new operator, the *single join*. It behaves nearly identical to an outer join, but reports an error if more than one join partner is found:

$$T_1 ⟕^1_p T_2 := \begin{cases} \text{runtime error,} & \text{if } \exists t_1 \in T_1 : (|\{t_1\} ⟕_p T_2| > 1) \\ T_1 ⟕_p T_2, & \text{otherwise} \end{cases}$$

Using this operator, we can translate the scalar subquery into a join:

$$Professors ⟕^1_{\text{true}} \sigma_{PersId=Boss \land JobTitle='personal\ assistant'}(Assistants)$$

And of course the query optimizer will move the (correlated) predicate into the join operator, resulting in

$$Professors ⟕^1_{PersId=Boss} \sigma_{JobTitle='personal\ assistant'}(Assistants).$$

There are both performance and correctness arguments for introducing the single join. On the performance side a hash-based implemented of the single join ideally has a runtime of $O(n)$, which is much better than the $O(n^2)$ runtime of the recursive evaluation. And in general it is not possible to use other join implementations, as they would not report an error if more than one join partner is found. There are a few exceptions. If the subquery is known to produce at most one tuple, e.g., when binding the primary key or single-tuple aggregation, $⟕$ can be used instead of $⟕^1$. But these are optimizations that are introduced later by the query optimizer, the initial translation step always translates scalar subqueries into single joins.

3.3 Mark Join

Another class of unusual join constructs are predicate subqueries that arise from *exists*, *not exists*, *unique*, and quantified comparisons. Consider for example the following query:

```
select *
from Professors
where exists (select *
              from Courses
              where Lecturer = PersId)
      or Sabbatical = true
```

It would be tempting to translate the subquery into a semi join, and indeed this is what most systems would do without the disjunction, but this is not possible here. We have to output a professor even if no course exists that he or she lectures, and therefore cannot use a semi join.

Instead, we introduce the *mark join*, which creates a new attribute to mark a tuple as having join partners or not:

$$T_1 \ltimes_p^{M:m} T_2 \quad := \quad \{t_1 \circ (m : (\exists t_2 \in T_2 : p(t_1 \circ t_2))) | t_1 \in T_1\}$$

Using the mark join, one can translate the query into a relatively normal join query:

$$\sigma_{(m \vee Sabbatical)}(Professors \ltimes_{PersId=Lecturer}^{M:m} Courses)$$

If the marker is used on only conjunctive predicates the query optimizer can usually translate the mark join into semi or anti semi joins. But this is not possible in general, disjunctions, for example, prevent that. Still, the mark join can be evaluated efficiently, usually in $O(n)$ when using hashing, and introducing it is not a problem for the query optimizer.

Note that the semantics of the mark join becomes significantly more subtle than one might think at a first glance when taking NULL values into account. In the following query

```
select Title, ECTS = any (select ECTS from Courses c2
                          where Lecturer = 123) someEqual
from Courses c1
```

which can be translated directly into the following mark join:

$$Courses\ c_1 \ltimes_{c_1.ECTS=c_2.ECTS}^{M:someEqual} \sigma_{c_2.Lecturer=123} Courses\ c_2$$

The result column *someEqual* can have the values TRUE, FALSE, and NULL (i.e., **unknown**). Accordingly, some care is needed to implement the mark join correctly, as we will discuss in Section 5. But the great benefit is that we now translate arbitrary exists/not exists/unique/quantified-comparison queries into a join construct and eventually, after further optimizations, obtain an efficient evaluation strategy.

3.4 Translating SQL Queries

Putting it all together we can now translate arbitrary SQL queries into relational algebra using the following high-level algorithm:

1. translate the *from* clause, from left to right
   a) for each entry produce an operator tree
   b) if there is no correlation combine with the previous tree using $\times$, otherwise use $\bowtie$
   c) the result is a single operator tree
2. translate the *where* clause (if it exists)
   a) for exists/not exists/unique and quantified subqueries add the subquery on top of the current tree using $\ltimes^{M:m}$. Translate the expression itself with m.
   b) for scalar subqueries, introduce $\ltimes^{1}$ and translate the expression with the (single) result column/row.
   c) all other expressions are scalars, translate them directly
   d) the result is added to the top of the current tree using σ
3. translate the *group-by* clause (if it exists)
   a) translate the grouped expressions just like in the *where* clause
   b) the result is added to the top of the current tree using Γ (group-by)
4. translate the *having* clause (if it exists)
   a) logic is identical to the *where* clause
5. translate the *select* clause
   a) translate the result expressions just like in the *where* clause
   b) the result is added to the top of the current tree using Π
6. translate the *order by* clause (if it exists)
   a) translate the result expressions just like in the *where* clause
   b) the result is added to the top of the current tree using a *sort* operator

This procedure translates an arbitrary SQL query into relational algebra, without having to fall back to mutual recursion between operators and expressions. The result can then be optimized by the query optimizer leveraging efficient join implementations, if applicable.

4 Optimizations

Figure 1 gives a high-level overview of HyPer's optimizer. Translating the SQL abstract syntax tree (AST) to relational algebra is done by the *semantic analysis* component. In this step only inner, outer, (left) mark and single joins are introduced. All other variants appear during later optimization phases in order to improve performance.

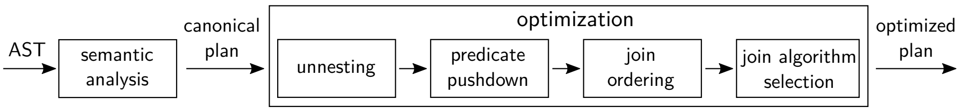


Fig. 1: Overview over optimization process

4.1 Unnesting

One important—and for some queries crucial—optimization is unnesting. For many users, correlated queries are easier to state than semantically equivalent join-based queries. Without unnesting, such queries result in nested-loop joins with $O(n^2)$ runtime. Most systems detect certain correlation patterns and transform them to joins, but are not capable to unnest complex correlations.

Our unnesting technique [NK15], in contrast, is capable of unnesting arbitrary queries—not just some patterns. This is achieved by a set of systematic, algebra-based transformations—as described previously in detail [NK15]. The mark and single join are an implicit building block of our unnesting technique. Without these joins, some queries could not be expressed in relational algebra and, as a result, our unnesting technique would not work.

4.2 Reordering

One of the most important tasks of a query optimizer is to find a good join order since a bad join order can be slower by orders of magnitude [Le15]. As mentioned earlier, in comparison with a mutual recursion-based approach, single and mark joins offer additional reordering opportunities. For example, consider the following query:

```
select *
from Professors p, Assistants a
where p.PersId = a.Boss
      and (exists (select *
                   from Courses c
                   where c.Lecturer = p.PersId)
           or p.Sabbatical = true)
```

Without mark join, the join order is effectively predetermined, and the inner join between *Professors* and *Assistants* is always performed first. Using mark join, on the other hand, it is also possible to start with the mark join before joining with Assistants:

$$(\sigma_{(m \vee Sabbatical)}(Professors \ltimes^{M:m}_{PersId=Lecturer} Courses)) \bowtie_{PersId=Boss} Assistants$$

If there are more Assistants than Professors, performing the mark join first is faster than starting with the inner join. Due to transitivity of the join predicates (*PersId* = *Boss* and *Lecturer* = *PersId*), it is also possible to start with a mark join between Courses and

Assistants. The decision between the three join orders is done by the cost-based join enumeration algorithm. HyPer uses a graph-based dynamic programming algorithm called *DPhyp* that only enumerates connected components without cross products [MN08]. The algorithm takes ordering constraints of non-inner joins into account.

4.3 Left and Right Join Variants

For most join types HyPer has a left (e.g., left mark join) and a right (e.g., right mark join) variant. Both variants semantically produce the same result (with left and right inputs swapped). They differ, however, in their performance.

In hash-based execution, for example, a hash table is built from the left[5] input (the build input). A tuple from the right input will (the probe input) result in a hash table lookup in this hash table. Because hash table insertion is usually slower than lookup, for performance reasons, HyPer's query optimizer swaps the argument order of joins such that the smaller[6] input is on the left. To summarize, having two variants for each join gives the optimizer the freedom that leads to better query performance.

4.4 Other Optimizations

Mark joins are slightly slower than (anti) semi joins because they have to maintain a marker. If possible, mark joins should therefore be translated to (anti) semi joins. In HyPer, the query

```
select *
from Professors
where exists (select *
              from Courses
              where Lecturer = PersId)
```

is first expressed using a mark join, that is then replaced by a semi join ($Professors \ltimes Courses$) in a later optimization step.

Another optimization is to translate outer joins to inner joins. This is possible in the presence of null-rejecting predicates as in the following example:

```
select Title, Name
from Courses right outer join Professors on PersID = Lecturer
where ECTS > 1
```

Finally, a left single join can be replaced by a normal left outer join if the subquery is known to compute at most one row as in the following example:

[5] Note that contrary to our convention, some systems hash the right side.

[6] This is done based on cardinality estimates.

```
select Name, (select sum(ECTS) as Total
              from Courses
              where PersId = Lecturer)
from Professors
```

In this query, a normal outer join is sufficient because the subquery is an aggregate without a group by clause, which always produces a single row.

5 Algorithms

After discussing the various join variants and their optimization, we now come to their actual implementation. Note that we are primarily interested in the high-level algorithm, that is, we do not introduce a particularly tuned implementation, but rather discuss how they differ from regular joins. We therefore describe them in terms of simple main-memory algorithms. The generalization from that to, e.g., external memory algorithms is relatively simple. We will start the discussion with equi-joins, as they are the most common joins and can be implemented efficiently, and then cover non-equi joins.

5.1 Regular Equi-Joins

To highlight the differences between joins, we start with regular hash-based equi joins. Because we describe only the in-memory case here, the code is relatively short, and serves as basis for the different variants. For simplicity we assume that we compute $R \bowtie_{a=b} S$. This can be implemented as follows:

List. 1: Equality Hash Join

```
for each r in R
   store r into H[r.a]
for each s in S
   for each r in H[s.b]
      if r.a = s.b
         emit r,s
```

A real implementation will be much more involved [AKN13, La13, Le14], of course, but this pseudo code helps to illustrate the basic algorithm: A hash table holds all tuples from one side, organized by the join attribute, and the other side probes that hash table to find join partners.

5.2 Joins with Mixed Types

Even the simple equi-join becomes more involved if mixed data types are involved. In our $R \bowtie_{a=b} S$ example, how should we organize the hash table if, e.g., a has the data type `numeric(6,3)` and *b* has the data type `integer`? The internal representation of numbers will be quite different, but still we have to make sure that 3 joins with 3.000, but not with 3.001. This will usually not be the case when using the native hash functions of the different data types.

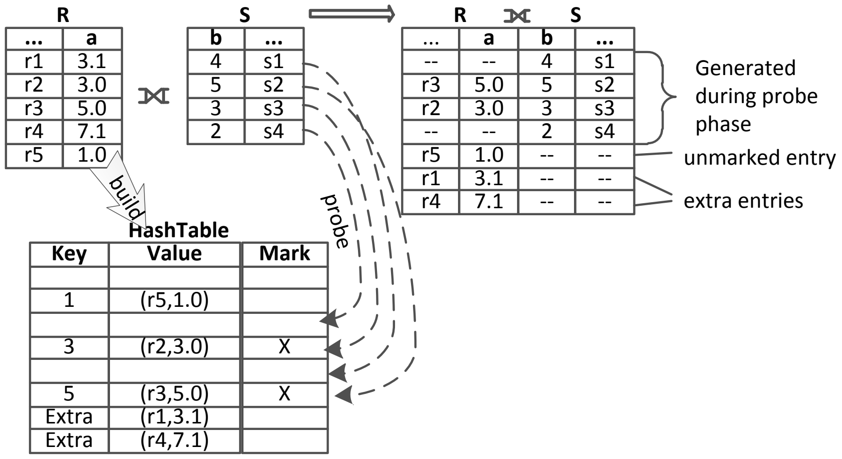
The key insight here is that one should perform the join on the *most restrictive* data type, in our example `integer`. Every value that cannot be represented exactly as an integer will never have a join partner, and thus can be omitted from the hash table. In pseudo code this can be expressed like this (assuming *b* has the most restrictive data type):

List. 2: Equality Hash Join with Mixed Types

```
for each r in R
   a' = r.a cast to the type of S.b
   a'' = a' cast to the type of R.a
   if r.a = a''
      store r into H[a']
for each s in S
   for each r in H[s.b]
      if r.a = s.b
         emit r,s
```

Note that while the pseudo code here assumes that *a* is cast to the type of *b*, it could also be the other way round. For the following algorithms we call the most restrictive data type “compare type” and will shorten that logic to “if cast was exact”.

5.3 Outer Joins



...	a
r1	3.1
r2	3.0
r3	5.0
r4	7.1
r5	1.0

b	...
4	s1
5	s2
3	s3
2	s4

...	a	b	...
--	--	4	s1
r3	5.0	5	s2
r2	3.0	3	s3
--	--	2	s4
r5	1.0	--	--
r1	3.1	--	--
r4	7.1	--	--

Key	Value	Mark
1	(r5,1.0)	
3	(r2,3.0)	X
5	(r3,5.0)	X
Extra	(r1,3.1)	
Extra	(r4,7.1)	

Fig. 2: Outer Join Example

Outer joins output the same tuples as inner joins, and in addition also output all tuples that did not find join partners. This can be implemented by *marking* tuples. As illustrated in Figure 2, each hash table entry has one additional byte, initially 0, that is set to 1 when a join partner is found. This allows us to output all tuples that did not have a join partner after the join is done. For the tuples from the right-hand side we know immediately if we had a join partner or not. We show the pseudo code for the full outer join below, the left and right outer joins are the straightforward subsets of the algorithm.

List. 3: Full Outer Hash Join

```
for each r in R
   a' = r.a cast to compare type
   if cast was exact
      store r into H[a']
   else
      store r into H["extra"]
for each s in S
   sm = 0
   b' = s.b cast to compare type
   if cast was exact
      for each r in H[b']
         if r.a = s.b
            mark r as joined
            sm = 1
            emit r,s
   if sm = 0
      emit null,s
for each r in H
   if r is not marked
      emit r,null
```

The code first processes all R tuples and stores them (unmarked) in the hash table. One extra difficulty here is that some R values potentially cannot be represented in the *compare type* when mixing data types. For inner joins we could drop these, but for left and full outer joins we must produce them anyway, and thus store them in an *extra* hash table bucket that is never selected by the hash function.

Afterwards, we process all S tuples, initialize their local marker (*sm*) to zero, and probe the hash table for potential candidates. When we find one, we mark both the S and the R tuple as having join partners before emitting the joined tuple. After that hash table probe, we know whether the S tuple has a join partner; if not, we emit it after padding it with NULL. After processing the S tuples, we do one sweep over the hash table H and emit all tuples that did not have join partners. This includes the *extra* tuples from data type mismatches.

5.4 (Anti) Semi Joins

This idea of marking tuples can be extended to (anti) semi joins. In semi joins, we output a tuple only if it has not been marked before. In anti semi joins, we emit the tuples afterwards that did not have join partners. These are largely variants of the full outer join code shown above, for space reasons we only show the left semi join:

List. 4: Left Semi Hash Join

```
for each r in R
   a' = r.a cast to compare type
   if cast was exact
      store r into H[a']
for each s in S
   b' = s.b cast to compare type
   if cast was exact
      for each r in H[b']
         if r.a = s.b and r is not marked
            mark r as joined
            emit r
```

The semi join code is not very complex, but some care is needed when handling the marker. If the semi join is executed multi-threaded, only one thread must be allowed to emit a particular *r* value. The easiest way to achieve this is to update the marker using atomic instructions. Anti semi joins mark the tuple, but do not emit it, and instead add a pass at the end to emit unmarked tuples.

5.5 Single Joins

Single joins use the same marking mechanism as left outer joins, and in addition use the marker to detect multiple output values. This is a very attractive way to implement single joins, because in practice this means that the single join is for free, it costs virtually the same as an outer joins. The left single join variant ($⟕^1$) is shown below, it is basically an extension of the left outer join.

List. 5: Left Single Hash Join

```
for each r in R
   a' = r.a cast to compare type
   if cast was exact
      store r into H[a'] else
      store r into H["extra"]
for each s in S
   b' = s.b cast to compare type
   if cast was exact
```

```
      for each r in H[b']
         if r.a = s.b
            if r was marked as joined
               throw an exception
            mark r as joined
            emit r,s
for each r in H
   if r is not marked
      emit r,null
```

Analogously, the right single join is an extension of the right outer join that throws an exception if more than one join partner is found.

5.6 Mark Joins

While the mark join uses a similar marking mechanism as the other algorithms, it is more complex because the marker may not only be TRUE (had a join partner) or FALSE (had no join partner), but also NULL (had a join partner where the comparison result is NULL, but none where the comparison is TRUE). This complicates the code considerably. In cases where the NULL implicitly behaves like a FALSE (e.g., in a disjunctive WHERE clause) the NULL case can be simplified to be identical to the FALSE case, but in the general case extra logic for NULL is needed. The code for the left mark join is shown below:

List. 6: Left Mark Hash Join

```
for each r in R
   if r.a is NULL
      mark r as NULL
      store r into H["null"]
   else
      mark r as FALSE
      a' = r.a cast to compare type
      if cast was exact
         store r into H[a'] else
         store r into H["extra"]
hadNull = false
for each s in S
   if s.b is NULL
      hadNull = true
   else
      b' = s.b cast to compare type
      if cast was exact
         for each r in H[b']
            if r.a = s.b
               mark r as TRUE
```

```
if |S| = 0
   mark all entries in H["null"] as FALSE
for each r in H
   if r is marked FALSE and hadNull
      mark r as NULL
   emit r,marker of r
```

There are multiple subtleties here: First, we now need two extra lists. One list for the values outside the domain of the comparison type, marked as FALSE (i.e., having no join partner). And one list for the tuples where the join attribute is NULL, as all comparisons with this tuple will result in NULL. We do not perform this comparison, but instead statically mark them as NULL and put them in an extra list.

During the join itself, we check for NULL values in the join attribute of S. If we encounter a NULL value, we know that each output tuple will either have the marker TRUE or NULL, but never FALSE (as the NULL value would "join" with all of them). If it is not null, we do the hash table lookup and mark all matching tuples with TRUE.

Afterwards, we scan the hash table and output all tuples with their respective markers. Again there are two special cases: If we did not see any tuples in S at all, the initial NULL marker from the "null" list must be converted into FALSE. And if a tuple is marked as FALSE, but we did see a NULL value in S, the whole tuple is now marked as NULL, as the NULL value would have implicitly joined with it. The code for the right mark join is analogous, except that we now mark the right hand side.

5.7 Non-Equi Joins

While equi-joins are the most common, they are not the only kind of joins. In general, a join predicate can be an arbitrary expression, which cannot always be evaluated using a hash join. Before coming to the general case, let us first discuss "nearly" equality joins, for example joins of the form $R \bowtie_{a=b \wedge c>d} S$. These predicates have an equality component that can be evaluated using hash joins, and doing so is usually a good idea. However, some care must be taken for the non-equal part. For an inner join it is possible to split the predicate, i.e., $R \bowtie_{a=b \wedge c>d} S \equiv \sigma_{c>d}(R \bowtie_{a=b} S)$, which can be answered easily. But for outer joins, for example, this split is not possible, and the additional restrictions must be evaluated directly during the join to avoid incorrect results. For compiling database systems like HyPer [Ne11] this combined evaluation is quite natural, but systems like Vectorwise [ZB12] require extra logic to evaluate arbitrary expressions during a hash join. Note that this non-equality part of the join condition can also return NULL as result, which is relevant for the mark join and results in a NULL marker if the current marker is FALSE.

For arbitrary predicates hash joins are not an option, and the database system needs algorithms for these, too. The best choice is usually a blockwise nested loop join, where chunks of R are loaded into memory and joined with the tuples from S. Note that while this algorithm has the same asymptotic cost as a naive nested loop join, it is much faster in practice, often by orders of magnitude. The pseudo code for the full outer join is shown below, the other join types follow analogously:

List. 7: Full Outer Non-Equality Blockwise Nested Loop Join

```
for each r in R
   if B is full
      call joinBuffer(B)
   store r into B
if B is not empty
   call joinBuffer(B)
for each s in S
   if S is not marked
      emit null,s

function joinBuffer(B):
for each s in S
   for each r in B
      if p(r,s)
         mark r as joined
         mark s as joined
         emit r,s
for each r in B
   if r is not marked
      emit r,null
clear B
```

The join initializes an empty buffer, and then loads as many tuples from *R* into the buffer as possible. When the buffer is full, *joinBuffer* is called, which joins all tuples from *S* with the current buffer content, marks join partners and emits results. After reading *S*, all unmarked tuples from the buffer are emitted padded with NULL, and the buffer is cleared. This continues until *R* has been processed completely. Finally, all unmarked tuples from *S* are emitted padded with NULL.

While this is a different algorithm, the marking uses the same logic as in the equality case. The main question is, how do we implement these markers? Marking the left side is easy, we just use one byte per tuple in the buffer for marking. But marking the right hand side is difficult, as we read the right hand side multiple times and do not materialize it in memory. One could maintain an extra vector for this and spool it to disk as needed, but this is expensive. HyPer instead uses an associative data structure that assigns a bit value to each tuple and that uses interval compression. The idea is that often either very few tuples qualify or nearly all tuples qualify, which both result in small data structures due to the interval compression.

6 Experiments

Evaluating the approach from this paper in a meaningful way is not easy, as the conversion from an approach with mutual recursion into one with specialized joins often transforms an $O(n^2)$ algorithm into an $O(n)$ algorithm. As such, numerical comparisons are largely pointless, because the differences get huge even for modest data sizes. In the following we therefore first show that these kinds of joins occur in widely used benchmarks, and then emphasize that the runtime effects of avoiding mutual recursion can be arbitrarily large.

We compare two systems: HyPer, which implements the techniques and algorithms described in this paper, and PostgreSQL, which implements the mutual recursion strategy. When comparing the query execution times of the two systems it is, however, hard to disentangle the effects (query engine performance, join types, optimizations) using complex benchmark queries. We therefore show some simple example queries and discuss how these queries are evaluated and how well they perform.

	TPC-H		TPC-DS	
	before opt.	after opt.	before opt.	after opt.
inner $\bowtie$	yes	yes	yes	yes
left outer $⟕$	yes	no	yes	yes
right outer $⟖$	no	no	no	yes
full outer $⟗$	no	no	yes	yes
single $⟕^1$	no	no	yes	yes
left mark $⟕^M$	yes	no	yes	yes
right mark $⟖^M$	-	no	-	yes
left semi $\ltimes$	-	yes	-	yes
right semi $\rtimes$	-	yes	-	yes
left anti semi $\triangleright$	-	yes	-	yes
right anti semi $\triangleleft$	-	yes	-	no
group join	-	yes	-	yes

Tab. 1: Join types occurring in TPC-H and TPC-DS before optimization and after optimization. The entries marked as "-" are never introduced by the translation phase (before optimization).

For each join variant, Table 1 shows whether this join occurs in TPC-H and TPC-DS. The table also distinguishes between the plan canonical translation ("before opt.") and the optimized plan ("after opt."). Incidentally, in the TPC queries *all* join types occur, either before or after optimization. There are fewer TPC-H than TPC-DS queries, and they are generally less complex. In TPC-H, single joins do not occur and all the left mark joins can be translated to 4 left/right semi (anti) join variants. In TPC-DS, on the other hand, both single and mark joins are sometimes needed even after optimization. The table also shows that both the right and left variants are chosen by the query optimizer. To summarize, Table 1 indicates that the "zoo" of join variants is indeed needed and a query optimizer will benefit from having all these variants.

Let us now turn to a simple example query on the TPC-H data set (scale factor 1) that illustrates the performance benefit of the single join:

```
select p_name,
       (select l_orderkey
        from lineitem
        where l_partkey = p_partkey
        and l_returnflag = 'R' and l_linestatus = 'O')
from part
```

In HyPer, this query is evaluated in 17 ms (with 1 thread), while PostgreSQL requires 26 hours. The reason for the abysmal performance of PostgreSQL is that it has to perform a full table scan for each tuple of part, which results in quadratic runtime. Note that some of the techniques described in this paper are required in order to evaluate certain queries to avoid mutual recursive with $O(n^2)$ runtime. Therefore, depending on the data set size, our approach can achieve arbitrary speedups.

7 Prior and Related Work

The SQL translation of Microsoft's SQL server was described in [GJ01]. It covers some of the advanced non-standard joins; but unlike our approach it lacks completeness such that not all query templates can be represented in relational algebra. Therefore, SQL server also cannot unnest all query variants as we described in [NK15]. The work by Hölsch et al. [HGS16] uses the nested relational algebra of the NF^2 model for unnesting transformations.

HyPer uses the prior published join ordering [MN08] and simplification [Ne09] for the logical query optimization. In addition to the logical join operators described here, HyPer also exploits synergies between successive operators; in particular it combines joins and grouping using the group join [MN11].

The physical evaluation of HyPer's joins was described in prior publications: Albutiu et. al. [AKN12] developed the massively parallel sort-merge join MPSM. In a later BTW paper [AKN13] this basic join was extended for outer join and semi join variants. Currently, HyPer is mainly based on parallel hash joins whose parallelization was covered in [La13, Le14].

There is a large body of research on optimizing hash joins for the multi-core arena; most notably the works by Jens Teubner's group [Ba15] and by Jens Dittrich's research group [SCD16] which also investigated the effect of different hash functions [RAD15].

8 Conclusions

Based on our HyPer database system we described a blue print for the complete query translation and optimization pipeline for join queries. As SQL has evolved into an (almost) fully orthogonal query language that allows (arbitrarily deeply) nested subqueries in nearly all parts of the query there is a practical need for adavanced join operators to avoid recursive evaluation with an unbearable $O(n^2)$ runtime. The practical need for the advanced join operators was proven by an analysis of the two well known TPC-H and TPC-DS benchmarks which revealed that all join variants discussed in this paper are actually used in these query sets. The paper covered the logical query translation and optimization as well as the physical algorithmic implementation of these join operators. Here, we concentrated on the hash join and block nested loop join variants which are predominantly used in HyPer's current query engine. For practical "hands-on" evaluation the interested readers are invited to experiment with our on-line demo at `hyper-db.de` that provides a graphical visualization of the step-by-step query translation and optimization phases.

References

[AKN12] Albutiu, Martina-Cezara; Kemper, Alfons; Neumann, Thomas: Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. PVLDB, 5(10):1064–1075, 2012.

[AKN13] Albutiu, Martina-Cezara; Kemper, Alfons; Neumann, Thomas: Extending the MPSM Join. In: Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings. pp. 57–71, 2013.

[Ba15] Balkesen, Cagri; Teubner, Jens; Alonso, Gustavo; Özsu, M. Tamer: Main-Memory Hash Joins on Modern Processor Architectures. IEEE Trans. Knowl. Data Eng., 27(7):1754–1766, 2015.

[GJ01] Galindo-Legaria, César A.; Joshi, Milind: Orthogonal Optimization of Subqueries and Aggregation. In (Mehrotra, Sharad; Sellis, Timos K., eds): Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001. ACM, pp. 571–581, 2001.

[HGS16] Hölsch, Jürgen; Grossniklaus, Michael; Scholl, Marc H.: Optimization of Nested Queries using the NF2 Algebra. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016. pp. 1765–1780, 2016.

[La13] Lang, Harald; Leis, Viktor; Albutiu, Martina-Cezara; Neumann, Thomas; Kemper, Alfons: Massively Parallel NUMA-aware Hash Joins. In: Proceedings of the 1st International Workshop on In Memory Data Management and Analytics, IMDM 2013, Riva Del Garda, Italy, August 26, 2013. pp. 1–12, 2013.

[Le14] Leis, Viktor; Boncz, Peter A.; Kemper, Alfons; Neumann, Thomas: Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In (Dyreson, Curtis E.; Li, Feifei; Özsu, M. Tamer, eds): International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014. ACM, pp. 743–754, 2014.

[Le15] Leis, Viktor; Gubichev, Andrey; Mirchev, Atanas; Boncz, Peter A.; Kemper, Alfons; Neumann, Thomas: How Good Are Query Optimizers, Really? PVLDB, 9(3):204–215, 2015.

[MN08] Moerkotte, Guido; Neumann, Thomas: Dynamic programming strikes back. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008. pp. 539–552, 2008.

[MN11] Moerkotte, Guido; Neumann, Thomas: Accelerating Queries with Group-By and Join by Groupjoin. PVLDB, 4(11):843–851, 2011.

[Mo14] Moerkotte, Guido: Building Query Compiler [Draft]. December 8, 2014.

[Ne09] Neumann, Thomas: Query simplification: graceful degradation for join-order optimization. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009. pp. 403–414, 2009.

[Ne11] Neumann, Thomas: Efficiently Compiling Efficient Query Plans for Modern Hardware. PVLDB, 4(9):539–550, 2011.

[NK15] Neumann, Thomas; Kemper, Alfons: Unnesting Arbitrary Queries. In: Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings. pp. 383–402, 2015.

[RAD15] Richter, Stefan; Alvarez, Victor; Dittrich, Jens: A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing. PVLDB, 9(3):96–107, 2015.

[SCD16] Schuh, Stefan; Chen, Xiao; Dittrich, Jens: An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016. pp. 1961–1976, 2016.

[ZB12] Zukowski, Marcin; Boncz, Peter A.: Vectorwise: Beyond Column Stores. IEEE Data Eng. Bull., 35(1):21–27, 2012.