Optimization of Nested Queries using the NF² Algebra

Jürgen Hölsch Michael Grossniklaus Marc H. Scholl

Department of Computer and Information Science, University of Konstanz
P.O. Box 188, 78457 Konstanz, Germany

{juergen.hoelsch,michael.grossniklaus,marc.scholl}@uni-konstanz.de

ABSTRACT

A key promise of SQL is that the optimizer will find the most efficient execution plan, regardless of how the query is formulated. In general, query optimizers of modern database systems are able to keep this promise, with the notable exception of nested queries. While several optimization techniques for nested queries have been proposed, their adoption in practice has been limited. In this paper, we argue that the NF² (non-first normal form) algebra, which was originally designed to process nested tables, is a better approach to nested query optimization as it fulfills two key requirements. First, the NF² algebra can represent all types of nested queries as well as both existing and novel optimization techniques based on its equivalences. Second, performance benefits can be achieved with little changes to existing transformation-based query optimizers as the NF² algebra is an extension of the relational algebra.

1. INTRODUCTION

One of the reasons why declarative languages are popular and successful in query processing is the fact that developers do not need to worry about performance. As long as they can formulate a query that returns the desired result, the query optimizer of the database system promises to find the best possible execution plan. In the case of SQL, modern query optimizers are, for the most part, able to deliver on that promise. Still the exception to this general rule are nested queries, for which the actual formulation can have a large impact on execution time.

In order to close this gap, several optimization techniques for nested queries have been proposed. These techniques either work at the level of SQL [14, 20], define new operators [3, 9] and algebras [4] that are specifically targeted at nested queries, or build on entirely different formalisms, *e.g.*, comprehension calculus [8, 12, 13]. We argue that these approaches all have drawbacks that have limited their adoption in database systems. Using tailor-made algebras and operators requires considerable changes to the query optimizer for a single class of optimizations, whereas introducing an entirely different formalism can even lead to a complete rewrite. Finally, approaches that work at the level of SQL restrict the space of possible optimizations by addressing nested queries in isolation from other optimizations. As an example, consider the following query, which returns all parts that are cheaper and bigger than the average of parts of the same type, according to the schema of the TPC-H benchmark[1], which we use for examples throughout this paper.

```
SELECT P1.p_name
FROM Part P1
WHERE P1.p_retailprice <
        (SELECT AVG(P2.p_retailprice)
         FROM Part P2 WHERE P2.p_type = P1.p_type)
AND P1.p_size >
        (SELECT AVG(P3.p_size)
         FROM Part P3 WHERE P3.p_type = P1.p_type)
```

Both subqueries can be unnested according to Kim [14], which will result in the following SQL query.

```
SELECT P1.p_name
FROM Part P1,
  (SELECT p_type, AVG(p_retailprice) AS avg_price
   FROM Part GROUP BY p_type) AS P2,
  (SELECT p_type, AVG(p_size) AS avg_size
   FROM Part GROUP BY p_type) AS P3
WHERE P1.p_retailprice < avg_price AND P1.p_size > avg_size
   AND P1.p_type = P2.p_type AND P1.p_type = P3.p_type
```

Since both subqueries access the same relation, they can be merged (coalesced) into a single subquery. The result of this transformation is as follows.

```
SELECT P1.p_name
FROM Part P1,
  (SELECT p_type, AVG(p_retailprice) AS avg_price,
                  AVG(p_size) AS avg_size
   FROM Part GROUP BY p_type) AS P2
WHERE P1.p_retailprice < avg_price AND P1.p_size > avg_size
   AND P1.p_type = P2.p_type
```

Even if query optimizers can perform both of these steps in isolation, our experiments (*cf.* Section 7) have shown that they are not able to combine them. Since the inner workings of commercial query optimizers are well-kept company secrets, we can only speculate about the precise reasons for these limitations. We note, however, that both nesting and grouping are concepts of SQL that cannot be represented

[1] http://www.tpc.org/tpch/ (November 18, 2015)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-3531-7/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2882903.2915241

by the relational algebra in its original form [6]. As query plan enumeration typically relies on transformations that are based on equivalences of the relational algebra, it is at least conceivable that optimizations, which cannot be expressed algebraically, are performed in separate phases.

Nesting (in the `WHERE` clause) and grouping are concepts that have been part of SQL since the beginning. Over time, many other extensions were made to the language, which are also not covered by the relational algebra. In this paper, we therefore argue that it is time to re-think the choice of the original relational algebra as a basis for query execution plan enumeration. Rather than defining yet another specialized algebra, we show how an existing algebra, which was developed for the NF^2 (non-first normal form) data model [18], can be used for the uniform optimization of nested SQL queries. Even though it was not their originally intended use, the operators defined by the NF^2 algebra to access nested relations can also be used to represent nested queries.

While the focus of this paper is mainly on nested queries, we note that the NF^2 algebra can also be used to represent SQL concepts such as grouping, `CASE` statements, and window functions with the `PARTITION BY` clause. Furthermore, the NF^2 algebra naturally represents operations provided by object-relational database systems, such as Oracle and PostgreSQL. As a consequence, we argue that the NF^2 algebra is a much better match to the current state of SQL than the original relational algebra. Also, since the NF^2 algebra is an extension of the relational algebra, all existing equivalences remain valid. Finally, all logical operators described in this paper can be implemented by the physical operators that are present in any relational database system, which is a key distinction of our approach with respect to some of the previous proposals. Specifically, the contributions of this paper are as follows.

- We show how all types of nested queries supported by SQL, *i.e.*, nesting in the `SELECT`, `FROM`, and `WHERE` clause, are represented by NF^2 expressions (Section 3).
- We define NF^2 equivalences that formalize existing optimization techniques, which were previously described as edit operations at the SQL code level (Section 4).
- Beyond the current state of the art, we introduce new optimization techniques, which are made possible by the NF^2 approach (Section 5).
- We demonstrate the feasibility of the proposed approach by discussing the necessary changes to a query optimizer that is based on Cascades framework (Section 6).
- We quantify the performance benefits of our approach by measuring the execution times of nested queries in four major database systems and comparing them to execution times of the corresponding queries that were optimized as proposed in this paper (Section 7).

We begin in the next section with a short overview of the NF^2 algebra and the notation used throughout this paper. Related work is presented in Section 8 and concluding remarks as well an an outlook on future work are given in Section 9.

2. NF^2 ALGEBRA OVERVIEW

For the sake of a self-contained presentation, this section introduces the concepts and notation of the NF^2 algebra that are used in this paper. A detailed introduction to the NF^2 algebra can be found, for example, in Schek and Scholl [18]. The NF^2 algebra is an extension of the traditional relational algebra. In the scope of this paper, the extended selection and projection operator, which both support nested subexpressions, are most relevant.

The following NF^2 expression introduces the notation for a selection operator that contains a subexpression in its selection condition.

$$\sigma[\text{c\_custkey} \in \pi[\text{o\_custkey}](\text{Orders})](\text{Customer})$$

For every tuple of the Customer relation, the subexpression in the selection condition is evaluated. If the key of the customer is contained in the result set of the subexpression, the customer tuple is added to the result set of the outer selection operator. The NF^2 expression above is equivalent to the following SQL statement.

```
SELECT *
FROM Customer
WHERE c_custkey IN (SELECT o_custkeyr FROM Orders)
```

The following NF^2 expression introduces the notation for a projection operator that contains a subexpression in its projection list.

$$\pi[\text{c\_custkey}, G := \sigma[\text{o\_custkey} = \text{c\_custkey}](\text{Orders})](\text{Customer})$$

The expression creates a new attribute G that contains all orders of a customer by evaluating the subexpression for each tuple of relation Customer and selecting all order tuples containing the corresponding customer key.

3. NF^2 REPRESENTATION OF NESTED QUERIES

Both generation-based and transformation-based optimizers use algebraic equivalences to enumerate query plans in the search space [16]. As a first step, a SQL statement therefore needs to be transformed into an algebraic expression. In this section, we first show how `GROUP BY`s can be represented in the NF^2 algebra. Then, we discuss how the different types of nested queries in the `WHERE` clause are represented by NF^2 expressions. Finally, we introduce the NF^2 representation for subqueries in the `SELECT` and `FROM` clause.

3.1 Preliminary Definitions

Aggregation functions are an important feature of SQL. However, since they are not part of the original NF^2 algebra, we define them for the purpose of this work.

Definition 3.1.1 (Aggregation Function) *Let f be an aggregation function and $\mathcal{D}$ a domain of numeric values. The input of f is a list of atomic values and the output of f is a single atomic value.*

$$f : \mathcal{P}(\mathcal{D}) \to \mathcal{D}$$

In SQL, aggregation functions are typically used together with the `GROUP BY` clause. Grouping can be represented in the NF^2 algebra using a nested projection.

Definition 3.1.2 (Grouping) *Let R be a relation and $A = \{A_1, \ldots, A_n\} \subseteq attr(R)$ a set of atomic attributes. Further, $R' := \pi[A'_1 := A_1, \ldots, A'_n := A_n, attr(R) \setminus A](R)$ is a relation obtained by renaming the attributes of R. The grouping of R by the attributes A is then defined as follows:*

$$\pi[A, G := \pi[attr(R') \setminus \{A'_1, \ldots, A'_n\}](\sigma[A'_1 = A_1 \wedge \ldots \wedge A'_n = A_n](R'))](R).$$

For each outer tuple t of R, the subset of tuples of R' that have the same values in the attributes $A'_1, \ldots, A'_n$ as t in its attributes $A_1, \ldots, A_n$ is selected. In order to prevent naming conflicts, the attributes of R' are renamed from $A_1, \ldots, A_n$ to $A'_1, \ldots, A'_n$. By applying an aggregation function f on an attribute of G, the groups of G can be aggregated.

Example 3.1.1 *Consider the relation Orders given below.*

o_orderkey	*o_custkey*	*o_totalprice*
1	*5*	*100*
2	*5*	*500*
3	*2*	*30*
4	*2*	*60*

The NF^2 expression to group the relation on the attribute o_custkey is as follows.

$$\pi[o\_custkey, G := \pi[attr(Orders') \setminus \{o\_custkey'\}](\sigma[o\_custkey' = o\_custkey](Orders'))](Orders)$$

The result is shown in the following relation.

o_custkey	*G*	
	o_orderkey'	*o_totalprice'*
2	*3*	*30*
	4	*60*
5	*1*	*100*
	2	*500*

The logical group-by/aggregate operator is defined by combining a nested projection with an aggregation function. This representation enables further transformations on the logical level, *e.g.*, the removal of redundant groupings (*cf.* Section 5). The following definition introduces a dedicated operator to abbreviate the corresponding logical group-by/aggregate NF^2 expression. Note that this operator directly corresponds to the physical group-by/aggregate operator of a relational database system.

Definition 3.1.3 (Group-By/Aggregate Operator) *Let R be a relation, $A = \{A_1, \ldots, A_n\} \subseteq attr(R)$, $\{B_1, \ldots, B_k\} \in attr(R) \setminus A$ a set of attributes, and $F = \{f_1, \ldots, f_k\}$ a set of aggregation functions. The group-by/aggregate operator $\gamma[A; F](R)$ groups R on the attributes of A and computes the functions $f_1, \ldots, f_k$ on the attributes $B_1, \ldots, B_k$:*

$$\gamma[A; F](R) := \pi[A, agg_{f_1} := f_1(\pi[B_1](G)), \ldots, agg_{f_k} := f_k(\pi[B_k](G))](\pi[A, G := \pi[attr(R') \setminus \{A'_1, \ldots, A'_n\}](\sigma[A'_1 = A_1 \wedge \ldots \wedge A'_n = A_n](R'))](R)).$$

As an abbreviation, the set braces in the expression

$$\gamma[\{A_1, \ldots, A_n\}; \{f_1, \ldots, f_k\}](R)$$

are ignored. Instead we write

$$\gamma[A_1, \ldots, A_n; f_1, \ldots, f_k](R).$$

Example 3.1.2 *The expression*

$$\gamma[o\_custkey; MAX(o\_totalprice)](Orders)$$

groups the orders table (cf. Example 3.1.1) on the attribute o_custkey and computes the most expensive order for each customer. The result is shown in the following relation.

o_custkey	agg_{max}
2	*60*
5	*500*

3.2 WHERE Clause Subqueries

Subqueries in the `WHERE` clause can occur in combination with quantifiers such as `ANY`, `ALL`, and `EXISTS`. In this subsection, we present how these different types of nested queries can be represented by NF^2 expressions.

3.2.1 Representation of the IN Operator

The `IN` comparison operator returns true, if the value of the selected column is contained in the result set of the subquery.

Example 3.2.1 *The query below returns all parts delivered by a supplier. This is the case, if the key of a part tuple is contained in the PartSupp relation.*

```
SELECT p_name
FROM Part
WHERE p_partkey IN (SELECT ps_partkey FROM PartSupp)
```

This query is represented by the following NF^2 expression:

$$\pi[p\_name](\sigma[p\_partkey \in \pi[ps\_partkey](PartSupp)](Part)).$$

Definition 3.2.1 *Consider the following SQL statement containing a subquery S in the `WHERE` clause that projects on an attribute, which has the same data type as attribute A.*

`SELECT` L `FROM` R `WHERE` A `IN` (S)

This query is represented by the following NF^2 expression, in which S' is an NF^2 expression that is equivalent to the SQL statement S:

$$\pi[L](\sigma[A \in S'](R)).$$

3.2.2 Representation of the EXISTS Quantifier

The `EXISTS` quantifier holds, if the result set of the corresponding subquery is not empty.

Example 3.2.2 *The query given below is an alternative formulation of the query from Example 3.2.1.*

```
SELECT p_name
FROM Part
WHERE EXISTS (SELECT *
              FROM PartSupp
              WHERE ps_partkey = p_partkey)
```

In the NF^2 algebra, this query is represented by the following expression:

$$\pi[p\_name](\sigma[COUNT(\sigma[ps\_partkey = p\_partkey](PartSupp)) \neq 0](Part)).$$

Definition 3.2.2 *Consider the following SQL statement containing a subquery S in the `WHERE` clause.*

`SELECT` L `FROM` R `WHERE EXISTS` (S)

This query is represented by the following NF² expression, in which S' is an NF² expression that is equivalent to the SQL statement S:

$$\pi[L](\sigma[COUNT(S') \neq 0](R)).$$

Note that for `NOT EXISTS` the predicate COUNT(S') ≠ 0 has to be changed to COUNT(S') = 0. Alternatively, `EXISTS` (or `NOT EXISTS`) could be represented by S' ≠ ∅ (or S' = ∅).

3.2.3 Representation of the ANY Quantifier

The `ANY` quantifier can be used in selection conditions, which compare the value of a given column to the result set of a subquery. The quantifier holds if *at least one* tuple of the subquery result set satisfies the comparison condition.

Example 3.2.3 *The following query returns all parts that are cheaper than the most expensive part.*

```
SELECT p_name
FROM Part
WHERE p_retailprice < ANY (SELECT p_retailprice FROM Part)
```

This query is expressed in the NF² algebra as follows:

$$\pi[p\_name](\sigma[p\_retailprice < MAX(\pi[p\_retailprice'](Part'))](Part)).$$

Definition 3.2.3 *Consider the following SQL statement containing a subquery S in the* `WHERE` *clause that projects on an attribute which has the same data type as attribute A. Let θ ∈ {<, ≤, >, ≥, =} be a comparison operator.*

`SELECT` L `FROM` R `WHERE` A θ `ANY` (S)

Depending on θ, there are three cases that define how the `ANY` *quantifier is represented as an NF² expression. Again, S' is an NF² expression that is equivalent to the SQL statement S.*

1. $\theta \in \{<, \leq\} : \pi[L](\sigma[A\ \theta\ MAX(S')](R))$
2. $\theta \in \{>, \geq\} : \pi[L](\sigma[A\ \theta\ MIN(S')](R))$
3. $\theta \in \{=\} : \pi[L](\sigma[A \in S'](R))$

3.2.4 Representation of the ALL Quantifier

The `ALL` quantifier can be used in selection conditions, which compare the value of a given column with the result set of a subquery. The quantifier holds if *all* tuples of the subquery result set satisfy the corresponding condition.

Example 3.2.4 *The following query returns the most expensive part.*

```
SELECT p_name
FROM Part
WHERE p_retailprice >= ALL (SELECT p_retailprice FROM Part)
```

This query is expressed in the NF² algebra as follows:

$$\pi[p\_name](\sigma[p\_retailprice \geq MAX(\pi[p\_retailprice'](Part'))](Part)).$$

Definition 3.2.4 *Consider the following SQL statement containing a subquery S in the* `WHERE` *clause that projects on an attribute which has the same data type as attribute A. Let θ ∈ {<, ≤, >, ≥} be a comparison operator.*

`SELECT` L `FROM` R `WHERE` A θ `ALL` (S)

Depending on θ, there are two cases that define how the `ALL` *quantifier is represented as an NF² expression. Again, S' is an NF² expression that is equivalent to the SQL statement S.*

1. $\theta \in \{<, \leq\} : \pi[L](\sigma[A\ \theta\ MIN(S')](R))$
2. $\theta \in \{>, \geq\} : \pi[L](\sigma[A\ \theta\ MAX(S')](R))$

3.3 SELECT and FROM Clause Subqueries

Originally, SQL only supported nesting in the `WHERE` clause. In an effort to make the language more orthogonal, subqueries were allowed in the `SELECT` and `FROM` clause as well.

Example 3.3.1 *The given query returns the total number of orders for each customer.*

```
SELECT o_custkey, (SELECT COUNT(*) FROM Orders O2
    WHERE O2.o_custkey = O1.o_custkey) AS nrOrders
FROM Orders O1
```

Note that this SQL statement can be directly represented by the nested projection operator.

$$\pi[o\_custkey, nrOrders := COUNT(\sigma[o\_custkey' = o\_custkey](Orders'))](Orders)$$

Definition 3.3.1 *Consider a SQL statement with a subquery in the SELECT clause.*

`SELECT` L, (S) `AS` Y `FROM` R

The query is represented by the following NF² expression, in which S' is an NF² expression that is equivalent to the SQL statement S:

$$\pi[L, Y := S'](R).$$

A subquery in the *FROM* clause can be directly represented as an input of the corresponding NF² operator.

Definition 3.3.2 *Consider a SQL statement with a subquery S in the* `FROM` *clause.*

`SELECT` L `FROM` S `WHERE` F

The query is represented by the following NF² expression, in which S' is an NF² expression that is equivalent to the SQL statement S:

$$\pi[L](\sigma[F](S')).$$

4. EQUIVALENCES FOR NESTED QUERY OPTIMIZATION TECHNIQUES

In the previous section, we defined the algebraic representation of SQL subqueries in terms of NF² expressions. Based on this representation, we now derive equivalence rules for well-know nested query optimization techniques.

4.1 Unnesting Subqueries

The work of Kim [14] describes how nested queries in the `WHERE` clause can be replaced by joins. The transformations are directly applied on the SQL code of the given query. Kim classifies the `WHERE` clause subqueries into five categories (Type N, J, A, JA, and D). In this paper, we define equivalence rules for the transformations of the Types N, J, A, and JA. Since Type D queries contain the relational division operator, which is no longer supported by the current SQL standard, we do not provide equivalence rules for queries of this type.

4.1.1 Unnesting Type N and Type J Subqueries

A subquery is of Type N, if it occurs in the `WHERE` clause in combination with the `IN` comparison operator and has no aggregation. If the subquery is additionally correlated with the outer query, the subquery is of Type J. The following example illustrates a Type J subquery and its equivalent NF^2 expression.

Example 4.1.1 *The query given below returns all parts delivered by a supplier.*

```
SELECT p_name, s_name
FROM Part, Supplier
WHERE p_partkey IN (SELECT ps_partkey
                    FROM PartSupp
                    WHERE ps_suppkey = s_suppkey)
```

The equivalent NF^2 expression is as follows:

$$\pi[p\_name,\ s\_name](\sigma[p\_partkey \in \pi[ps\_partkey](\sigma[ps\_suppkey = s\_suppkey](PartSupp))](Part \times Supplier)).$$

As described by Kim, this subquery can be replaced by a join. Example 4.1.2 shows the result of the transformation and the equivalent NF^2 expression.

Example 4.1.2

```
SELECT p_name, s_name
FROM Part, Supplier, PartSupp
WHERE p_partkey = ps_partkey AND ps_suppkey = s_suppkey
```

Below, the corresponding expression of the NF^2 algebra is given. Note that this is not the most efficient expression, since there is still a cross product. The cross product can be removed, however, based on equivalences defined by the traditional relational algebra.

$$\pi[p\_name,\ s\_name](PartSupp \bowtie_{p\_partkey=ps\_partkey \wedge ps\_suppkey=s\_suppkey} (Part \times Supplier))$$

In Definition 4.1.1, the equivalence rule for unnesting Type J and N subqueries is introduced.

Definition 4.1.1 (Unnesting Type N/J Subqueries) *Let $Inner$ and $Outer$ be NF^2 expressions, $A \in attr(Outer)$ and $B \in attr(Inner)$ attributes, and F a condition. The equivalence rule for unnesting a Type N or J subquery is defined as follows (for Type N, F is true; for Type J, F is a correlation condition):*

$$\sigma[A \in \pi[B](\sigma[F](Inner))](Outer) \equiv \pi[attr(Outer)](Outer \bowtie_{A=B \wedge F} Inner).$$

4.1.2 Unnesting Type A Subqueries

A subquery with aggregation, but without correlation is of Type A. Example 4.1.3 shows a Type A subquery.

Example 4.1.3 *The following query returns all parts with a price greater than the average price of parts.*

```
SELECT p_name
FROM Part P1
WHERE P1.p_retailprice > (SELECT AVG(P2.p_retailprice)
                          FROM Part P2)
```

The equivalence rule given in Definition 4.1.2 can be used to transform a Type A subquery into an expression, in which the aggregation value of the subquery is computed once. This precomputing is possible because the subquery is independent from the outer query and returns a single value as a result.

Definition 4.1.2 (Unnesting Type A Subqueries) *Let $Inner$ and $Outer$ be NF^2 expressions, $A \in attr(Outer)$ and $B \in attr(Inner)$ attributes, f an aggregation function, and $\theta \in \{<, \leq, >, \geq, =, \neq\}$ a comparison operator. The equivalence rule for unnesting Type A subqueries is defined as follows:*

$$\sigma[A\ \theta\ f(\pi[B](Inner))](Outer) \equiv \pi[attr(Outer)](\sigma[A\ \theta\ agg](Outer \times (agg := f(\pi[B](Inner))))).$$

4.1.3 Unnesting Type JA Queries

A Type JA subquery is correlated with the outer query and uses aggregation. Example 4.1.4 shows a Type JA subquery and its equivalent NF^2 expression.

Example 4.1.4 *The following query returns all parts with a price greater than the average price of parts of the same type.*

```
SELECT P1.p_name
FROM Part P1
WHERE P1.p_retailprice > (SELECT AVG(P2.p_retailprice)
                          FROM Part P2
                          WHERE P2.p_type = P1.p_type)
```

The equivalent NF^2 expression contains a relation Part', which is a renamed relation of Part. The renaming is necessary to unambiguously define to which relation (inner or outer relation) an attribute belongs to.

$$\pi[p\_name](\sigma[p\_retailprice > AVG(\pi[p\_retailprice'](\sigma[p\_type' = p\_type](Part')))](Part))$$

According to Kim, the nested query of Example 4.1.4 can be transformed as follows. The relation of the subquery is grouped by the attribute that is contained in the correlation condition. For each group, the value of the aggregation function is computed. Afterwards, the original subquery in the `WHERE` clause is removed and the grouping is added as a new subquery to the `FROM` clause of the outer query. Example 4.1.5 shows the result of this transformation.

Example 4.1.5

```
SELECT P1.p_name
FROM Part P1, (SELECT p_type, AVG(p_retailprice) AS avg_price
               FROM Part
               GROUP BY p_type) P2
WHERE P2.p_type = P1.p_type
      AND P1.p_retailprice > P2.avg_price
```

This query is represented using the NF^2 algebra as follows:

$$\pi[p\_name](\sigma[p\_retailprice > avg\_price](Part \bowtie_{p\_type'=p\_type} \gamma[p\_type';\ avg\_price := AVG(p\_retailprice')](Part'))).$$

In Definition 4.1.3, the equivalence rules that can be used to unnest Type JA subqueries are introduced. As pointed

out by Ganski and Wong [10], subqueries that use the COUNT aggregation function and/or have inequality join predicates need to be treated separately. These different cases lead to three equivalence rules for Type JA subqueries.

Definition 4.1.3 (Unnesting Type JA Subqueries) *Let Inner and Outer be NF² expressions, $A \in attr(Outer)$ and $B \in attr(Inner)$ attributes, f an aggregation function, and $\theta \in \{<, \leq, >, \geq, =, \neq\}$ be a comparison operator. Assume that F is the condition that creates the correlation between the outer and inner query. Depending on the aggregation function f and the condition F, the equivalence rules for unnesting Type JA subqueries are defined as follows.*

1. *$f \neq COUNT$ and F consists of conjunctive subconditions $A_i = B_j$ with $A_i \in attr(Outer)$, $B_j \in attr(Inner)$. The set G contains the attributes $B_j \in attr(Inner)$.*

$$\sigma[A\ \theta\ f(\pi[B](\sigma[F](Inner)))](Outer)$$
$$\equiv \pi[attr(Outer)](\sigma[A\ \theta\ agg](Outer \bowtie_F \gamma[G; agg := f(B)](Inner)))$$

2. *$f \neq COUNT$ and F consists of conjunctive subconditions $A_i\ \theta'\ B_j$ with $\theta' \in \{<, >, \neq\}$, $A_i \in attr(Outer)$, and $B_j \in attr(Inner)$. The set G contains the attributes $A_i \in attr(Outer)$.*

$$\sigma[A\ \theta\ f(\pi[B](\sigma[F](Inner)))](Outer)$$
$$\equiv \pi[attr(Outer)](\sigma[A\ \theta\ agg](Outer \bowtie \gamma[G; agg := f(B)](Outer \bowtie_F Inner)))$$

3. *$f = COUNT$ and F consists of conjunctive subconditions $A_i\ \theta'\ B_j$ with $\theta' \in \{<, >, =, \neq\}$, $A_i \in attr(Outer)$, and $B_j \in attr(Inner)$. The set G contains the attributes $A_i \in attr(Outer)$.*

$$\sigma[A\ \theta\ f(\pi[B](\sigma[F](Inner)))](Outer)$$
$$\equiv \pi[attr(Outer)](\sigma[A\ \theta\ agg](Outer \bowtie \gamma[G; agg := f(B)](Outer \rtimes Inner)))$$

4.2 Subquery Coalescing

Bellamkonda *et al.* [2] describe which types of redundant subqueries can be merged. In this context the term "redundant" means that both subqueries access the same relations. Based on the work of Bellamkonda *et al.*, we define equivalence rules that can be used to merge redundant subqueries. Example 4.2.1 shows a query with two subqueries that can be merged into a single subquery.

Example 4.2.1 *The following query returns all orders that are at least as expensive as orders with high and medium priority.*

```
SELECT *
FROM Orders
WHERE o_totalprice >= ALL (
        SELECT o_totalprice
        FROM Orders
        WHERE o_orderpriority = '2-HIGH'
) AND o_totalprice >= ALL (
        SELECT o_totalprice
        FROM Orders
        WHERE o_orderpriority = '3-MEDIUM'
)
```

This query can be represented by the following NF² expression.

$$\sigma[o\_totalprice \geq MAX(\pi[o\_totalprice'](\sigma[o\_orderpriority = '2\text{-}HIGH'](Orders')))\wedge$$
$$o\_totalprice \geq MAX(\pi[o\_totalprice'](\sigma[o\_orderpriority = '3\text{-}MEDIUM'](Orders')))](Orders)$$

In order to eliminate the redundant table access, the predicate of the second subquery is combined with the predicate of the first subquery. Afterwards, the second subquery can be removed. In the following, the transformed query is shown.

```
SELECT *
FROM Orders
WHERE o_totalprice >= ALL (
        SELECT o_totalprice
        FROM Orders
        WHERE o_orderpriority = '2-HIGH'
            OR o_orderpriority = '3-MEDIUM'
)
```

The NF² expression that is equivalent to this query is:

$$\sigma[o\_totalprice \geq MAX(\pi[o\_totalprice'](\sigma[o\_orderpriority' = '2\text{-}HIGH' \vee o\_orderpriority' = '3\text{-}MEDIUM'](Orders')))](Orders).$$

There are three equivalence rules that cover the different cases in which subqueries in the WHERE clause can be merged into a single subquery. In the following definitions, the different cases that are covered by a rule are represented by a triple of the form $(\theta_1, \theta_2, \theta_3)$. θ_1 is the comparison operator that is used in the outer query to compare the result of the subquery with an outer column. θ_2 corresponds to the Boolean operator connecting both subqueries. Finally, θ_3 denotes the aggregation function which is used in the subqueries.

The rule given in Definition 4.2.1 covers the cases, in which two subqueries can be merged into a single subquery by combining their predicates with OR. We already applied this rule to transform the query shown in Example 4.2.1. In the following definitions the function $val(E)$ returns the result set of tuples of an expression E.

Definition 4.2.1 (Subquery Merge I) *Let F_1 and F_2 be conditions with $val(\sigma[F_1](Inner)) \not\subseteq val(\sigma[F_2](Inner))$ and $val(\sigma[F_2](Inner)) \not\subseteq val(\sigma[F_1](Inner))$. In addition, $\theta_1 \in \{<, \leq, >, \geq, =, \neq\}$, $\theta_2 \in \{\wedge, \vee\}$, and $\theta_3 \in \{MIN, MAX, COUNT\}$ are given. For the assignments $(\neq, \vee, COUNT)$, $(=, \wedge, COUNT)$, $(<, \wedge, MIN)$, $(>, \wedge, MAX)$, $(<, \vee, MAX)$, and $(>, \vee, MIN)$ for $(\theta_1, \theta_2, \theta_3)$, the following equivalence holds:*[2]

$$\sigma[(A\ \theta_1\ \theta_3(\pi[B](\sigma[F_1](Inner)))\ \theta_2 (A\ \theta_1\ \theta_3(\pi[B](\sigma[F_2](Inner))))](Outer)$$
$$\equiv \sigma[A\ \theta_1\ \theta_3(\pi[B](\sigma[F_1 \vee F_2](Inner)))](Outer).$$

The rule given in Definition 4.2.2 covers the cases, in which the second of two subqueries can be removed.

Definition 4.2.2 (Subquery Merge II) *Let F_1 and F_2 be conditions with $val(\sigma[F_2](Inner)) \subseteq val(\sigma[F_1](Inner))$. In addition, $\theta_1 \in \{<, \leq, >, \geq, =, \neq\}$, $\theta_2 \in \{\wedge, \vee\}$, and*

[2] If θ_3 equals $COUNT$, then A is zero.

$\theta_3 \in \{MIN, MAX, COUNT\}$ *is given. For the assignments* $(\neq, \vee, COUNT)$, $(=, \wedge, COUNT)$, $(<, \wedge, MIN)$, $(>, \wedge, MAX)$, $(<, \vee, MAX)$, *and* $(>, \vee, MIN)$ *for* $(\theta_1, \theta_2, \theta_3)$, *the following equivalence holds:*[2]

$$\begin{aligned}&\sigma[(A\ \theta_1\ \theta_3(\pi[A](\sigma[F_1](Inner)))\ \theta_2\\&\quad(A\ \theta_1\ \theta_3(\pi[A](\sigma[F_2](Inner))))](Outer)\\&\equiv \sigma[A\ \theta_1\ \theta_3(\pi[A](\sigma[F_1](Inner)))](Outer).\end{aligned}$$

The rule given in Definition 4.2.3 covers the cases, in which the first of two subqueries can be removed.

Definition 4.2.3 (Subquery Merge III) *Let* F_1 *and* F_2 *be conditions with* $val(\sigma[F_2](Inner)) \subseteq val(\sigma[F_1](Inner))$. *In addition,* $\theta_1 \in \{<, \leq, >, \geq, =, \neq\}$, $\theta_2 \in \{\wedge, \vee\}$, *and* $\theta_3 \in \{MIN, MAX, COUNT\}$ *is given. For the assignments* $(\neq, \wedge, COUNT)$, $(=, \vee, COUNT)$, $(<, \vee, MIN)$, $(>, \vee, MAX)$, $(<, \wedge, MAX)$, *and* $(>, \wedge, MIN)$ *for* $(\theta_1, \theta_2, \theta_3)$, *the following equivalence holds:*[2]

$$\begin{aligned}&\sigma[(A\ \theta_1\ \theta_3(\pi[A](\sigma[F_1](Inner)))\ \theta_2\\&\quad(A\ \theta_1\ \theta_3(\pi[A](\sigma[F_2](Inner))))](Outer)\\&\equiv \sigma[A\ \theta_1\ \theta_3(\pi[A](\sigma[F_2](Inner)))](Outer).\end{aligned}$$

In the previous definitions, whenever $>$ or $<$ is valid, the corresponding cases for $\geq$ and $\leq$ are also valid. In order to demonstrate how the correctness of these rules can be proven, a sample proof for a specific case of Definition 4.2.1 is included in Appendix C.

5. NEW OPTIMIZATION POSSIBILITIES

Up to now, we have demonstrated how the NF^2 algebra and its equivalences can be used to represent existing nested query optimizations. In this section, we present additional optimization techniques that are made possible at the algebraic level by our approach. We begin by introducing four auxiliary equivalence rules that we need for the definition of these new algebraic optimization possibilities.

5.1 Auxiliary Equivalence Rules

The first equivalence rule can be used to move a projection out of a join.

$$\pi[L_1](R) \bowtie_F \pi[L_2](S) \equiv \pi[L_1, L_2](R \bowtie_F S) \qquad (5.1.1)$$

The second equivalence rule can be used to eliminate a redundant equi-join. A join between a relation R and its renamed relation R' can be removed, if the subsequent operators (*e.g.*, a projection) are only accessing attributes of R.

Equi-join Elimination. *Let* R *be a relation,* $A_1, \ldots, A_n \in attr(R)$ *atomic attributes, and* $L \subseteq attr(R)$ *a projection list. In addition,* $R' := \pi[L'](R)$ *with* $L' = \{A' | A \in attr(R)\}$ *is given. Then the following holds:*

$$\pi[L](R \bowtie_{A_1=A'_1 \wedge \ldots \wedge A_n=A'_n} R') \equiv \pi[L](R). \qquad (5.1.2)$$

The next equivalence rule combines two projections in cases where the inner projection generates a nested attribute which the outer projection is accessing.

$$\begin{aligned}&\pi[L, \langle op\rangle(E)](\pi[L, E := \langle expr\rangle](R))\\&\equiv \pi[L, E := \langle op\rangle(\langle expr\rangle)](R)\end{aligned} \qquad (5.1.3)$$

The last equivalence rule moves a redundant subexpression of a projection into a separate projection. As a result, the subexpression is computed only once and the result is stored in a nested attribute.

$$\begin{aligned}&\pi[L, \langle op_1\rangle(\langle expr\rangle), \langle op_2\rangle(\langle expr\rangle)](R)\\&\equiv \pi[L, \langle op_1\rangle(E), \langle op_2\rangle(E)](\pi[L, E := \langle expr\rangle](R))\end{aligned} \qquad (5.1.4)$$

5.2 Redundant GROUP BY Elimination

The following query returns the price of the cheapest and most expensive order for each customer.

```
SELECT O1.o_custkey, max_price, min_price
FROM (SELECT o_custkey, MAX(o_totalprice) AS max_price
      FROM Orders
      GROUP BY o_custkey) O1,
     (SELECT o_custkey, MIN(o_totalprice) AS min_price
      FROM Orders
      GROUP BY o_custkey) O2
WHERE O1.o_custkey = O2.o_custkey
```

Obviously, this SQL statement is not an efficient formulation of the query as there is a redundant `GROUP BY` in the `FROM` clause. Nevertheless, as this query computes the correct result, a modern query optimizer should be able to remove this redundancy in order to obtain the best possible execution plan. As our experiments show, however, most current database systems do not perform this optimization (*cf.* Section 7).

The following NF^2 expression represents the above query. In the expression, "O" is an abbreviation for the Orders table (O' is the renamed Orders table).

$$\begin{aligned}&\pi[\text{o\_custkey, max\_price, min\_price}](\\&\quad\pi[\text{o\_custkey, max\_price} := \text{MAX}(\pi[\text{o\_totalprice'}](\text{G}))](\\&\qquad\pi[\text{o\_custkey, G} := \sigma[\text{o\_custkey'} = \text{o\_custkey}](\text{O'})](\text{O}))\\&\quad\bowtie_{\text{o\_custkey=o\_custkey"}}\\&\quad\pi[\text{o\_custkey"} := \text{o\_custkey, min\_price} :=\\&\qquad\text{MIN}(\pi[\text{o\_totalprice'}](\text{G}))](\pi[\text{o\_custkey, G} :=\\&\qquad\quad\sigma[\text{o\_custkey'} = \text{o\_custkey}](\text{O'})](\text{O})))\end{aligned}$$

The above expression is the result of strictly applying the definitions from Section 3 to the SQL statement. As a consequence, the grouping and the computation of the aggregation values are represented by two separate nested projections. In order to further transform and optimize the expressions, these projections have to be merged into a single projection, which can be achieved by applying Rule 5.1.3 from Section 5.1. While this intermediate step is included here for reasons of understandability, it is not necessary to implement this transformation (*cf.* Section 6) as it can be performed at the time when the SQL statement is translated into an NF^2 expression.

$$\begin{aligned}&\overset{(5.1.3)}{\equiv}\ \pi[\text{o\_custkey, max\_price, min\_price}](\\&\qquad\pi[\text{o\_custkey, max\_price} :=\\&\qquad\quad\text{MAX}(\pi[\text{o\_totalprice'}](\\&\qquad\qquad\sigma[\text{o\_custkey'} = \text{o\_custkey}](\text{O'})))](\text{O})\\&\qquad\bowtie_{\text{o\_custkey=o\_custkey"}}\\&\qquad\pi[\text{o\_custkey"} := \text{o\_custkey, min\_price} :=\\&\qquad\quad\text{MIN}(\pi[\text{o\_totalprice'}](\\&\qquad\qquad\sigma[\text{o\_custkey'} = \text{o\_custkey}](\text{O'})))](\text{O}))\end{aligned}$$

In the next step, the projections of the two group-by operators are moved outside the join using Rule 5.1.1.

$$\overset{(5.1.1)}{\equiv} \pi[\text{o\_custkey, max\_price, min\_price}](\pi[\text{o\_custkey}, \text{max\_price} := \text{MAX}(\pi[\text{o\_totalprice'}](\sigma[\text{o\_custkey'} = \text{o\_custkey}](\text{O'}))), \text{min\_price} := \text{MIN}(\pi[\text{o\_totalprice'}](\sigma[\text{o\_custkey'} = \text{o\_custkey}](\text{O'})))](\text{O} \bowtie_{\text{o\_custkey=o\_custkey"}} \text{O"}))$$

Now, the redundant join can be removed by applying Rule 5.1.2.

$$\overset{(5.1.2)}{\equiv} \pi[\text{o\_custkey, max\_price, min\_price}](\pi[\text{o\_custkey}, \text{max\_price} := \text{MAX}(\pi[\text{o\_totalprice'}](\sigma[\text{o\_custkey'} = \text{o\_custkey}](\text{O'}))), \text{min\_price} := \text{MIN}(\pi[\text{o\_totalprice'}](\sigma[\text{o\_custkey'} = \text{o\_custkey}](\text{O'})))](\text{O}))$$

Finally, the group-by/aggregate operator is split into two projections using the rule from Definition 3.1.3 in Section 3. While this step illustrates how these projections can be mapped to the physical group-by/aggregate operator, it is again not necessary to implement a transformation rule for this step (*cf.* Section 6).

$$\overset{(5.1.4)}{\equiv} \pi[\text{o\_custkey}, \text{max\_price} := \text{MAX}(\pi[\text{o\_totalprice'}](\text{G})), \text{min\_price} := \text{MIN}(\pi[\text{o\_totalprice'}](\text{G})](\pi[\text{o\_custkey}, \text{G} := \sigma[\text{o\_custkey'} = \text{o\_custkey}](\text{O'}))](\text{O})$$

$$\overset{(3.1.3)}{\equiv} \gamma[\text{o\_custkey}; \text{max\_price} := \text{MAX}(\text{o\_totalprice}), \text{min\_price} := \text{MIN}(\text{o\_totalprice})](\text{O})$$

To conclude, below the SQL statement that is equivalent to the NF^2 expression above is shown.

```
SELECT o_custkey, max_price, min_price
FROM (SELECT o_custkey, MAX(o_totalprice) AS max_price,
                        MIN(o_totalprice) AS min_price
      FROM Orders
      GROUP BY o_custkey) O1
```

5.3 SELECT Clause Subquery Elimination

The example given in this section shows how a correlated subquery in the `SELECT` clause can be eliminated. For each part type the following query returns the number and the average price of the corresponding parts.

```
SELECT p_type, AVG(p_retailprice), (SELECT COUNT(*)
               FROM Part P2 WHERE P2.p_type = P1.p_type)
FROM Part P1
GROUP BY p_type
```

The equivalent NF^2 expression is as follows:

$$\pi[\text{p\_type}, \text{AVG}(\pi[\text{p\_retailprice'}](G)), \text{COUNT}(\sigma[\text{p\_type'} = \text{p\_type}](\text{P'}))](\pi[\text{p\_type}, G := \sigma[\text{p\_type'} = \text{p\_type}](\text{P'})](\text{P})).$$

First, the projections can be merged by applying Rule 5.1.3.

$$\overset{(5.1.3)}{\equiv} \pi[\text{p\_type}, \text{AVG}(\pi[\text{p\_retailprice'}](\sigma[\text{p\_type'} = \text{p\_type}](\text{P'}))), \text{COUNT}(\sigma[\text{p\_type'} = \text{p\_type}](\text{P'}))](\text{P})$$

Now, the aggregation functions have the same groupings as input. Therefore, the redundant groupings can be removed and added as a separate projection.

$$\overset{(5.1.4)}{\equiv} \pi[\text{p\_type}, \text{AVG}(\pi[\text{p\_retailprice'}](G)), \text{COUNT}(G)](\pi[\text{p\_type}, G := \sigma[\text{p\_type'} = \text{p\_type}](\text{P'})](\text{P}))$$

$$\overset{(3.1.3)}{\equiv} \gamma[\text{p\_type}; \text{AVG}(\text{p\_retailprice}), \text{COUNT}(*)](\text{P})$$

Finally, the SQL statement that is equivalent to the above NF^2 expression is shown below.

```
SELECT p_type, AVG(p_retailprice), COUNT(*)
FROM Part
GROUP BY p_type
```

6. IMPLEMENTATION

In order to demonstrate the feasibility of our approach, we extended an existing optimizer, which is based on the Cascades framework designed by Graefe [11]. We chose this optimizer framework for two reasons. First, its general and modular architecture was specifically designed to support the definition of new operators and transformation rules. Second, the Cascades framework has been used to build commercial query optimizers such as the one of Microsoft SQL Server.

As a starting point for our NF^2 optimizer, we use an existing implementation of the Cascades framework that is based on the operators and transformation rules of the traditional relational algebra. As the NF^2 algebra is an extension of the relational algebra, it is only necessary to extend this implementation at three points.

First, we have to define how NF^2 algebra expressions are represented. In Cascades, expressions are modeled by class `EXPR` that consists of an operator and its input expressions. The number of input expressions that an operator can have is defined by its arity. In order to work with NF^2 expressions, nested selections and nested projections need to be introduced. Representing nested selections does not require any changes to the data structures, as the Cascades implementation described by Graefe already represents selection conditions as an expression tree consisting of so-called item operators. Accommodating nested projections with minimal changes is more challenging. In contrast to selection conditions, Graefe models the projection list as a parameter to the projection operator, rather than a (list of) input expressions. Unlike input expressions, operator parameters are not explored and therefore not optimized. A reason for this design is the fact that it would lead to a projection operator with variable arity, which is at odds with the static patterns used to match transformation rules to expressions. In order to keep the arity of the projection constant, while enabling nested expression, we introduce a new `GET_NESTED_EXPR` logical operator, which models the subexpressions in the projection list without affecting the rule engine. Figure 1 illustrates this design for a simple nested projection on relation R.

Second, we have to slightly adapt the control flow of the optimizer. The optimization algorithm of the Cascades frame-

Figure 1: Representation of a nested projection.

work is structured as a set of tasks. For our NF^2 optimizer, we have to modify the existing "optimize expression" task (class `O_EXPR`), which applies all relevant transformation rules to an expression in order to enumerate equivalent expressions. As described by Graefe, this task does not traverse expressions built from item operators. However, since the comparison operator that has the nested query as an input is an item operator, this behavior has to be changed.

Finally, transformation rules for the NF^2 equivalences have to be added to the optimizer. In principle, each equivalence gives rise to two transformations, depending on the direction it is applied. For the subquery unnesting and coalescing rules, we only implement the direction that removes a nesting or redundancy, since it is obvious which variant is more efficient. Note that the three subquery coalescing rules (*cf.* Definitions 4.2.1, 4.2.2, 4.2.3) can be implemented as one transformation rule. As they share the same input pattern, the output pattern can be created depending on the case. In order to apply some of our optimizations, projections need to be *pushed up* to move them out of a join. This transformation is in contradiction to the optimizer's general strategy of *pushing down* projections. This conflict is resolved by adding a compound rule that both moves the projection out of the join and performs the subsequent optimization. Some equivalences (*cf.* Definition 4.1.1) rely on set semantics, *i.e.*, it is important to consider the handling of duplicates in their implementation as a transformation rule. To obtain equivalent results, these rules insert explicit duplicate-elimination operators that enforce set semantics, if necessary.

To conclude this implementation section, we study the impact of our extensions on the code complexity of the optimizer. Originally, the optimizer consisted of 5972 LOC, whereas the extended optimizer has 7630 LOC. This constitutes an increase of 27.8%. However, most of this additional code implements the new transformation rules and only 48 LOC were added to the control flow of the optimization algorithm. Additionally, we study the cyclomatic complexity [15] of our extensions. The original optimizer had an average cyclomatic complexity of 2.241, whereas the value for the extended optimizer is 2.398, *i.e.*, an increase of 7%.

7. EVALUATION

In order to quantitatively evaluate the performance benefits that can be obtained with our approach in commercial database systems, we first define a set of eleven nested queries on the schema of the TPC-H benchmark. For each query, we use our optimizer to generate an optimized NF^2 expression, which is translated back to its equivalent SQL statement. We execute both the original and the optimized SQL query in four state-of-the-art database systems. In this way, we can show that our NF^2 optimizer is able to find more efficient execution strategies than the optimizers of these systems. Finally, we study how NF^2 transformation rules impact the optimization time and the memory usage of our optimizer.

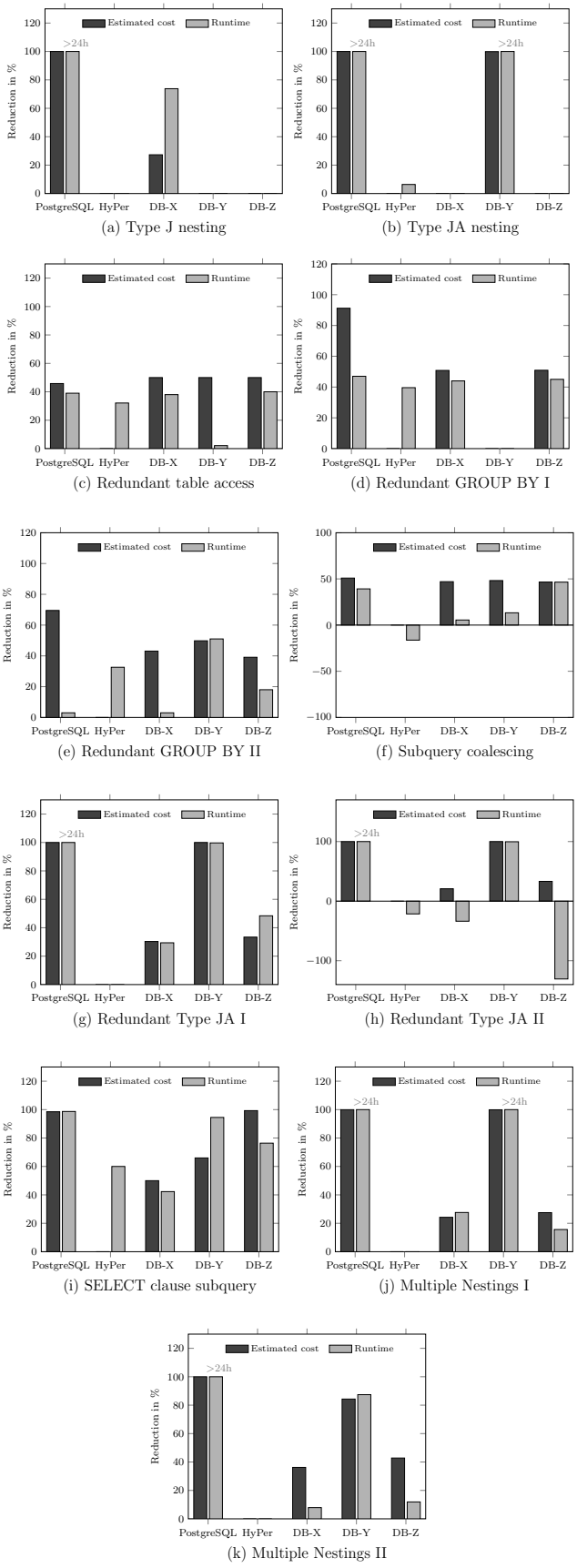

Figure 2: Reduction of estimated cost and runtime.

7.1 Experimental Setup

All experiments presented in this work were performed on a Mac Pro with a 3.5 GHz 6-Core Intel Xeon E5 processor with 64 GB main memory. In our evaluation, we use PostgreSQL Version 9.4.2, the in-memory database HyPer[3] and three market-leading commercial database systems. Due to license terms, the commercial systems are referred to as "DB-X", "DB-Y", and "DB-Z". PostgreSQL, DB-X, DB-Y and DB-Z were installed in a 64-bit Windows 8.1 virtual machine. HyPer was installed in a 64-bit Ubuntu 14.04 LTS virtual machine. 32 GB main memory were allocated to both virtual machines. For all systems, except HyPer, the buffer pool size was set to 2 GB[4]. For all systems, a 10 GB TPC-H database was stored on external storage. Runtime measurements for each of the eleven queries were repeated five times. In order to reduce the bias induced by caching, the queries were executed in random order. Out of the five measurements, the smallest and largest value was discarded and the average of the remaining values was computed.

7.2 Cost and Runtime of the Queries

For each database system, we determine the execution time of the original and the transformed query. With the exception of HyPer, we also determine the cost (estimated by the optimizer of the corresponding system) of the original and the transformed query. To the best of our knowledge, it is not possible to access the estimated cost in HyPer. In order to obtain a measure that is relevant in practical applications, the optimization time needed by our NF^2 optimizer is added to the execution time. In the following, we simply refer to this sum of optimization and execution time as "runtime". The optimization times for the original and transformed query are measured by deactivating and activating the NF^2 rules in our optimizer. Figure 2 plots the reduction of estimated costs and runtime for each query, which is computed as

$$\frac{(\text{value original query} - \text{value transformed query})}{\text{value original query}} * 100,$$

where "value original query" denotes the estimated cost/runtime of the original query and "value transformed query" denotes the estimated cost/runtime of the corresponding transformed query. Table 1 gives the absolute runtimes of the original and transformed queries. In the table, "–" is used to indicate that the execution plan of the original and transformed query is the same. In this case, the query was not executed at all. The SQL statements for both the original and the transformed query are given in Appendix A.

Query (a) contains a Type J subquery. HyPer, DB-X, DB-Y and DB-Z are all able to optimize this query. However, HyPer and DB-X use different join orderings and join implementations for the original and transformed query. In DB-X, the transformed query leads to a performance increase, whereas the runtime in HyPer is not reduced by the alternative plan. DB-Y and DB-Z execute the original and transformed query in the same way. For that reason, there is no reduction in the runtime and the estimated costs. However, PostgreSQL cannot transform the original query. Instead, it executes the subquery for each outer tuple. Even after 24 hours the original query did not terminate and was stopped manually. For each query that was manually aborted after 24 hours, the label ">24h" is added in the corresponding chart. The query transformed by our NF^2 optimizer was executed in 17 seconds by PostgreSQL.

Query (b) contains a Type JA subquery. As before, PostgreSQL cannot transform this query. Again, the original query was aborted after 24 hours, whereas the transformed query executed in 22 seconds. The other systems are able to perform a subquery unnesting. Interestingly, DB-Y finds a better plan for the query when given our transformed query than when it performs the same transformation itself. In the former case, a hash join is used to implement the join that results from the transformation, whereas in the latter case a nested-loops join is used. As a consequence, the original query was aborted after 24 hours. Likewise, HyPer produces a different plan for the original and transformed query.

Query (c) contains two subqueries in the `FROM` clause that access the same relation, but compute different aggregation functions. None of the systems can merge these subqueries into a single subquery. The transformed query, which has only one subquery, reduced the runtime in PostgreSQL, HyPer, DB-X and DB-Z by about 40%. While DB-Y estimates a reduction of about 50%, the transformed query does not lead to a substantial reduction of the runtime.

Query (d) from Section 5 uses a redundant `GROUP BY` in the `FROM` clause. Only DB-Y is able to remove the redundancy. Hence, for all other systems, the transformed query containing only one `GROUP BY` leads to a runtime reduction of around 50%.

Query (e) is an extension of Query (d), in which the `WHERE` clause of the subqueries have different predicates. The two subqueries can be merged into a single subquery by moving the predicates into the aggregation function as `CASE` statements and by adding an additional predicate to the outer query. None of the systems can eliminate this type of redundancy. For PostgreSQL and DB-X, we can again observe that the cost estimation does not accurately reflect the actual runtimes.

Query (f) has redundant subqueries in the `WHERE` clause that can be merged using the technique described by Bellamkonda *et al.* [2]. Surprisingly, no system eliminates this redundancy, even though they estimate a performance increase of about 50%. Although the transformed query leads to a reduction of the actual execution time in HyPer, the increase of the optimization time reduces the overall performance. This is due to the fact that the runtime of a query in HyPer falls within the scope of milliseconds. Therefore, the increase of the optimization time reduces the performance.

Query (g), which we introduced as a motivating example in Section 1, contains two redundant Type JA queries. As the query uses different predicates to compare an outer column value to the result of a subquery, the subquery coalescing proposed by Bellamkonda *et al.* cannot be applied. Instead, our NF^2 optimizer eliminates the redundant subquery by first unnesting both subqueries. As a result, two subqueries with the same `GROUP BY` clause are contained in the `FROM` clause, which can be merged into one subquery. As stated, no current database system features an optimizer that can perform all of these steps. Although DB-Y is able to do a subquery unnesting and a removal of redundant `GROUP BY`s, it cannot combine these steps to optimize the given query. PostgreSQL was aborted after 24 hours, whereas the transformed query executed in 3 seconds. Another significant

[3]http://www.hyper-db.de (February 16, 2016)

[4]One system was unable to execute the queries in reasonable time using a smaller buffer pool size.

Database	Query (a)	Query (b)	Query (c)	Query (d)	Query (e)	Query (f)	Query (g)	Query (h)	Query (i)	Query (j)	Query (k)
PostgreSQL	> 24 h	> 24 h	9896	58611	34108	17710	> 24 h	> 24 h	96228	> 24 h	> 24 h
	17377	22057	6070	31155	33050	10770	3216	3355	1292	7079	18315
HyPer	131	233	78	96	92	160	137	145	55	156	×
	134	218	53	58	62	186	133	176	22	158	×
DB-X	4817	–	1402	5152	2581	1471	588	418	506	1038	8525
	1261	–	869	2887	2503	1390	415	559	292	751	7853
DB-Y	–	> 24 h	29700	–	45224	23470	1177484	1171713	40908	> 24 h	395831
	–	8907	29076	–	21979	20353	3509	2965	2240	2522	50065
DB-Z	–	–	1273	14590	7948	72747	1006	301	1597	9987	4740
	–	–	760	8079	6551	38901	519	694	377	8426	4178

Table 1: Runtimes (in ms) of the original queries (top) and the transformed queries (bottom).

performance gain is achieved in DB-Y, where the original query is a factor 350 slower than the query transformed by our NF^2 optimizer. Due to the increase of the optimization time, there is no performance gain in HyPer.

Query (h) is an extension of Query (g), where the Type JA subqueries have different predicates in their `WHERE` clause. Again, the NF^2 optimizer unnests both subqueries. Then, the redundant `GROUP BY` with different predicates can be removed (*cf.* Query (e)). No system is able to remove this type of redundancy. For PostgreSQL and DB-Y, a significant performance increase can be achieved by executing the transformed query. Even though they estimate a performance benefit, the execution time of the transformed query is larger in DB-X and DB-Z. The reason for the performance decrease could be that the predicates in the original query are very selective. Therefore, only a small amount of tuples have to be grouped by each of the two `GROUP BY` statements. However, in the transformed query the grouping is performed on all tuples of the input relation. Tuples that should not be in the input of the aggregation function are then removed using a `CASE` statement. Again, in HyPer there is no performance gain, since the increase of the optimization time dominates the overall performance.

Query (i) from Example 3.3.1 uses a subquery in the `SELECT` clause. In section 5 we showed, how this type of subquery can be removed. However, no system performed this optimization. Therefore, the transformed query of our NF^2 optimizer led to a performance increase in all systems. For PostgreSQL, the original query is a factor 75 slower than the transformed query. For DB-Y the original query is a factor 18 slower and for DB-Z the original query is a factor 4 slower than the transformed query.

Query (j) has multiple subqueries within other subqueries. PostgreSQL and DB-Y were aborted after 24 hours, whereas the transformed query executed in less than 10 seconds in both systems. Again, due to the increase of the optimization time, there is no performance increase in HyPer.

Query (k) combines different types of nested queries. PostgreSQL was aborted after 24 hours. HyPer could not execute Query (k) and returned an abstract syntax tree error (denoted by "×" in Table 1).

7.3 NF^2 Optimization Time and Memory

In order to quantify the implications of our approach to the query optimizer itself, we measured the optimization time and the memory consumption with and without NF^2 rules.

Figure 3 (top) plots the optimization time required by our NF^2 optimizer for each query with NF^2 rules activated and deactivated. For Query (a) to (e) and Query (i), there is only a small increase when the NF^2 rules are activated. For the other queries, the increase in optimization time is larger. The cause is that the implementation of the Cascades framework that we extended never merges expression groups, even if they contain equivalent expressions. For example, the search space for Query (k) currently consists of 2,058 groups. However, if redundant groups were merged during optimization, only 146 groups would be required. We are therefore integrating this functionality into our NF^2 optimizer.

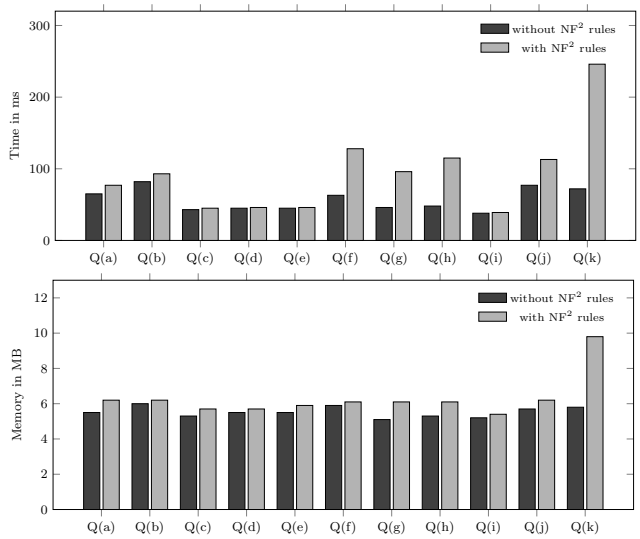

Figure 3: Optimization time (top) and memory usage (bottom) of the NF^2 optimizer.

Figure 3 (bottom) shows the memory usage of our NF^2 optimizer for each query with NF^2 rules activated and deactivated. Only Query (k) uses notably more memory when NF^2 rules are activated. Again, this is due to the fact that equivalent expression groups are not merged. The memory usage of the other queries is stable at around 5 to 6 MB.

7.4 Discussion and Limitations

To conclude this section, we discuss limitations of the presented evaluation that we are aware of. One possible weakness is the set of queries that we used to quantify the performance benefits of our approach. Since the queries of the TPC-H benchmark do not cover all optimization techniques that have been proposed in the literature, we defined our own set of eleven queries. Our first goal was to cover all types of nested queries for which optimizations have been proposed in the literature. Next, we aimed at covering all types of nesting that are possible in SQL. Finally, we introduced queries that

combine different types of nesting, requiring different combinations of optimizations. We argue that this methodology is sound and that our set of queries can be used to identify the optimization opportunities that are still present in current database systems w.r.t. nested query optimization. At the same time, we cannot make any claims as to the relevance or frequency of these queries in practice.

Furthermore, our query set does not include queries that use disjunction, non-inner-joins, or non-equi-joins. Such queries can be executed using either nested-loops evaluation or special physical operators. In this paper, we do not claim to improve the state of the art at the physical level, but we clearly demonstrate that there is still much to be gained at the logical level by using the NF^2 algebra to represent and transform nested queries. This claim even holds for database systems that have special physical operators for nested queries (*e.g.*, SQL Server [9] and HyPer [17]). If such special physical operators exist, they can be incorporated into our approach by appropriate implementation rules.

Another limitation of our evaluation is its focus on nested queries. While this approach is appropriate to quantify the benefits of the optimization technique that we propose, it does not shed light on potential drawbacks it might have, if our NF^2 optimizer was applied in a setting with mostly non-nested queries. Although we acknowledge that such a study could be conducted, we also point out that it is possible to determine at parse-time whether or not a SQL query has subqueries. Since our Cascades-based NF^2 optimizer implementation supports the dynamic activation and deactivation of transformation rules, this parse-time information could be used to configure different "optimization profiles", which could mitigate some of the overhead introduced by our approach, when it is not needed.

8. RELATED WORK

The optimization techniques for nested queries that have been proposed over the last 30 years can be categorized into three classes. Approaches of the first category perform transformations at the level of SQL (or a query representation close to SQL). Approaches of the second category extend the relational algebra with new logical and physical operators that are specifically designed to represent nested queries. Finally, approaches of the third category address nested query optimization using an entirely different formalism.

Kim [14] is the first to address the optimization of nested SQL queries. His approach falls into the first catergory as it uses stepwise edits of the SQL code with the goal of replacing subqueries with joins. Our approach can express all transformations described by Kim, given that they are still possible in the current version of SQL. Zuzarte *et al.* [20] describe how certain subqueries can be replaced by window functions. Even though not included in this paper, our approach can also support these transformations (*cf.* Appendix B). Finally, Bellamkonda *et al.* [2] describe under which conditions subqueries in the `WHERE` clause can be merged. Their approach is implemented in Oracle based on their cost-based query transformation framework that uses "*query trees, which are different from algebraic operator trees in that query trees retain all the declarativeness of SQL*" [1]. In Section 4, we showed how subquery coalescing is realized in our approach. In contrast to these approaches, however, our transformation are integrated into the logical optimization phase, rather than performed at the level of SQL.

The work of Dayal [7], which falls into the second category, extends the relational algebra with special join and aggregation operators. It also sketches how these operators could be implemented in a database system. Cluet and Moerkotte [5] define an algebra and equivalences to optimize nested queries in object databases. Whereas most of the operators that they propose are nowadays supported by database systems, they also introduce a special join (*d-join*) to perform unnesting. Galindo-Legaria and Joshi [9] describe how Microsoft SQL Server transforms nested queries. They extend the relational algebra with the so-called apply operator that is used to algebraically represent nested queries in the `WHERE` clause. Both Brantner *et al.* [3] and Neumann and Kemper [17] extend the relational algebra with new operators to algebraically represent subquery unnestings. In the latter case, these operators use sideways information passing to execute nested queries more efficiently. Similar to our work, the approaches of Wang *et al.* [19] and Cao and Badia [4] also propose the use of nested relational algebras, but assume the presence of physical nest and unnest operators. Compared to these approaches of the second category, our approach performs only logical optimizations and every logical operator that we use can be mapped to a physical operator found in any current database system. For that reason, the plans generated by our NF^2 optimizer can be executed in every existing relational database system.

Instead of extending the relational algebra, approaches of the third category introduce an entirely different formalism by extending the relational calculus. For example, Fegaras and Maier [8] propose the monoid comprehension calculus and demonstrate how it can be used to remove any form of nesting in OQL. Similarly, the work of Grust and Scholl [13] extends monad comprehension calculus for which it can also be shown that normalization is complete [12]. Since the relational algebra rather than the relational calculus is the formal foundation of existing query optimizers, the goal of our work is to express unnesting rules in a formalism that fosters their practical adoption.

9. CONCLUSION AND FUTURE WORK

In this paper, we presented how nested queries can be optimized algebraically by representing them in the NF^2 algebra and applying transformation rules that are based on the equivalences of this algebra. A key advantage of this approach is that the optimization of nested queries can be performed as part of the logical plan enumeration phase. Furthermore, since the NF^2 algebra is an extension of the traditional relational algebra, all existing equivalences (and the corresponding transformation rules) are still valid and can work hand-in-hand with new rules. We demonstrated how this approach can be integrated into a Cascades-based query optimizer with little effort. Finally, we used this implementation to quantify the potential performance benefits that current database systems can achieve based on our approach.

Although we have focused on the optimization of nested queries in this paper, the applications of the NF^2 algebra in the context of SQL are not limited to this use case. As another line of future work, we therefore plan to extend our definitions to other SQL concepts, which then could be similarly optimized by algebraic transformation rules. For example, the `CUBE` or `ROLLUP` operator could be represented by NF^2 expressions and new transformation rules could be derived to optimize OLAP queries.

10. REFERENCES

[1] R. Ahmed, A. Lee, A. Witkowski, D. Das, H. Su, M. Zait, and T. Cruanes. Cost-based Query Transformation in Oracle. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 1026–1036, 2006.

[2] S. Bellamkonda, R. Ahmed, A. Witkowski, A. Amor, M. Zait, and C.-C. Lin. Enhanced Subquery Optimizations in Oracle. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, volume 2, pages 1366–1377, 2009.

[3] M. Brantner, N. May, and G. Moerkotte. Unnesting Scalar SQL Queries in the Presence of Disjunction. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 46–55, 2007.

[4] B. Cao and A. Badia. SQL Query Optimization Through Nested Relational Algebra. *ACM Transactions on Database Systems (TODS)*, 32, 2007.

[5] S. Cluet and G. Moerkotte. Nested Queries in Object Bases. In *Proc. Intl. Workshop on Database Programming Languages (DBPL)*, pages 226–242, 1993.

[6] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13:377–387, 1970.

[7] U. Dayal. Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. In *Proc. Intl. Conf. on Management of Data (SIGMOD)*, pages 197–208, 1987.

[8] L. Fegaras and D. Maier. Optimizing Object Queries Using an Effective Calculus. *ACM Trans. Database Syst.*, 25(4):457–516, 2000.

[9] C. Galindo-Legaria and M. Joshi. Orthogonal Optimization of Subqueries and Aggregation. In *Proc. Intl. Conf. on Management of Data (SIGMOD)*, pages 571–581, 2001.

[10] R. A. Ganski and H. K. T. Wong. Optimization of Nested SQL Queries Revisited. In *Proc. Intl. Conf. on Management of Data (SIGMOD)*, pages 23–33, 1987.

[11] G. Graefe. The Cascades Framework for Query Optimization. *Data Engineering Bulletin*, 18:19–29, 1995.

[12] T. Grust. *Comprehending Queries*. PhD thesis, University of Konstanz, 1999.

[13] T. Grust and M. H. Scholl. How to Comprehend Queries Functionally. *J. Intell. Inf. Syst.*, 12(2-3):191–218, 1999.

[14] W. Kim. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems (TODS)*, 7:443–469, 1982.

[15] T. J. McCabe. A Complexity Measure. In *Proc. Intl. Conf. on Software Engineering (ICSE)*, pages 308–320, 1976.

[16] G. Moerkotte. Building Query Compilers. Available at: http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf, Dec. 2014. Under Construction.

[17] T. Neumann and A. Kemper. Unnesting Arbitrary Queries. In *Proc. Datenbanksysteme für Business, Technologie und Web (BTW)*, 2015.

[18] H.-J. Schek and M. H. Scholl. The Relational Model with Relation-Valued Attributes. *Information Systems*, 11:137–147, 1986.

[19] Q. Wang, D. Maier, and L. Shapiro. Algebraic Unnesting for Nested Queries. *CSETech. Paper 252*, 1999.

[20] C. Zuzarte, H. Pirahesh, W. Ma, Q. Cheng, L. Liu, and K. Wong. Winmagic: Subquery Elimination Using Window Aggregation. In *Proc. Intl. Conf. on Management of Data (SIGMOD)*, pages 652–656, 2003.

APPENDIX

In this appendix, the eleven queries used in the evaluation, NF^2 representations of additional SQL concepts, and a sample proof for an equivalence rule are given.

A. QUERIES

In this section, we present the queries that were used in the evaluation of this work. For each query, we also show the transformed version generated by our NF^2 optimizer.

Query (a): Type J Nesting

This query returns all parts delivered by suppliers from the region 'America'.

Original query

```
SELECT s_name, p_name
FROM Part, Supplier, Nation, Region
WHERE s_nationkey = n_nationkey AND n_regionkey = r_regionkey
    AND r_name = 'AMERICA'
    AND p_partkey IN (SELECT ps_partkey
                      FROM PartSupp
                      WHERE ps_suppkey = s_suppkey)
```

Transformed query

```
SELECT DISTINCT p_name, s_name
FROM Part, Supplier, PartSupp, Nation, Region
WHERE p_partkey = ps_partkey AND ps_suppkey = s_suppkey
    AND s_nationkey = n_nationkey
    AND n_regionkey = r_regionkey
    AND r_name = 'AMERICA'
```

Query (b): Type JA Nesting

This query returns all parts delivered by suppliers from Asia that are more expensive than the average price of parts of the same type delivered by suppliers from the region 'America'.

Original query

```
SELECT P1.p_name
FROM Part P1, PartSupp, Supplier, Nation, Region
WHERE p_partkey = ps_partkey AND ps_suppkey = s_suppkey
  AND s_nationkey = n_nationkey AND n_regionkey = r_regionkey
AND r_name = 'ASIA' AND P1.p_retailprice > (
  SELECT AVG(P2.p_retailprice)
  FROM Part P2, PartSupp PS2,
    Supplier S2, Nation N2, Region R2
  WHERE P2.p_type = P1.p_type
    AND P2.p_partkey = PS2.ps_partkey
    AND PS2.ps_suppkey = S2.s_suppkey
    AND S2.s_nationkey = N2.n_nationkey
    AND N2.n_regionkey = R2.r_regionkey
    AND R2.r_name = 'AMERICA')
```

Transformed query

```
SELECT P1.p_name
FROM Part P1, PartSupp PS1, Supplier S1, Nation N1,
```

```
  Region R1, (SELECT p_type, AVG(p_retailprice) AS avg_price
   FROM Part, PartSupp, Supplier, Nation, Region
   WHERE p_partkey = ps_partkey AND ps_suppkey = s_suppkey
     AND s_nationkey = n_nationkey
     AND n_regionkey = r_regionkey
     AND r_name = 'AMERICA'
   GROUP BY p_type) P2
WHERE P2.p_type = P1.p_type
   AND P1.p_retailprice > P2.avg_price
   AND P1.p_partkey = PS1.ps_partkey
   AND PS1.ps_suppkey = S1.s_suppkey
   AND S1.s_nationkey = N1.n_nationkey
   AND N1.n_regionkey = R1.r_regionkey
   AND R1.r_name = 'ASIA'
```

Query (c): Redundant Table Access

This query returns the price of the cheapest and most expensive order.

Original query

```
SELECT min_price, max_price
FROM (SELECT MIN(o_totalprice) AS min_price
       FROM Orders) O1,
      (SELECT MAX(o_totalprice) AS max_price
       FROM Orders) O2
```

Transformed query

```
SELECT min_price, max_price
FROM (SELECT MIN(o_totalprice) AS min_price,
       MAX(o_totalprice) AS max_price
      FROM Orders) O1
```

Query (d): Redundant GROUP BY I

This query returns the cheapest and most expensive order for each customer.

Original query

```
SELECT O1.o_custkey, max_price, min_price
FROM (SELECT o_custkey, MAX(o_totalprice) AS max_price
       FROM Orders
       GROUP BY o_custkey) O1,
      (SELECT o_custkey, MIN(o_totalprice) AS min_price
       FROM Orders
       GROUP BY o_custkey) O2
WHERE O1.o_custkey = O2.o_custkey
```

Transformed query

```
SELECT o_custkey, max_price, min_price
FROM (SELECT o_custkey, MAX(o_totalprice) AS max_price,
       MIN(o_totalprice) AS min_price
       FROM Orders
       GROUP BY o_custkey) O1
```

Query (e): Redundant GROUP BY II

This query returns for each customer the fraction between the urgent orders and the total number of orders. Compared to the previous query, the two subqueries of this query contain different predicates in the `WHERE` clause.

Original query

```
SELECT O1.o_custkey, nrUrgentOrders/nrOrders
FROM (SELECT o_custkey, COUNT(*) AS nrOrders
       FROM Orders
       GROUP BY o_custkey) O1,
      (SELECT o_custkey, COUNT(*) AS nrUrgentOrders
       FROM Orders
       WHERE o_orderpriority = '1-URGENT'
       GROUP BY o_custkey) O2
WHERE O1.o_custkey = O2.o_custkey
```

Transformed query

```
SELECT o_custkey, nrUrgentOrders/nrOrders AS ratio
FROM
  (SELECT o_custkey, COUNT(*) AS nrOrders, COUNT(
   CASE WHEN o_orderpriority = '1-URGENT' THEN 1 END)
    AS nrUrgentOrders
   FROM Orders
   GROUP BY o_custkey) O1
WHERE nrUrgentOrders <> 0
```

Query (f): Subquery Coalescing

This query returns all orders from customers located in Asia that are at least as expensive as the most expensive orders with urgent, high and medium priority. In contrast to the previous queries, the redundancy is now contained in the `WHERE` clause.

Original query

```
SELECT o_orderkey
FROM Orders, Customer, Nation, Region
WHERE o_custkey = c_custkey AND c_nationkey = n_nationkey
    AND n_regionkey = r_regionkey AND r_name='ASIA'
    AND o_totalprice >= (
        SELECT MAX(o_totalprice)
        FROM Orders
        WHERE o_orderpriority = '1-URGENT'
) AND o_totalprice >= (
        SELECT MAX(o_totalprice)
        FROM Orders
        WHERE o_orderpriority = '2-HIGH'
) AND o_totalprice >= (
        SELECT MAX(o_totalprice)
        FROM Orders
        WHERE o_orderpriority = '3-MEDIUM'
)
```

Transformed query

```
SELECT o_orderkey
FROM Orders, Customer, Nation, Region
WHERE o_custkey = c_custkey AND c_nationkey = n_nationkey
    AND n_regionkey = r_regionkey AND r_name='ASIA'
AND o_totalprice >= (
        SELECT MAX(o_totalprice)
        FROM Orders
        WHERE o_orderpriority = '1-URGENT'
              OR o_orderpriority = '2-HIGH'
              OR o_orderpriority = '3-MEDIUM'
)
```

Query (g): Redundant Type JA I

This query returns all parts that are cheaper and bigger than the average of parts of the same type. In contrast to the previous query, the aggregation functions of the subqueries are computed on different attributes.

Original query

```
SELECT P1.p_name
FROM Part P1
WHERE P1.p_retailprice < (
        SELECT AVG(P2.p_retailprice)
        FROM Part P2
        WHERE P2.p_type = P1.p_type
) AND P1.p_size > (
        SELECT AVG(P2.p_size)
        FROM Part P2
        WHERE P2.p_type = P1.p_type
)
```

Transformed query

```
SELECT p_name
FROM Part P1,
  (SELECT p_type, AVG(p_size) AS avg_size,
    AVG(p_retailprice) AS avg_price
   FROM Part
   GROUP BY p_type) P2
WHERE P1.p_type = P2.p_type
    AND p_retailprice < avg_price AND p_size > avg_size
```

Query (h): Redundant Type JA II

This query returns all parts that are cheaper than the average price of parts of the same type of brand "Brand#13" and that are bigger than the average size of parts of the same type of brand "Brand#14". Compared to the previous query, the two subqueries of this query contain different predicates in the `WHERE` clause.

Original query

```
SELECT P1.p_name
FROM Part P1
WHERE P1.p_retailprice < (
     SELECT AVG(P2.p_retailprice)
     FROM Part P2
     WHERE P2.p_type = P1.p_type
       AND P2.p_brand = 'Brand#13'
) AND P1.p_size > (
     SELECT AVG(P2.p_size)
     FROM Part P2
     WHERE P2.p_type = P1.p_type
       AND P2.p_brand = 'Brand#14'
)
```

Transformed query

```
SELECT p_name
FROM Part P1,
  (SELECT p_type,
   AVG(CASE WHEN p_brand = 'Brand#14' THEN p_size END)
    AS avg_size
   AVG(CASE WHEN p_brand = 'Brand#13' THEN p_retailprice
    END) AS avg_price
   FROM Part
   GROUP BY p_type) P2
WHERE P1.p_type = P2.p_type AND p_retailprice < avg_price
        AND p_size > avg_size
        AND avg_size IS NOT NULL AND avg_price IS NOT NULL
```

Query (i): SELECT Clause Subquery

This query returns for each part type the number of parts and the average price.

Original query

```
SELECT p_type, AVG(p_retailprice),
(SELECT COUNT(*) FROM Part P2 WHERE P2.p_type = P1.p_type)
FROM Part P1
GROUP BY p_type
```

Transformed query

```
SELECT p_type, AVG(p_retailprice), COUNT(*)
FROM Part
GROUP BY p_type
```

Query (j): Multiple Nestings I

This query returns all suppliers from the United States delivering parts that are cheaper and bigger than the average of parts of the same type.

Original query

```
SELECT s_suppkey, s_name
FROM Supplier, Nation
WHERE s_nationkey = n_nationkey
        AND n_name = 'UNITED STATES'
        AND s_suppkey IN (
                SELECT ps_suppkey
                FROM PartSupp
                WHERE ps_partkey IN (
                        SELECT p_partkey
                        FROM Part P1
                        WHERE P1.p_retailprice < (
                                SELECT AVG(P2.p_retailprice)
                                FROM Part P2
                                WHERE P2.p_type = P1.p_type
                        ) AND P1.p_size > (
                                SELECT AVG(P3.p_size)
                                FROM Part P3
                                WHERE P3.p_type = P1.p_type
                        )
                )
        )
```

Transformed query

```
SELECT DISTINCT s_suppkey, s_name
FROM Supplier, Nation, PartSupp, Part P1,
     (SELECT p_type, AVG(p_retailprice) AS avg_price
                     AVG(p_size) AS avg_size
     FROM Part
     GROUP BY p_type) P2
WHERE s_nationkey = n_nationkey
     AND n_name='UNITED STATES'
     AND s_suppkey = ps_suppkey
     AND ps_partkey=P1.p_partkey AND P1.p_type = P2.p_type
     AND P1.p_retailprice < avg_price
     AND P1.p_size > avg_size
```

Query (k): Multiple Nestings II

This query returns all parts delivered by a supplier from the United States that are cheaper and bigger than the average of parts of the same type. In addition, the price of a part has to be less than the price of the most expensive orders with urgent, high and medium priority.

```
SELECT s_name, p_name
FROM Part P1, Supplier, Nation
WHERE s_nationkey = n_nationkey
```

```
    AND n_name = 'UNITED STATES' AND p_partkey IN (
        SELECT ps_partkey
        FROM PartSupp
        WHERE ps_suppkey = s_suppkey
    ) AND P1.p_retailprice > (
        SELECT AVG(P2.p_retailprice)
        FROM Part P2
        WHERE P2.p_type = P1.p_type
    ) AND P1.p_retailprice < ALL (
        SELECT o_totalprice
        FROM Orders
        WHERE o_totalprice >= (
            SELECT MAX(o_totalprice)
            FROM Orders
            WHERE o_orderpriority = '1-URGENT'
        ) AND o_totalprice >= (
            SELECT MAX(o_totalprice)
            FROM Orders
            WHERE o_orderpriority = '2-HIGH'
        ) AND o_totalprice >= (
            SELECT MAX(o_totalprice)
            FROM Orders
            WHERE o_orderpriority = '3-MEDIUM'))
```

Transformed query

```
SELECT DISTINCT s_name, p_name
FROM Part P1, Supplier, Nation, PartSupp, (
    SELECT p_type, AVG(p_retailprice) AS avg_price
    FROM Part
    GROUP BY p_type
) P2
WHERE s_nationkey = n_nationkey
    AND n_name='UNITED STATES'
    AND P1.p_partkey = ps_partkey AND ps_suppkey = s_suppkey
    AND P1.p_type = P2.p_type AND P1.p_retailprice > avg_price
    AND P1.p_retailprice < ALL
        (SELECT o_totalprice
         FROM Orders
         WHERE o_totalprice >=
            (SELECT MAX(o_totalprice)
             FROM Orders
             WHERE o_orderpriority='1-URGENT'
                OR o_orderpriority='2-HIGH'
                OR o_orderpriority='3-MEDIUM'))
```

B. OTHER SQL CONCEPTS

B.1 CASE Statements

CASE statements can be represented with a projection containing a selection as subexpression in its projection list. The following example shows a SQL query with a CASE statement and the equivalent NF^2 expression.

Example B.1.1 *The following query returns for each part type the number of parts that are larger than a given size.*

```
SELECT p_type, COUNT(CASE WHEN p_size > 100 THEN 1 END)
FROM Part
GROUP BY p_type
```

The equivalent NF^2 expression is as follows.

$$\pi[p\_type, \text{COUNT}(\sigma[p\_size > 100 \wedge p\_type' = p\_type](Part'))](Part)$$

B.2 Window Functions

Window functions can be represented by combining nested selections with projections.

Definition B.2.1 (Window Function) *Let R be a relation, $L \subseteq attr(R)$ and $A = \{A_1, ..., A_k\} \subseteq attr(R)$ sets of attributes, f an aggregation function and R' the renamed relation of R. In addition, the following query is given.*

```
SELECT L, f(B) OVER(PARTITION BY A)
FROM R
```

This query can be represented by the NF^2 expression

$$\pi[L, \omega[A; f(B)](R')](R).$$

The window function ω inside the projection is defined by:

$$\omega[A; f(B)](R') := f(\pi[B](\sigma[A'_1 = A_1 \wedge ... \wedge A'_k = A_k](R'))).$$

C. EXAMPLE PROOF

In this subsection, we demonstrate how the correctness of the subquery coalescing rules defined in Section 4 can be proven. As an example, we proove the correctness of the following case from Definition 4.2.1.

$$\begin{aligned}&\sigma[(A > \text{MAX}(\pi[B](\sigma[F_1](Inner))) \wedge \\ &\quad (A > \text{MAX}(\pi[B](\sigma[F_2](Inner))))](Outer) \\ &\equiv \sigma[A > \text{MAX}(\pi[B](\sigma[F_1 \vee F_2](Inner)))](Outer)\end{aligned}$$

Proof. Consider the expressions

$$\begin{aligned}E_1 := \sigma[(A > \text{MAX}(\pi[B](\sigma[F_1](Inner))) \\ \wedge (A > \text{MAX}(\pi[B](\sigma[F_2](Inner))))](Outer)\end{aligned}$$

and $E_2 := \sigma[A > \text{MAX}(\pi[B](\sigma[F_1 \vee F_2](Inner)))](Outer)$.

In order that $E_1 \equiv E_2$ is vaild, $sch(E_1) = sch(E_2)$ and $val(E_1) = val(E_2)$ have to hold (*i.e.*, both expressions have the same schema and return the same tuples). For both expressions the outer operator is a selection on the relation "Outer". For that reason, $sch(E_1) = sch(E_2)$ holds. As a next step, we have to show that $val(E_1) = val(E_2)$ holds. W.l.o.g. m_1 is the maximum of the first subexpression of E_1 and m'_1 the maximum of the second subexpression of E_1. Additionally, m_2 is the maximum of the subexpression of E_2.

- $val(E_1) \subseteq val(E_2)$: Let $t \in val(E_1)$ be a tuple. Then $t(A) > m_1$ and $t(A) > m'_1$ have to hold (t(A) is the value of tuple t in attribute A). Since $m_1 \in val(\pi[B](\sigma[F_1](Inner)))$ and $m'_1 \in val(\pi[B](\sigma[F_2](Inner)))$ hold, m_1, $m'_1 \in val(\pi[B](\sigma[F_1 \vee F_2](Inner)))$ also have to hold. As a consequence, $m_2 = max\{m_1, m'_1\}$ and $t(A) > m_2$ follow. Hence, $t \in val(E_2)$ holds. Consequently, $val(E_1) \subseteq val(E_2)$ holds.
- $val(E_2) \subseteq val(E_1)$: Let $t \in val(E_2)$ be a tuple. Then $t(A) > m_2$ has to hold. Since $m_2 = max\{val(\pi[B](\sigma[F_1 \vee F_2](Inner)))\}$ holds, it follows that $m_2 = max\{m_1, m'_1\}$ holds. Consequently, $t(A) > m_1$ and $t(A) > m'_1$ hold. Therefore, $t \in val(E_1)$ is fulfilled and $val(E_2) \subseteq val(E_1)$ holds.

□

The other cases of Definition 4.2.1 can be proven analogously.