Avoiding Materialisation for Guarded Aggregate Queries

Matthias Lanzinger
matthias.lanzinger@tuwien.ac.at
TU Wien
Vienna, Austria

Reinhard Pichler
reinhard.pichler@tuwien.ac.at
TU Wien
Vienna, Austria

Alexander Selzer
alexander.selzer@tuwien.ac.at
TU Wien
Vienna, Austria

ABSTRACT

Optimising queries with many joins is known to be a hard problem. The explosion of intermediate results as opposed to a much smaller final result poses a serious challenge to modern database management systems (DBMSs). This is particularly glaring in case of analytical queries that join many tables but ultimately only output comparatively small aggregate information. Analogous problems are faced by graph database systems when processing analytical queries with aggregates on top of complex path queries.

In this work, we propose novel optimisation techniques, both on the logical, and physical level, that allow us to avoid the materialisation of join results for certain types of aggregate queries. The key to these optimisations is the notion of *guardedness*, by which we impose restrictions on the occurrence of attributes in `GROUP BY` clauses and in aggregate expressions. The efficacy of our optimisations is validated through their implementation in Spark SQL and extensive empirical evaluation on various standard benchmarks.

PVLDB Reference Format:
Matthias Lanzinger, Reinhard Pichler, and Alexander Selzer. Avoiding Materialisation for Guarded Aggregate Queries. PVLDB, 18(5): 1398 - 1411, 2025.
doi:10.14778/3718057.3718068

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dbai-tuw/spark-eval.

1 INTRODUCTION

As the amounts of data to be processed increase, the limitations of established query evaluation methods become apparent. While modern DBMSs, such as Spark SQL, provide powerful frameworks for processing massive datasets, they may still struggle with many different types of complex queries. A key issue is the potential explosion of intermediate results, even if the final output is much smaller. This is particularly glaring in the context of analytical queries that combine data from many tables but ultimately produce comparatively small aggregate results. Analogous problems are faced by graph database systems when processing analytical queries with aggregates on top of complex path queries.

Traditionally, database engines try to avoid expensive intermediate blow-up by optimising the order in which joins are processed. More recently, worst-case optimal join techniques, which limit the blow-up to the theoretical worst-case, have gained popularity as an alternative approach for reducing intermediate materialisation. However, while these techniques may help to *alleviate* the problem of (unnecessarily) big intermediate results in certain cases, they do not *eliminate* the problem [6]. Furthermore, the problem of big intermediate results holds all the same even if joins are made only along foreign-key relationships [38].

For queries that exhibit certain favourable structural properties, Yannakakis [58] showed that it is possible to avoid the materialisation of unnecessary intermediate results. More specifically, if a query is acyclic – that is, if it has a join tree (formal definitions of acyclicity and join trees will be provided in Section 2) – then one can eliminate all dangling tuples (i.e., tuples not contributing to the final join result) via semi-joins before the actual join computation starts. However, even if dangling tuples have been eliminated and joins are evaluated in an optimal order, the intermediate results thus produced may still become prohibitively big. Especially in aggregate queries, where only a restricted amount of information is ultimately output, it would be highly desirable to reduce or, ideally, avoid altogether the materialisation of intermediate join results.

Actually, it is well known that, in case of Boolean acyclic queries (e.g., if we are only interested whether the result of a join query is non-empty), the final answer can be determined by carrying out only semi-joins and skipping the entire join step. So-called 0MA (= zero-materialisation aggregate) queries have recently been identified as a special class of queries with `MIN` or `MAX` aggregates over acyclic join queries, which can be evaluated without materialising any (intermediate or final) join results [14, 15] (a formal definition of 0MA queries is given in Section 2). Several works [13, 47] investigated how variations of the same algorithmic idea also apply to join queries with `COUNT` aggregates. Subsequently, these ideas were extended to more general aggregate queries in the FAQ-framework (Functional Aggregate Queries) [32] and, similarly, under the name AJAR (Aggregations and Joins over Annotated Relations) [31]. We offer a more detailed account of related work in Section 3.

However, previous works in this area have left a gap: Most approaches (such as FAQ and AJAR mentioned above) aim at *reducing* (not eliminating) the number of joins and/or the cost of computing them by applying sophisticated join techniques. But the computation and materialisation of joins remains the dominating cost factor. On the other hand, approaches that avoid the computation or materialisation of intermediate join results depend on severe restrictions of the class of queries, such as Boolean queries or 0MA queries. Indeed, as we will show in our empirical evaluation in Section 6, only a small fraction of the queries in the standard benchmarks considered here satisfies these restrictions.

The *goal of this work* is to identify a class of aggregate queries which can be evaluated without the need to compute or materialise any joins and which, nevertheless, cover many practical cases.

Proceedings of the VLDB Endowment, Vol. 18, No. 5 ISSN 2150-8097.
doi:10.14778/3718057.3718068

```
SELECT MEDIAN(s_acctbal)
FROM part, partsupp, supplier,
  nation, region
WHERE p_partkey   = ps_partkey
  AND s_suppkey   = ps_suppkey
  AND n_nationkey = s_nationkey
  AND r_regionkey = n_regionkey
  AND p_price >
   (SELECT avg (p_price) FROM part)
AND r_name IN ('Europe', 'Asia')
```

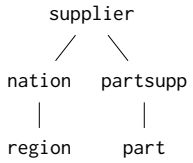


Figure 1: Query over the TPC-H schema and its corresponding join tree

The key to this class of queries is the notion of *guardedness*. More specifically, we call a query with aggregates on top of an acyclic join query *guarded* if all attributes involved in a GROUP BY clause and in any aggregate expression are contained in a single relation, referred to as the "*guard*". Here we allow any aggregate function from the ANSI SQL standard – including statistical functions such as MEDIAN, VARIANCE, STDDEV, CORR, etc.

Example 1.1. *We illustrate the basic ideas with the simple query given in Figure 1 over the well-known TPC-H schema. The query asks for the median of the account balance of suppliers from one of the regions 'Europe' or 'Asia' for parts with above average price. The join-structure of the query is clearly acyclic as is witnessed by the join tree displayed in Figure 1. Moreover the query is trivially* guarded, *since it has no grouping and the aggregation is over a single attribute.*

Note that the subquery is only used to realise a selection (locally) on the relation part. *After applying this selection on the* part *relation and also the selection on the* region *relation, the query can be evaluated by* propagating frequencies *of attribute value combinations rather than intermediate join results up the join tree. The* MEDIAN*-aggregate can then be evaluated on the resulting relation at the root node. Indeed, suppose that we have computed all tuples $t_1, \ldots, t_n$ of relation* supplier *together with the corresponding frequencies $c_1, \ldots, c_n$ of these tuples in the full join result of the five relations. Then we can order the values $v_1, \ldots, v_n$ of these tuples in ascending order and, by taking the frequencies $c_1, \ldots, c_n$ into account, it is an easy task to read off the median value. This is in sharp contrast to traditional query evaluation, which would first compute the* join of the five relations *and evaluate the aggregate on the full join result.* ⋄

As we will see in Section 6, all of the queries in the STATS-CEB and SNAP benchmarks are thus covered and so is a small number of queries in the other benchmarks studied here. However, for the most commonly used aggregate functions MIN, MAX, COUNT, SUM, and AVG, the guardedness restriction can be significantly relaxed. We thus define *piecewise-guarded* queries as queries with aggregates on top of an acyclic join query, where the attributes in a GROUP BY clause and the attributes jointly occurring in an aggregate expression *each* are contained in a single relation. That is, the GROUP BY clause and also each aggregate expression has a *guard*, but all these guards may be different. It will turn out that, with this relaxed restriction, we can cover all JOB queries and a significant number of queries in the TPC-H and TPC-DS benchmarks.

We will show how to realise Yannakakis-style evaluation for guarded and piecewise-guarded queries by rewriting subtrees in the logical query plan. In this process, joins are either replaced by semi-joins or they are immediately followed by aggregation. Importantly, in the latter case, the number of tuples to be propagated up the join tree is the same as in case of semi-joins. The only extension needed is to add columns for the total frequencies (corresponding to COUNT(*)) and for the various aggregate expressions contained in the query. All of this can, in a very natural way, be implemented as part of the logical optimisation step. This approach thus applies also to subqueries and automatically works in conjunction with other optimisation techniques such as subquery decorrelation. An additional benefit of this optimisation is that it requires no cost-based optimisation, making it particularly attractive for systems with a limited or no cost model such as Spark SQL.

As a further optimisation, we introduce a new physical operator that intuitively implements a semi-join that keeps track of frequencies and other aggregate values, and which can be implemented through minimal changes to standard join algorithms (see Section 5). It thus integrates smoothly in any typical SQL execution engine. We have implemented both, the logical optimisation and the new physical operator in Spark SQL, which was specifically designed to cope with complex analytical queries. The performance gain observed in our experimental evaluation on several standard benchmarks can reach up to one or two orders of magnitude for analytical queries involving aggregates on top of non-trivial join or path queries. Notably, our method incurs no performance degradation even for simple queries where the size of intermediate results never gets too big anyway.

In summary, our main contributions are as follows:

- We introduce the class of *guarded* aggregate queries and show that they allow for an evaluation without materialising any intermediate join result for queries involving GROUP BY and any of the aggregate functions contained in the ANSI SQL standard. We then relax the restrictions imposed by guardedness and introduce the class of *piecewise-guarded* aggregate queries. For these queries, we show that the favourable property of avoiding the propagation of any join results applies to the most commonly used aggregate functions, namely MIN, MAX, COUNT, SUM, and AVG.
- We achieve an additional optimisation of guarded and piecewise-guarded aggregate queries by introducing a *novel physical operator* that allows us to evaluate aggregate expressions and to compute frequencies of attribute combinations by an appropriate extension of the semi-join operator.
- We have implemented our logical and physical optimisations into Spark SQL and we have carried out an extensive empirical evaluation based on several standard benchmarks. It turns out that these benchmarks contain a significant number of queries or subqueries from our newly defined query classes. Our experimental results clearly prove the efficacy of our optimisation techniques.

The rest of the paper is organised as follows. We start with preliminaries in Section 2. In Section 3, we recall several paths of related work. Our novel query optimisation techniques on logical query plans are presented in Section 4, and we discuss the new physical operator in Section 5. In Section 6, we report on our experimental evaluation, and we conclude with Section 7. Further details

(both on our optimisations and the experimental evaluation) are provided in the full paper [35].

2 PRELIMINARIES

The basic form of queries considered here are *Conjunctive Queries* (CQs), which correspond to select-project-join queries in the Relational Algebra. Consider a CQ of the form $Q = \pi_U(R_1 \bowtie \ldots \bowtie R_n)$. Here we assume that equi-joins are replaced by natural joins via appropriate renaming of attributes. Moreover, we assume that selections applying to a single relation have been pushed immediately in front of this relation and the R_i's are the result of these selections. The R_i's are not necessarily distinct and our results are not affected by self-joins.

Such a CQ is called *acyclic* (an ACQ, for short), if it has a *join tree*, i.e., a rooted, labelled tree $\langle T, r, \lambda\rangle$ with root r and node-labelling function λ such that (1) for every relation R_i there exists exactly one node u of T with $\lambda(u) = R_i$ and (2) λ satisfies the so-called connectedness condition, i.e., if some attribute A occurs in both relations $\lambda(u)$ and $\lambda(v)$ for two nodes u, v of T, then A occurs in the relation $\lambda(w)$ for every node w on the path between u and v. Checking if a CQ is acyclic and, if so, constructing a join tree, can be done in linear time w.r.t. the size of the query by the so-called "GYO reduction" algorithm [19][59].

Yannakakis [58] has shown that ACQs can be efficiently evaluated (that is, essentially, linear w.r.t. the input+output data and linear w.r.t. the size of the query) via 3 traversals of the join tree: (1) a bottom-up traversal of semi-joins, (2) a top-down traversal of semi-joins, and (3) a bottom-up traversal of joins. Formally, let u be a node in T with child nodes $u_1, \ldots, u_k$ of u and let relations R, $R_{i_1}, \ldots, R_{i_k}$ be associated with the nodes $u, u_1, \ldots, u_k$ at some stage of the computation. Then we set

(1) $R = (((R \ltimes R_{i_1}) \ltimes R_{i_2}) \cdots) \ltimes R_{i_k}$,
(2) $R_{i_j} = R_{i_j} \ltimes R$ for every $j \in \{1, \ldots, k\}$, and
(3) $R = (((R \bowtie R_{i_1}) \bowtie R_{i_2}) \cdots) \bowtie R_{i_k}$

in the 3 traversals (1), (2), and (3). The final result of the query is the resulting relation associated with the root r of T. Following the SQL-standard, we are assuming bag semantics for the queries. Note however that Yannakakis' algorithm can be applied to both, set semantics and bag semantics.

In this work, we are mainly interested in queries that apply aggregates on top of ACQs and that may contain "arbitrary" selections applied to single relations (that is, not only equality conditions, as is usually assumed for CQs [2]). Moreover, we allow grouping, which can also take care of the projection. In other words, we are interested in queries of the form

$$Q = \gamma[g_1, \ldots, g_\ell,\ A_1(a_1), \ldots, A_m(a_m)]\big(R_1 \bowtie \cdots \bowtie R_n\big) \quad (1)$$

where $\gamma_{g_1,\ldots,g_\ell,\ A_1(a_1),\ldots,A_m(a_m)}$ denotes the grouping operation for attributes $g_1, \ldots, g_\ell$ and aggregate expressions $A_1(a_1), \ldots, A_m(a_m)$ for some (standard SQL) aggregate functions $A_1, \ldots, A_m$ applied to expressions $a_1, \ldots, a_m$. The grouping attributes $g_1, \ldots, g_\ell$ are attributes occurring in the relations $R_1, \ldots, R_n$ and $a_1, \ldots, a_m$ are expressions formed over the attributes from $R_1, \ldots, R_n$. A simple query of the form shown in Equation (1) is given in Figure 1 (in SQL-syntax), together with a possible join tree of this query.

In the sequel, it will be convenient to use the following notation: suppose that we want to assign the result of a query Q of the form according to Equation (1) to a relation S with attributes $g_1, \ldots, g_\ell, C_1, \ldots, C_m)$, such that the values of each aggregate expression $A_i(a_i)$ is assigned to the attribute C_i, then we will write

$$S := \quad \gamma[g_1, \ldots, g_\ell,\ C_1 \leftarrow A_1(a_1), \ldots, C_m \leftarrow A_m(a_m)]\big(R_1 \bowtie \cdots \bowtie R_n\big) \quad (2)$$

Recently [15], a particularly favourable class of ACQs with aggregates has been presented: the class of 0MA (short for "zero-materialisation answerable") queries. These are acyclic queries that can be evaluated by executing only the first bottom-up traversal of Yannakakis' algorithm. That is, we only need to perform the comparatively cheap semi-joins and can completely skip the typically significantly more expensive join phase. A query of the form given in Equation (1) is 0MA if it satisfies the following conditions:

- *Guardedness*, meaning that there exists a relation R_i that contains all grouping attributes $g_1, \ldots, g_\ell$ and all attributes occurring in the aggregate expressions $A_1(a_1), \ldots, A_m(a_m)$. Then R_i is called the *guard* of the query. If several relations satisfy this property, we arbitrarily choose one guard.
- *Set-safety*: we call an aggregate function *set-safe*, if its value over any set S of tuples remains unchanged if duplicates are eliminated from S. A query satisfies the set-safety condition, if all its aggregate functions $A_1 \ldots, A_m$ are set-safe.

The root of the join tree can be arbitrarily chosen. Hence, we may assume that the root is labelled by the guard and, therefore, all relevant attributes are contained in the root node. Note that the bottom-up semi-join traversal makes sure that all value combinations of the attributes in the root node indeed occur in the answer tuples of the inner part $R_1 \bowtie \cdots \bowtie R_n$ of the query. Set-safety of an aggregate function A_i means that multiplicities do not matter for the evaluation of A_i. Hence, if all aggregate functions are set-safe, then we can apply the grouping and aggregation $\gamma[g_1, \ldots, g_\ell,\ A_1(a_1), \ldots, A_m(a_m)]$ right after the first bottom-up traversal. In SQL, the `MIN` and `MAX` aggregates are inherently set-safe. Moreover, an aggregate becomes set-safe when combined with the `DISTINCT` keyword. For instance, `COUNT DISTINCT` is clearly a set-safe aggregate function. Note that the query in Figure 1 is trivially guarded (i.e., there is no grouping and the only aggregate expression is over the single attribute `s_acctbal`) but not set-safe, since multiplicities clearly matter for the evaluation of the `MEDIAN` function.

3 RELATED WORK

Acyclic queries. The algorithm for evaluating acyclic queries, presented by Yannakakis over 40 years ago [58], has long been central to the *theory* of query processing. In recent years, this approach to query evaluation has gained renewed momentum in *practice* as evidenced by several extensions and applications. Multiple recent works [26, 27, 55], propose extensions of Yannakakis' algorithm for dynamic query evaluation. Further research extends and applies Yannakakis' algorithm to comparisons spanning several relations [56], queries with theta-joins [27], and privacy preserving

query processing [57]. An important feature of Yannakakis' algorithm is the elimination of *dangling tuples* (i.e., tuples that do not contribute to the final result) via semi-joins. In a recent paper [25], a new join method was introduced that integrates the detection and elimination of dangling tuples into the join computation.

Decompositions. An important line of research has extended the applicability of Yannakakis-style query evaluation to "almost acyclic" queries. Here, "almost acyclic" is formalised through various notions of decompositions such as (normal, generalised, or fractional) hypertree decompositions [3, 17, 22]. Each of these decompositions is associated with a notion of "width" that measures the distance from acyclicity, with acyclic queries having a width of 1. Several works [1, 12, 46, 54], combine Yannakakis-style query evaluation based on various types of decompositions with multiway joins and worst-case optimal join techniques.

Aggregate queries. Aggregates are commonly used on top of join queries – especially in data analytics. Green et al. [21] gave a new perspective on aggregate queries by by considering K-relations, i.e., relations annotated with values from some semi-ring K. Join queries over K-relations then come down to evaluating sum-product expressions over the underlying semi-ring. The combination of aggregate queries with Yannakakis-style query evaluation was studied in the FAQ-framework (Functional Aggregate Queries) [32] and, similarly, under the name AJAR (Aggregations and Joins over Annotated Relations) [31]. A crucial problem studied in both papers is the interplay between the ordering of a sequence of aggregate functions and (generalised or fractional) hypertree decompositions. In both papers, the ultimate goal is an efficient, Yannankakis-style evaluation algorithm for aggregate queries based on finding a good variable order. Similar ideas to FAQs and AJAR queries also appear in earlier works on joins and aggregates over factorised databases [7, 45] and on quantified conjunctive queries (QCQs) [11]. A general framework for hybrid database and linear algebra workloads (as are typical for machine learning applications) has recently been proposed by Shaikhha et al. [50]. It provides a performant, unified framework for data science pipelines by introducing the purely functional language SDQL and combining optimisation techniques from databases (e.g., pushing aggregates past joins) and linear algebra (e.g., matrix chain ordering).

Distributed query processing. The potential of applying Yannakakis-style query evaluation to distributed processing comes from the fact that the evaluation of ACQs lies in the highly parallelisable class LogCFL [16]. This favourable property was later extended to "almost acyclic" queries by establishing the LogCFL-membership also for queries with bounded hypertree width [17]. A realisation of Yannakakis' algorithm in MapReduce [4] further emphasised the parallelisability of Yannakakis-style query evaluation.

Spark and Spark SQL. Spark, a top-level Apache project since 2014, is often regarded as a further development of the MapReduce processing model. Spark SQL [5] provides relational query capability within the Spark framework. Query optimisation is a primary focus of Spark SQL, with the powerful Catalyst optimiser being an integral component since its inception [5]. Several later works [8, 28, 40, 51, 60]) have proposed further measures to speed up query processing in Spark SQL. The recently presented SparkSQL$^+$ system [12] combines decompositions and worst-case optimal join techniques as well as the optimisations for CQs with comparisons spanning several relations [56] and allows users to experiment with different query plans. Zhang et al. [61] recently implemented specific worst-case optimal join algorithms in combination with decomposition-based methods on top of Spark SQL as part of a system focused specifically on subgraph counting.

Reducing the number of join computations. Several works have addressed the need to compute a high number of joins in different contexts and have aimed at reducing this number. The work by Schleich et al. [48] on LMFAO (Layered Multiple Functional Aggregate Optimization) specifically targets machine learning applications that require the computation of large batches of aggregate queries over the same join query. A dramatic speed-up is achieved by decomposing aggregates into views and arranging them at nodes in a join tree to avoid the re-computation of the same intermediate joins time and again. In principle, the need to re-compute similar joins time and again also arises in the area of IVM (incremental view maintenance). A revolutionary approach to IVM was proposed by Koch et al. [34] with the DBToaster system, that avoids the re-computation of joins in case of updates to the database by maintaining "higher-order" delta views, i.e., delta queries (= first-order deltas), delta queries to the deltas (= second-order deltas), etc.. A further performance gain is achieved with F-IVM (factorised IVM) [43], that groups various aggregates together and thus reduces the number of views to be maintained. Moreover, factorisation is applied, for instance, to avoid the materialisation of Cartesian products in views. Of course, independently of IVM, factorisation [44] is a generally applicable method to keep the query result in a compressed form and avoid its complete materialisation.

In summary, a lot of work has been done on optimising aggregate queries – including worst-case optimal join techniques (primarily targeting cyclic queries) and Yannakakis-style query evaluation (for acyclic or almost acyclic queries). There also have been very successful approaches to *reduce* the number of joins that have to be computed when processing batches of related aggregate queries. Higher-order incremental view maintenance aims at *completely avoiding* the need to (re-) compute joins in case of updates to the database. Our goal in this work is also to avoid the need to compute joins – applicable to ad hoc query answering rather than IVM. To this end, we identify a class of queries with aggregates on top of joins that can be evaluated without actually computing the result of any joins and that covers many relevant cases. To the best of our knowledge, apart from some severely restricted cases (specifically, Boolean queries and 0MA queries), this has not been the focus of previous work.

4 RULE-BASED OPTIMISATIONS

In Section 2, we have recalled the definition of 0MA (zero-materialisation answerable) queries from Gottlob et al. [15]. Queries in this class, which have to satisfy the set-safety and guardedness conditions, can be evaluated by rooting the join tree at the node labelled by the guard and then executing the first bottom-up traversal of Yannakakis' algorithm. This means, that all joins are replaced by semi-joins. The grouping and aggregation can then be evaluated by considering only the resulting relation at the root node.

However, the set-safety condition is quite restrictive in that it is only satisfied by a small number of aggregate functions – primarily MIN, MAX, and COUNT DISTINCT. The vast majority of aggregate functions – in particular, COUNT (without DISTINCT), SUM, AVG, and the entire collection of statistical aggregate functions provided by the ANSI SQL standard are thus disallowed. For instance, the query in Figure 1 involving the MEDIAN aggregate is not 0MA.

In this section, we significantly extend the class of queries with aggregates on top of join queries that can be evaluated without actually materialising any joins. To this end, we will first drop the set-safety condition in Section 4.1 and then also introduce a relaxation of guardedness in Section 4.2. To emphasise the smooth integration of our optimisations into standard SQL execution technology, we will describe our optimisations in the form of equivalence-preserving transformations of Relational Algebra subexpressions, which can be applied anywhere in the logical query plan.

4.1 Guarded Aggregate Queries

In order to cover *all* aggregate functions of the ANSI SQL standard, we now drop the safety condition and define the class of guarded aggregate queries as follows:

Definition 4.1. Let Q be a query of the form given in Equation (1), i.e., $Q = \gamma[g_1, \ldots, g_\ell,\ A_1(a_1), \ldots, A_m(a_m)]\big(R_1 \bowtie \cdots \bowtie R_n\big)$. We call Q a *guarded aggregate query* (or simply, "guarded query"), if $(R_1 \bowtie \cdots \bowtie R_n)$ is acyclic and there exists a relation R_i (= the *guard*) that contains all attributes that are either part of the grouping or occur in one of the aggregate expressions. If several relations have this property, we arbitrarily choose one as the guard.

Note that we consider an aggregate expression COUNT(*) as trivially guarded, since it contains no attributes at all. We will now show that, for any aggregate functions of the ANSI SQL standard, guarded queries can be evaluated without propagating any join results up the join tree. To this end, we revisit an extension of Yannakakis' algorithm by Pichler and Skritek [47] to acyclic queries with a COUNT(*) aggregate on top. We adapt this approach to integrate it into the logical query plan of relational query processing and we further extend it to all other aggregate functions.

The crucial idea for evaluating a query Q of the form given in Equation (1) is to propagate frequencies up the join tree rather than duplicating tuples. It is convenient to introduce the following notation: let u denote a node in the join tree T and let T_u denote the set of all nodes in the subtree rooted at u. Moreover, for any node u in T, we write $R(u)$ to denote the relation labelling node u and we write $Att(u)$ to denote the list of attributes of $R(u)$. The goal of the bottom-up propagation of frequencies is to compute, for every node u in T, the result of the following query:

$$\gamma[Att(u), \texttt{COUNT(*)}]\left(\bowtie_{v \in T_u} R(v)\right) \tag{3}$$

This propagation is realised by recursively constructing extended Relational Algebra expressions $Freq(u)$ for every node u of the join tree, such that $Freq(u)$ gives the same result as the query in Equation (3). Hence, $Freq(u)$ has as attributes all attributes of $R(u)$ plus one additional attribute (which we will denote as c_u), where we store frequency information for each tuple of $R(u)$. If u is a leaf node of the join tree, then we initialise the attribute c_u to 1. Formally, we thus have $Freq(u) = R(u) \times \{(1)\}$.

Now consider an internal node u of the join tree with child nodes $u_1, \ldots, u_k$. The extended Relational Algebra expression $Freq(u)$ is constructed iteratively by defining subexpressions $Freq_i(u)$ with $i \in \{0, \ldots, k\}$. To avoid confusion, we refer to the frequency attribute of such a subexpression $Freq_i(u)$ as c_u^i. That is, each relation $Freq_i(u)$ consists of the same attributes $Att(u)$ as $R(u)$ plus the additional frequency attribute c_u^i. Then we define $Freq_i(u)$ for every $i \in \{0, \ldots, k\}$ and, ultimately, $Freq(u)$ as follows:

$$\begin{aligned} Freq_0(u) &:= R(u) \times \{(1)\} \\ Freq_i(u) &:= \gamma[Att(u), c_u^i \leftarrow \texttt{SUM}(c_u^{i-1} \cdot c_{u_i})](Freq_{i-1}(u) \bowtie Freq(u_i)) \\ Freq(u) &:= \rho_{c_u \leftarrow c_u^k}(Freq_k(u)) \end{aligned}$$

Intuitively, after initialising c_u^0 to 1 in $Freq_0(u)$, the frequency values $c_u^1, \ldots, c_u^k$ are obtained by grouping over the attributes $Att(u)$ of $R(u)$ and computing the number of possible extensions of each tuple $t \in R(u)$ to the relations labelling the nodes in the subtrees rooted at $u_1, \ldots, u_k$. By the connectedness condition of join trees, these extensions are independent of each other, i.e., they share no attributes outside $Att(u)$. Moreover, the frequency attributes $c_u^1, \ldots, c_u^k$ are functionally dependent on the attributes $Att(u)$. Hence, by distributivity, the value of c_u^k obtained by iterated summation and multiplication for given tuple t of $R(u)$ is equal to computing, for every $i \in \{1, \ldots, k\}$ the sum s_i of the frequencies of all join partners of t in $Freq(u_i)$ and then computing their product, i.e., $c_u = c_u^k = \Pi_{i=1}^k s_i$.

In the logical query plan of query Q, we replace the subexpression corresponding to the join query $R_1 \bowtie \cdots \bowtie R_n$ by $Freq(r)$, where r is the root node of the join tree. This root node was chosen in such a way that $R(r)$ contains all grouping attributes $g_1, \ldots, g_\ell$. Hence, the grouping can be applied to $Freq(r)$ in the same way as to the original join query. Also the set-safe aggregates (such as MIN, MAX, COUNT DISTINCT) can be applied to $Freq(r)$ "as usual" by simply ignoring the additional attribute c_r. However, all other (i.e., not set-safe) aggregate functions have to be replaced by variants that take the special frequency attribute c_r into account. We thus modify the aggregate functions in expressions like COUNT(*), COUNT(B), SUM(B), and AVG(B) so that they directly operate on tuples with frequencies. For instance, let B be an attribute of the guard $R(r)$ (and, hence, also of $Freq(r)$). Then, in SQL-notation, we can rewrite common aggregate expressions as follows:

- COUNT(*) $\rightarrow$ SUM(c_r)
- COUNT(B) $\rightarrow$ SUM(IF(ISNULL(B), 0, c_r))
- SUM(B) $\rightarrow$ SUM($B \cdot c_r$)
- AVG(B) $\rightarrow$ SUM($B \cdot c_r$)/COUNT(B)

Recall from Example 1.1 that the MEDIAN aggregate (like any other statistical function) can be evaluated by considering $Freq(r)$ as a compressed form of the list of all values of attribute B in each group, where the value of the additional attribute c_r indicates the number of copies of the corresponding value of attribute B in the result of the join query $R_1 \bowtie \cdots \bowtie R_n$. Evaluating an aggregate expression MEDIAN(B) or any other statistical function such as STDDEV(B) can be easily realised for this compressed form of value list. Similarly, the evaluation of aggregate functions on 2 attributes such as CORR(B_1, B_2) or aggregate expressions involving functions on several attributes such as SUM(f($B_1, \ldots, B_k$)) is straightforward by

considering $Freq(r)$ as a compressed form of the list of all values of the attribute combinations $B_1, \ldots, B_k$. Again, this crucially depends on the guardedness property, which guarantees that all attributes used in aggregate expressions are contained in $R(r)$.

Actually, in Spark SQL, the `MEDIAN` aggregate has a convenient rewriting via the `PERCENTILE` function. The latter is not part of the ANSI SQL standard, but can be found in Spark SQL. This function allows one to provide a frequency attribute, which Spark uses to build a map of values and frequencies, sort them, and finally find the desired percentile value by an efficient search on the sorted map. The rewriting of the `MEDIAN` aggregate looks as follows:

- $\texttt{MEDIAN}(A) \longrightarrow \texttt{PERCENTILE}(0.5, A, c_r)$

Example 4.1. *Consider again the query of Figure 1. The logical query plan generated by Spark SQL is shown in Figure 2a. There, we write σ_ψ and σ_ϕ to denote the selections applied to the relations* `region` *and* `part`*, respectively. That is, ψ checks the condition* `r_name IN ('Europe', 'Asia')` *and ϕ checks the condition* `p_price > (SELECT avg (p_price) FROM part)`*. The plan produced by Spark SQL including our optimisation is shown in Figure 2b.*

We observe that, in the unoptimised query plan, the entire join of all relations is computed before the `MEDIAN` *aggregate is applied. In contrast, in the optimised plan, only the additional frequency attribute has to be propagated upwards in the plan. This propagation of frequencies for each join is realised by 2 nodes in the plan directly above the node realising the join: first, as part of the projection to the attributes which are used further up in the plan, the frequency attributes of the two join operands are multiplied with each other. Here we use the notation c_{xy} when frequency attributes c_x and c_y are combined. In the second step, these frequency values c_{xy} are summed up or, in case of the final result, their median is computed, which can be further optimised by making use of the* `PERCENTILE` *function.* ◇

We conclude this section with an example where we display the information that has to be propagated in the optimised evaluation of the query from Figure 1. Actually, it is illustrative to first observe how the tree structure of the join tree is transformed into the tree structure of the optimised plan. Of course, in the latter, the relations must be at the leaf nodes, whereas, in the former, they also occur at inner nodes. Nevertheless, the bushy optimised plan clearly reflects the join order from the join tree. That is, first, `region` and `nation` are joined to get intermediate result-1, and `part` and `part_supp` are joined to get intermediate result-2. The join of these two intermediate results with the relation `supplier` is then split into two 2-way joins, i.e.: first joining `supplier` with result-1, which is then joined with result-2. Hence, for the sake of simplicity, we will discuss the evaluation of this query by looking at the relations at each node of the join tree. It is then clear, what the intermediate results at the nodes of the logical plan in Figure 2b look like.

Example 4.2. *Consider again the query from Figure 1. Note that all joins in this query are along foreign-key/primary-key relationship. This allows us additional optimisations, which we discuss in the full paper [35]. For the sake of illustration, let us ignore the primary keys for a while and allow multiple occurrences of values in these attributes. In Figure 3, we illustrate the evaluation of the query on a small sample database. The tables (with attribute names of the join attributes abbreviated to single characters) are arranged in the form of the join tree. Attributes not relevant to our discussion are captured by "…". The original contents of the tables is shown to the left of the "…" column. In the right-most column, we display the frequency attribute c for each tuple at the end of the entire bottom-up traversal. For instance, the* `Region` *table has 3 tuples with attribute value $R = r_1$. Hence, all tuples in* `Nation` *with $R = r_1$ have $c = 3$, i.e., the number of possible extensions to the subtree below. The tuple with value $R = r_4$ is deleted since it has no join partner below. In the root node, the sums of the frequency attributes of all join partners to the left and to the right are multiplied. For instance, the tuple with attribute values $N = n_1$ and $S = s_1$ has 5 possible extensions to the subtree on the left and 6 on the right. Hence, for this tuple, we get $c = 30$.*

For the evaluation of `MEDIAN(A)` *we see at the root node, that the first tuple has attribute value $A = 20$ and its frequency in the overall join result is $c = 30$. Likewise, the values $A = 40$, 10, 30 occur 20, 36, and 24 times, respectively in the join result. We thus get* `MEDIAN(A)` $= 20$. ◇

4.2 Piecewise-guarded Aggregate Queries

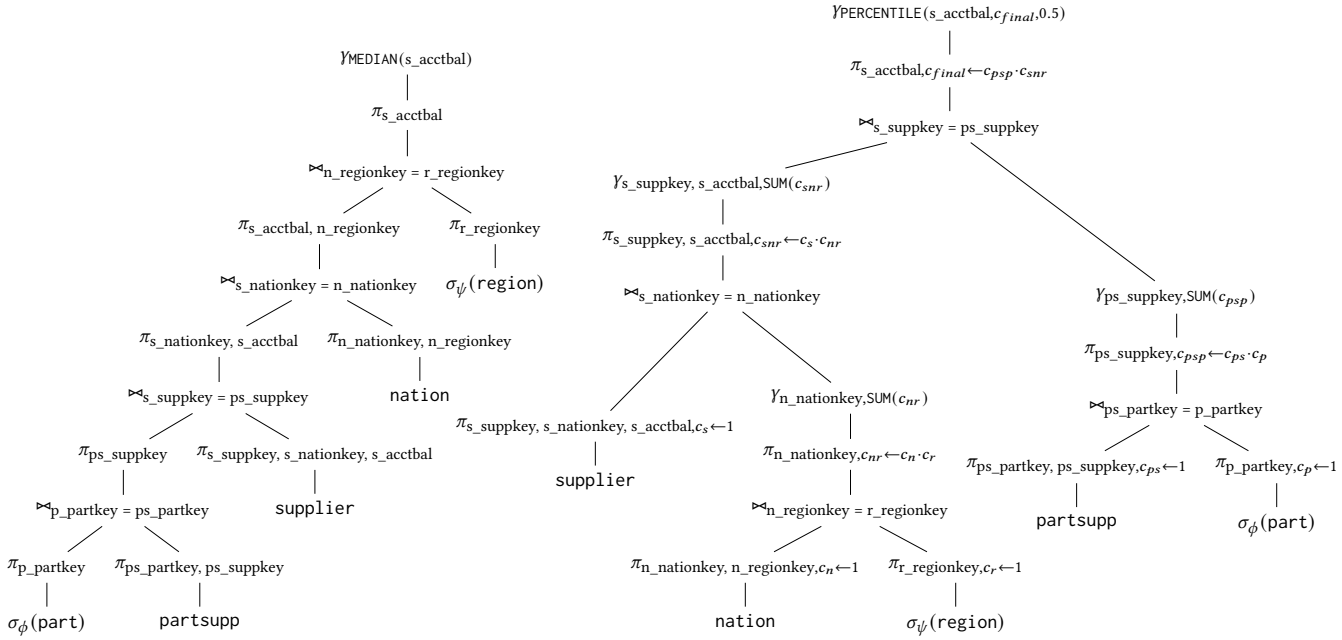
As we will see in our experimental evaluation in Section 6, the class of guarded queries covers significantly more cases from common benchmarks than 0MA. However, the requirement of a single guard for *all* attributes occurring in the `GROUP BY` clause or in any of the aggregate expressions is still quite restrictive. In this section, we show that for the most commonly used aggregate functions `MIN`, `MAX`, `SUM`, `COUNT`, and `AVG`, we can further extend the class of queries that can be evaluated without materialising any joins. We thus introduce the class of piecewise-guarded aggregate queries:

Definition 4.2. Let Q be a query of the form given in Equation (1), i.e., $Q = \gamma[g_1, \ldots, g_\ell,\ A_1(a_1), \ldots, A_m(a_m)]\big(R_1 \bowtie \cdots \bowtie R_n\big)$. We call Q a *piecewise-guarded aggregate query* (or simply, "piecewise-guarded query"), if $(R_1 \bowtie \cdots \bowtie R_n)$ is acyclic and there exists a relation R_{i_0} that contains all grouping attributes and, for every $j \in \{1, \ldots, m\}$, the following conditions hold:

- If $A_j \in \{\texttt{MIN}, \texttt{MAX}, \texttt{SUM}, \texttt{COUNT}, \texttt{AVG}\}$, then there exists a relation R_{i_j} that contains all attributes occurring in $A_j(a_j)$.
- Otherwise, i.e., $A_j \notin \{\texttt{MIN}, \texttt{MAX}, \texttt{SUM}, \texttt{COUNT}, \texttt{AVG}\}$, then R_{i_0} contains all attributes occurring in $A_j(a_j)$.

Each of these relations R_{i_0} and R_{i_j} is called the "guard" of the corresponding set of attributes. We refer to R_{i_0} as the *root* guard. By slight abuse of notation, we also refer to the nodes labelled by R_{i_0} and R_{i_j} as guards. If several relations could be chosen as guard for a group of attributes, we arbitrarily choose one.

For the evaluation of piecewise-guarded queries, we choose the node of the join tree T corresponding to the root guard as the root node of T. The bottom-up propagation of the frequency attribute works exactly as for guarded queries. Hence, also the evaluation of all aggregate expressions that are guarded by the root guard is realised exactly as in case of guarded queries. In the rest of this section, we concentrate on the evaluation of aggregate expressions that are *not guarded by the root node* of the join tree and whose aggregate function is one of `MIN`, `MAX`, `SUM`, `COUNT`, and `AVG`. Clearly, the evaluation of `AVG` is based on `SUM` and `COUNT`. Hence, it suffices to describe the evaluation of the remaining four aggregate functions.



(a) Query plan generated by Spark SQL

(b) Query plan generated by Spark SQL with rewritten aggregation

Figure 2: Query plans for Example 4.1

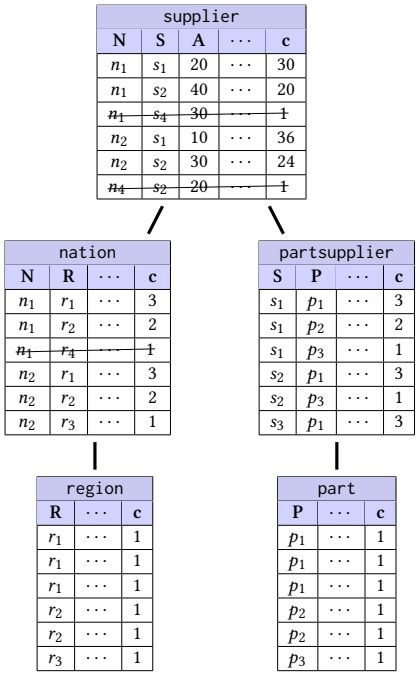


Figure 3: Evaluation of the query from Figure 1

Similarly to Koch et al. [34], we extend the relations by additional attributes to carry information on aggregate expressions. Below, we describe which information has to be propagated up the join tree in order to evaluate a single aggregate expression $A_j(a_j)$. For the evaluation of Q, we add the frequency attribute plus *all* these additional attributes to the corresponding nodes in the join tree.

Suppose that $A_j(a_j)$ is of the form $A_j(f_j(\bar{B}_j)$ with $A_j \in \{\texttt{MIN}, \texttt{MAX}, \texttt{SUM}, \texttt{COUNT}\}$ and f_j is an arbitrary function on attributes $\bar{B}_j$ jointly occurring in one of the relations $R_1, \ldots, R_n$. We choose as guard of the aggregate expression $A_j(a_j)$ the node w that contains all attributes $\bar{B}_j$ and that is highest up in the join tree T with this property. Since we are assuming that $A_j(a_j)$ is not guarded by the root node r of T, this means that w is different from r. Then we add to all relations along the path from w to r an additional attribute Agg_j. Analogously to Equation (3), the intended meaning of Agg_j for every node u on the path between w and r is as follows:

$$\gamma[Att(u), Agg_j \leftarrow A_j(f_j(\bar{B}_j)]\big(\bowtie_{v \in T_u}(R(v))\big), \tag{4}$$

For the *initialisation* of Agg_j, suppose that the frequency attribute of relation $R(w)$ at node w has already been computed as described in Section 4.1. Hence, in particular, $R(w)$ is restricted to the tuples t with positive frequency. For an arbitrary tuple t in $R(w)$, we write $t.c$, $t.Agg_j$, and $t.\bar{B}_j$ to denote the values of t at the frequency attribute c, at the aggregate attribute Agg_j, and at the attributes $\bar{B}_j$, respectively. Then we define $t.Agg_j$ as follows:

- If $A_j \in \{\texttt{MIN}, \texttt{MAX}\}$, then we set $t.Agg_j := f_j(t.\bar{B}_j)$.
- If $A_j = \texttt{COUNT}$, then we distinguish two cases: If $f_j(t.\bar{B}_j) = \texttt{NULL}$, then we set $t.Agg_j := 0$; otherwise $t.Agg_j := t.c$.
- If $A_j = \texttt{SUM}$, then we set $t.Agg_j := f_j(t.\bar{B}_j) * t.c$.

To verify that Agg_j is equal to the additional attribute according to Equation (4), we note that all tuples in a group defined by a

value combination of the original attributes $Att(w)$ of $R(w)$ (thus, corresponding to a single tuple $t \in R(w)$) coincide on the attributes $\bar{B}_j$. Hence, the MAX and MIN of $f_j(\bar{B}_j)$ over the tuples in such a group is simply the value of $f_j(t.\bar{B}_j)$. For $A_j \in \{\texttt{COUNT}, \texttt{SUM}\}$, we have to take the number of tuples in each such group into account, which corresponds to the frequency value $t.c$. For the COUNT aggregate, we also have to consider the special case that $f_j(\bar{B}_j) = \text{NULL}$, which means that $\texttt{COUNT}(f_j(\bar{B}_j))$ for the entire group is 0.

For the *propagation* of the additional attribute Agg_j along the path from w to the root r, consider an ancestor node u of w and let $u_1, \ldots, u_k$ denote the child nodes of u. W.l.o.g., we assume that the child node u_1 is on the path from w to r. Suppose that the frequency attribute at every child node u_i of u and the attribute Agg_j at node u_1 have already been computed. We are assuming that w is the highest node in the join tree that contains all attributes $\bar{B}_j$. Hence, u does not contain all attributes $\bar{B}_j$, and, by the connectedness conditions, neither does any of the nodes $u_2, \ldots, u_k$. For an arbitrary tuple $t \in R(u)$, let $\{t_1, \ldots, t_\alpha\}$ denote the set of all tuples in $R(u_1)$ that join with t. We compute the value $t.Agg_j$ as follows:

- First suppose that $A_j \in \{\texttt{MIN}, \texttt{MAX}\}$. Then we set $t.Agg_j := A_j(\{t_1.Agg_j, \ldots, t_\alpha.Agg_j\})$.
- Now let $A_j \in \{\texttt{SUM}, \texttt{COUNT}\}$. For every $i \in \{2, \ldots, k\}$, let s_i denote the sum of the frequencies of all join partners of t in $R(u_i)$. Then we set $t.Agg_j := \left(\sum_{\lambda=1}^{\alpha} t_\lambda.Agg_j[u_1]\right) * \Pi_{i=2}^{k} s_i$.

For the correctness of this propagation of attribute Agg_j, recall that we are assuming that the attributes $\bar{B}_j$ are not fully contained in the relation $R(u)$ and, hence, by the connectedness condition, they cannot be fully contained in any of the child nodes $\{u_2, \ldots, u_k\}$ either. Hence, the value combinations of $\bar{B}_j$ in $\left(\bowtie_{v \in T_u}(R(v)\right)$ must already occur in $\left(\bowtie_{v \in T_{u_1}}(R(v)\right)$. The MIN or MAX of $f_j(\bar{B}_j)$ of a tuple t when grouping over the attributes of $R(u)$ is, therefore, simply obtained by grouping over the attributes of $R(u_1)$ and aggregating over the join partners of t in $R(u_1)$. Similar considerations apply to the computation of Agg_j in case of COUNT and SUM. The aggregation of Agg_j over the join partners of t in u_1 yields the value $\left(\sum_{\lambda=1}^{\alpha} t_\lambda.Agg_j\right)$. In contrast to MIN and MAX, we now also have to take the possible extensions of t to the relations in the subtrees of T rooted at the nodes $u_2, \ldots, u_k$ into account. The number of possible extensions of t to $\left(\bowtie_{v \in T_{u_i}}(R(v)\right)$ corresponds to the sum s_i of the frequencies of all join partners of t in $R(u_i)$. Hence, by the connectedness conditions, the number of extensions of t to the relations at *all* subtrees of $T_{u_2}, \ldots, T_{u_k}$ is obtained as $\Pi_{i=2}^{k} s_i$.

We conclude this section by an important observation concerning the *size of the relations* that we propagate up the join tree T: for every node u of T, the relation $Freq(u)$ contains precisely the tuples of $R(u)$ that one would get by the first bottom-up traversal of Yannakakis' algorithm via semi-joins, extended by the frequency attribute c_u. That is, we never add tuples, we only add one attribute to each relation. Similarly, for every aggregate expression $A_j(f_j(\bar{B}_j))$ that is not guarded by the root r of T, we add an attribute Agg_j to all nodes along the path between the guard of $A_j(f_j(\bar{B}_j))$ and the root. In other words, the data structures that we have to materialise and propagate in the course of our evaluation of piecewise-guarded queries is *linearly bounded* in the size of the data.

We note that this property would no longer be guaranteed for one of the following two extensions of the piecewise-guarded fragment: either allowing aggregates other than MIN, MAX, SUM, COUNT, AVG to be guarded by a relation different from the root guard or allowing aggregate expressions $A_j(f_j(\bar{B}_j))$ whose attributes are not guarded by a single relation. In both cases, one would have to propagate (all possible) individual values of attributes rather than aggregated values up the join tree, which would destroy this linear bound. A more detailed discussion is given in the full paper [35].

5 OPTIMISED PHYSICAL OPERATORS

The optimisations presented in Section 4 avoid a good deal of materialisation of intermediate results. But there are still joins needed, namely, between the relations at a parent node and its child nodes. Only after these joins, we apply the grouping and aggregation and thus bring the intermediate relations back to linear size (data complexity). Similarly to Schleich et al. [48], we now combine the join computation with aggregation into a single operation. We thus introduce a new physical operator (referred to as AggJoin) that computes and propagates the frequency attribute c from Section 4.1 and the additional aggregate attributes Agg_j from Section 4.2 in a semi-join-like style. Below, we describe a possible *join-less* realisation of this operator. As a preprocessing step (yet before the AggJoin is called), the following operations are carried out:

Every relation is extended by a *frequency attribute* c, and, for every tuple t in every relation R, we initialise $t.c$ as $t.c := 1$. Moreover, for every aggregate expression $A_j(f_j(\bar{B}_j))$ that is not guarded by the root guard, we determine the node w highest up in the join tree that contains all attributes in $f_j(\bar{B}_j)$. Then we add an attribute Agg_j to every relation along the path from w to r. For every tuple $t \in R(w)$, we initialise this attribute as follows:

- If $A_j \in \{\texttt{MIN}, \texttt{MAX}, \texttt{SUM}\}$, then we set $t.Agg_j := f_j(t.\bar{B}_j)$, i.e., we apply f_j to the values of the attributes $\bar{B}_j$ in tuple t.
- If $A_j \in \{\texttt{COUNT}\}$, then we set $t.Agg_j := 1$ if $f_j(t.\bar{B}_j) \neq \texttt{NULL}$, and $t.Agg_j := 0$ if $f_j(t.\bar{B}_j) = \texttt{NULL}$.

For $A_j \in \{\texttt{SUM}, \texttt{COUNT}\}$, we thus deviate from the initialisation of $t.Agg_j$ at node w described in Section 4.2 by leaving out the multiplication of $t.Agg_j$ with the frequency value $t.c$. This multiplication by $t.c$ has to be integrated into the AggJoin operator, which will take care of this multiplication when it determines $t.c$.

Finally, if $A_j \in \{\texttt{SUM}, \texttt{COUNT}\}$, then we set $t.Agg_j := 1$ for every tuple t in a relation labelling an ancestor node u of w. The reason for this is that it will allow us to uniformly propagate the Agg_j value from one child of u and the frequency values from the other children of u in a uniform way via multiplication.

From now on, let R and S denote relations labelling nodes u_R and u_S in the join tree, such that u_S is a child node of u_R. We first describe the propagation of the frequency attribute by the AggJoin for a tuple $r \in R$.

- check that $r \in R \ltimes S$ holds;
- define $S' := S \ltimes \{r\}$, i.e., the tuples in S that join with r;
- define $sc := \sum_{s \in S'} s.c$, i.e., the sum of the frequencies of all tuples in S that join with r;
- Finally, we set $r.c := r.c \cdot sc$, i.e., the frequency of r is multiplied by the sum of the frequencies of the join partners in S. Here, it makes no difference, if $r.c$ still had its initial

value 1 or if R had already gone through calls of AggJoin with relations at other child nodes of u_R.

It is easy to verify that this new AggJoin operator does precisely the work needed to get from $Freq_{i-1}(u_R)$ to $Freq_i(u_R)$ according to Section 4.1. After the initialisation, we have $r.c = 1$ for all tuples in R. This corresponds to $Freq_0(u_R)$. Then we successively execute the AggJoin operator, where $R = Freq_{i-1}(u_R)$ and S is the relation at the i-th child node of u_R. Hence, in each such call, we either delete r (if it has no join partner in S) or we multiply the current value of $r.c$ by the sum of the frequencies of its join partners in S.

Let us now consider the aggregate expressions $A_j(f_j(\bar{B}_j))$ that are not guarded by the root guard. For the *initialisation* of the attribute Agg_j in case of $A_j \in \{\texttt{SUM}, \texttt{COUNT}\}$, we proceed as with the frequency attribute: Suppose that relation R is the one, where Agg_j has to be initialised. Now, for every tuple $r \in R$, the value initially assigned to $r.Agg_j$ ultimately has to be multiplied by $r.c$ to arrive at the initialisation according to Section 4.2. Hence, for every relation S at a child node of u_R, we multiply $r.Agg_j$ with sc = the sum of the frequencies of all tuples in S that join with r.

For the *propagation* of the attribute Agg_j in case of $A_j \in \{\texttt{MIN}, \texttt{MAX}, \texttt{SUM}, \texttt{COUNT}\}$, we distinguish two cases: First suppose that S does not contain the attribute Agg_j. Then, for every tuple $r \in R$ that has at least one join partner in S, we proceed as follows: For $A_j \in \{\texttt{MIN}, \texttt{MAX}\}$, we simply leave the value of $r.Agg_j$ unchanged, i.e., the Agg_j attribute is propagated to R from the relation at a different child node. For $A_j \in \{\texttt{SUM}, \texttt{COUNT}\}$, we again proceed analogously to the frequency propagation, i.e., we multiply $r.Agg_j$ with the sum of the frequencies of all tuples in S that join with r. Now suppose that S contains the attribute Agg_j. Then, for every tuple $r \in R$ that has at least one join partner in S, we proceed as follows: For $A_j \in \{\texttt{MIN}, \texttt{MAX}\}$, we assign to $r.Agg_j$ the minimum resp. maximum value of $s.Agg_j$ over all tuples $s \in S$ that join with r. For $A_j \in \{\texttt{SUM}, \texttt{COUNT}\}$, we determine the sum of the values $s.Agg_j$ over all tuples $s \in S$ that join with r and multiply the current value of $r.Agg_j$ with this sum.

The rewriting from Section 4 allows for a smooth integration of the AggJoin operator into the physical query plan. For instance, we have extended Spark SQL by three different implementations of the AggJoin operator, corresponding to the existing three join implementations *shuffled-hash join*, *sort-merge join*, and *broadcast-hash join*. In Algorithm 1, we sketch the realisation of the AggJoin operator based on the shuffled-hash join. We use pseudo-code notation to leave out the technical details so as not to obscure the simplicity of the extension from join computation to semi-join-like aggregation. As in the explanations above, we write R and S to denote pairs of relations whose nodes in the join tree are in parent-child relationship. Moreover, the AggJoin operator is only called after all the initialisations of additional attributes $t.c$ and $t.Agg_j$ have been carried out as described above. Clearly, the hash-phase (including the partitioning by Spark SQL) is left unchanged. Only the join-phase is affected, which we briefly discuss next:

The AggHashJoin takes as input a set of tuples from R and of tuples from S that join. Additionally, the indices of the aggregate attributes I_S (which have to be propagated from S to R) and I_R (which are only contained in R) are taken as input. In the first step, we initialise sc (that is used to sum up the frequency values

Algorithm 1: Hash Join with aggregate propagation

Input: Two lists R, S of tuples with the same values of the join attributes;
List $I_S = \{s_1, \ldots, s_m\}$ of indices of aggregate attributes Agg_{s_i} present in both S and R;
List $I_R = \{r_1, \ldots, r_n\}$ of indices of aggregate attributes Agg_{r_i} present only in R;

Function *AggHashJoin*(R, S, I_S, I_R)

```
sc ← 0;
foreach s ∈ I_S do
    if A_s ∈ {MIN, MAX} then val_s ← init[s];
    if A_s ∈ {SUM, COUNT} then val_s ← 0;
foreach t ∈ S do
    sc ← sc + t.c;
    foreach s ∈ I_S do
        if A_s ∈ {MIN, MAX} then
            val_s ← A_s(val_s, t.Agg_s);
        if A_s ∈ {SUM, COUNT} then val_s ← val_s + t.Agg_s;
foreach t ∈ R do
    t.c ← t.c · sc;
    foreach s ∈ I_S do
        if A_s ∈ {MIN, MAX} then t.Agg_s ← val_s;
        if A_s ∈ {SUM, COUNT} then
            t.Agg_s ← t.Agg_s · val_s;
    foreach r ∈ I_R do
        if A_r ∈ {SUM, COUNT} then t.Agg_r ← t.Agg_r · sc;
    emit t;
```

over the tuples in S) and val_s for every $s \in I_s$ (that is used for aggregating the attribute Agg_s). An aggregate attribute Agg_s is used to propagate values for the aggregate expression $A_s(f_s(\bar{B}_s))$. For $A_s \in \{\texttt{MIN}, \texttt{MAX}\}$ we assume that the (system-dependent) *maximal* element for this data type in case of MIN and the *minimal* one in case of MAX, respectively, is stored in the variable $init[s]$. The foreach-loop over the tuples of S aggregates the frequency attribute and all the other additional attributes. The foreach-loop over the tuples of R uses these aggregated values from the tuples of S to update the corresponding attributes of the tuples in R. The latter foreach-loop also has to multiply the initial value of aggregate attributes in case of SUM and COUNT by the sum sc of the frequency attributes.

In the full paper [35], we also discuss the realisation of the AggJoin in case of the *sort-merge join*, and *broadcast-hash join*. It is then straightforward to extend these ideas to join types not supported by Spark SQL such as the block-nested loops join.

Clearly, replacing a physical join operation by the respective AggJoin variant does not introduce any overhead (apart from the computationally cheap management of the additional frequency and aggregate attributes). Moreover, if none of the additional attributes is needed (e.g., if the query is 0MA), then our AggJoin operator actually degenerates to a simple semi-join.

6 EXPERIMENTAL EVALUATION

Experimental Setup. In order to evaluate our approach for materialisation-free query evaluation, we have implemented the methods presented in Sections 4 and 5 in Spark SQL. We perform experiments using a wide range of standard benchmarks, namely the *Join Order Benchmark (JOB)* [36], *STATS-CEB* [23], TPC-H [53], TPC-DS [52], and the *Large-Scale Subgraph Query Benchmark (LSQB)* [39]. In addition, we evaluate performance on simple graph queries evaluated on two real-world graphs from the *SNAP (Stanford Network Analysis Project* [37]) dataset. In particular, we experiment on the following two datasets which are commonly used in analyses of graph queries (e.g., by Hu and Wang [24]):

Graph	Nodes	Edges	(un)directed
web-Google	875,713	5,105,039	directed
com-DBLP	317,080	1,049,866	undirected

We evaluate the performance of basic graph queries, namely path queries requiring between 3 and 8 joins (i.e., between 4 and 9 edges) and three tree queries of depth between 3-4 joins and 4-7 joins in total. These queries can be viewed as counting the number of homomorphisms from certain patterns (i.e., paths and trees in this case); a task that has recently gained popularity in graph learning where the results of the queries are injected into machine learning models (e.g., [9, 10, 29, 42]).

The overall performance of our proposed optimisations on the applicable queries is summarised in Table 3. We refer to the reference performance of Spark SQL without any alterations as *Ref*. Our experiments on the SNAP graphs specifically are summarised in Table 2. The fastest execution time achieved for each case is printed in boldface. The results obtained by applications of the logical optimisations described in Section 4.1 for guarded queries are referred to as *Guarded Aggregate Optimisation (GuAO)*. We use *GuAO*$^+$ to refer to the further extension of our logical optimisations to piecewise-guarded queries described in Section 4.2 plus the enhancement of the physical query plan using the AggJoin operator described in Section 5. The speed-up achieved by *GuAO*$^+$ over *Ref* is explicitly stated in Table 3 in the column *GuAO*$^+$ *Speedup*. For most benchmarks, we report end-to-end (e2e) times for subsequently executing all queries of a given benchmark where our optimisations are applicable (to the full query, or at least one subquery).

In all experiments, we execute each query 6 times, with the first run being a warm-up run to ensure that our measurements are not affected by initial reads of tables into memory. We report statistics gathered from the last 5 runs and report mean query execution time as well as the standard deviation over these runs. Finally, note that we execute the full query, even if our optimisation applies only to a subquery. In such a case, the plan for the subquery is optimised according to Section 4, and the rest of the query plan remains unchanged.

Full experimental details are provided in the full paper [35].

Applicability.

To enable Yannakakis-style query evaluation in the context of standard query execution engines, we have focused on specific queries, namely guarded and piecewise-guarded acyclic aggregate queries (cf., Section 4). As a first step, we therefore provide a more detailed analysis as to how many of the queries in the studied benchmarks fit into this class, and what factors limit further applicability. The analysis is summarised in Table 1.

Despite the variety of considered benchmarks, we find that our optimisations for piecewise-guarded queries are widely applicable through all of them. In JOB and STATS-CEB, all queries fall into the schema of piecewise guarded-aggregation, and our method thus applies to these benchmarks as a whole. In contrast, the 0MA fragment covers only 19 of the 259 queries in total in these two benchmarks. Our methods also apply to all the tested basic graph queries (path or tree queries) that were tested on the SNAP dataset.

In LSQB, our approach applies to 2 of the 9 queries. But as reported in Table 1, only 4 of the queries are equi-join queries with aggregation, the others contain joins on inequalities, which requires entirely different techniques (e.g., [33]). Of the 4 equi-join queries, 2 are not acyclic.

Our method applies to half of the equi-join aggregate queries in the TPC-DS benchmark. The queries in the benchmark are typically highly complex, often combining multiple subqueries and employing more elaborate SQL features. We observe that in some instances, *GuAO*$^+$ even applies to multiple subqueries in the same query. In TPC-H, our optimisations apply to 7 out of the 15 acyclic equi-join aggregate queries in the benchmark. Notably, TPC-H Q2 contains a 0MA subquery (with `MIN` aggregation) and TPC-H Q11 contains a guarded sum aggregate subquery.

TPC-H Q2 is particularly illustrative as the subquery is correlated: the attribute `p_partkey` from the outer query is used in the aggregation subquery as follows:

```
SELECT MIN(ps_supplycost)
... WHERE p_partkey = ps_partkey ...
```

The Spark SQL query planner decorrelates this subquery via typical magic decorrelation (see [49]) – resulting in the following select statement for the decorrelated subquery. This query is still guarded and thus 0MA. Our rewriting rules then apply naturally after decorrelation, with no need for any special handling of these cases.

```
SELECT ps_partkey, MIN(ps_supplycost)
... GROUP BY ps_partkey
```

We recall that our method is fully integrated into the query optimisation phase. Hence, when our optimisations are not applicable to a query, its execution is not affected. Recognising whether the rewriting rules are applicable is trivial and requires, in our observations, negligible additional time in the query planning phase to perform our rewriting (about 2ms in all of our experiments). Going forward, it is additionally possible to pre-process queries to make them fit into the fragments where our methods are applicable. A brief discussion of such approaches is given in the outlook to future work in Section 7.

Performance impact of GuAO/GuAO$^+$*.*

Our main results over standard benchmarks are summarised in Table 3. We additionally report the mean number of joins of the (sub)queries affected by our optimisation. We make two key observations with respect to the performance of our methods.

When queries are challenging – e.g., they have many joins or the joins are not along PK/FK pairs – then our method provides

Table 1: Summary of the applicability of our method on benchmarks. We report the number of queries in benchmark (#), equi-join aggregate queries (⋈-*agg*), acyclic queries (*acyc*), piecewise-guarded queries (*pwg*), guarded queries (*g*), and 0MA queries. Fragments proposed in this work are highlighted in blue.

Benchmark	#	⋈-agg	acyc	pwg	g	0MA
JOB	113	113	113	113	19	19
STATS-CEB	146	146	146	146	146	0
TPC-H	22	15	14	7	3	1
LSQB	9	4	2	2	2	0
SNAP	18	18	18	18	18	0
TPC-DS	99	64	63	30	15	0

Table 2: Performance on SNAP graphs (t.o. marks timeouts, o.o.m marks out of memory errors).

	web-Google			com-DBLP		
Query	Spark	GuAO	GuAO$^+$	Spark	GuAO	GuAO$^+$
path-03	27.97±1.5	6.90±0.6	6.08±0.65	6.32±1.1	2.35±0.5	1.59±0.12
path-04	449.14±26.9	7.58±0.6	**6.89**±0.30	50.97±9.8	2.24±0.4	**1.76**±0.16
path-05	o.o.m.	8.95±1.0	**7.53**±0.48	400.87±15.2	2.74±0.2	**2.03**±0.25
path-06	o.o.m.	9.37±1.0	**8.80**±0.25	o.o.m.	2.98±0.2	**2.18**±0.14
path-07	o.o.m.	11.32±0.9	**9.76**±1.21	o.o.m.	3.64±0.2	**2.38**±0.26
path-08	o.o.m.	11.30±2.1	**10.05**±1.49	o.o.m.	3.75±0.4	**2.53**±0.30
tree-01	539.11±22.4	7.73±1.0	**6.53**±1.11	25.96±4.5	1.95±0.1	**1.47**±0.28
tree-02	o.o.m.	12.43±3.2	**7.29**±0.73	328.88±11.5	3.02±0.7	**1.69**±0.16
tree-03	o.o.m.	12.21±5.6	**8.16**±0.66	o.o.m.	3.17±0.2	**1.99**±0.16

Table 3: Summary of the impact of aggregate optimisation on execution times (seconds). Reported numbers are mean times over 5 runs of the same query with standard deviations given after ±. "–" indicates that the query is piecewise-guarded and, therefore, the optimisation from Section 4.1 for guarded queries is not applicable.

Query	# joins (mean)	Ref	GuAO	GuAO$^+$	GuAO$^+$ Speedup
STATS-CEB e2e	3.33	1558±7.3	97.9±6.1	**64.8**±7.9	24.04 x
JOB e2e	7.65	3217.84±106	-	**2189.46**±76	1.47 x
TPC-H e2e SF200	1.57	3757.2	-	**3491.06**	1.08 x
TPC-H Ex.1 SF200	4	168.4	107.5	**105.11**	1.60 x
LSQB Q1 SF300	9	3096±232	**677**±23	688±23	4.57 x
LSQB Q4 SF300	3	602±37	593±15	**592**±9	1.02x
TPC-DS e2e SF100	2.52	5154.5	-	5047.5	1.02 x

enormous potential for speed-up. In JOB, a benchmark where sub-optimal join orderings in large queries cause intermediate blow-up, we achieve almost 50% speed-up. STATS-CEB purposefully introduces joins along FK/FK relationships to challenge query evaluation systems with the resulting large number of intermediate tuples. Our method automatically avoids all of these difficulties and we see an immense 24-fold speed-up. Similarly, for the more difficult of the two LSQB queries (Q1), we see that the large number of joins creates significant intermediate blow-up with standard relational query evaluation. Again our method achieves a very large improvement of about 450%. Even in the simple query from Example 1.1 we observe 60% speed-up over unoptimised Spark SQL.

On the other hand, we observe that especially the two TPC benchmarks contain primarily queries where the join evaluation itself is very straightforward. In TPC-H, 4 of the 7 tested queries contain only a single join. In TPC-DS we observe similar patterns. As a result, there is little to no unnecessary materialisation in many of these queries. Our key insight here is that the experiments confirm that our method (and in particular *GuAO*$^+$) does not introduce any overhead in these cases. By not causing performance decrease in those cases where there is no unnecessary materialisation, combined with some gains in the few harder queries, we still see modest overall speed-ups for these benchmarks.

With respect to basic graph queries, we see in Table 2 that even with significant resources, counting short paths and small trees is effectively impossible on large graphs with current methods. This holds for both, Spark SQL and for specialised graph database systems. In stark contrast, *GuAO* and *GuAO*$^+$ effectively trivialise these types of queries even with significantly less resources than are available on our test system (the highest observed memory usage for *GuAO*$^+$ in our SNAP experiments was roughly 5GB). Since these experiments focus on graph data, we additionally compared with specialised graph database systems (Neo4j [41], KÙZU system [30], and GraphDB [20]). The results of these experiments are equally sobering as with standard Spark SQL, in that almost all queries failed with timeout (set to 30 min). This is in sharp contrast to GuAO and GuAO$^+$, which answer these queries in a matter of a few seconds. We refer to the full paper [35] for detailed results.

In summary, our experiments paint a clear picture. In more challenging queries, our approach offers very significant improvements. At the same time, in cases where little unnecessary materialisation is performed, *GuAO*$^+$ introduces no additional overhead and thus exhibits no performance degradation on simpler queries.

How much materialisation can be avoided?

Throughout the paper, we have been motivated by the premise that easy to implement logical optimisation rules for query plans can avoid a significant amount of intermediate materialisation in aggregate queries. Moreover, with the addition of natural physical operators, we can avoid any such materialisation altogether. However, this raises the question of how much unnecessary materialisation actually occurs when using standard query planning methods.

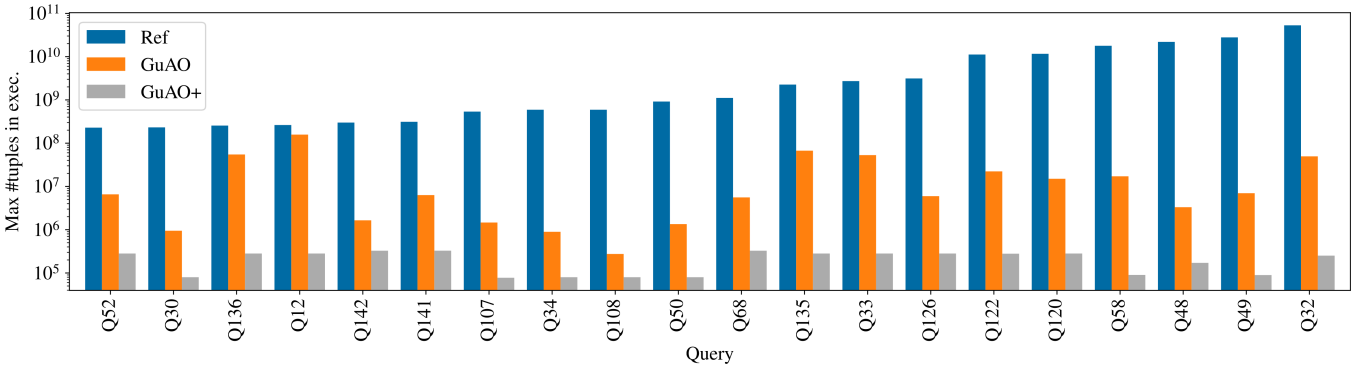


Figure 4: Comparison of the maximal number of materialised tuples in a table during query execution for 20 queries of STATS-CEB. Y-axis in logarithmic scale (base 10).

To study this question, we compare the maximum number of tuples that occur in an intermediate table during query execution for the STATS-CEB queries. Again, we report the mean over 5 runs (we omit error bars as the variation between runs is mostly 0 and negligible in other cases). We note that these intermediate table sizes are naturally closely correlated to overall memory consumption, as well as communication cost in a distributed setting. Figure 4 reports the peak number of materialised tuples during query execution for the 20 queries where standard Spark SQL materialises the most intermediate tuples. The data clearly shows that an improvement in the order of magnitudes of materialised tuples is often possible. In particular, we see the well documented effect of classical relational query processing techniques leading to substantial intermediate blow-up. The largest relations in the dataset have in the order of $3 \cdot 10^5$ tuples, an enormous difference to the observed sizes of up to 10^{10} intermediate tuples for *Ref*. The data shows that already by rewriting the logical query plans according to Section 4, we regularly see a reduction in peak intermediate table size of over 2 orders of magnitude.

However, the optimised logical query plan still requires some mild materialisation between aggregation steps, which we manage to eliminate with the physical operators described in Section 5. The resulting *GuAO*$^+$ system consistently reduces the number of materialised tuples by **at least 3 orders of magnitude** on the reported queries in Figure 4. In fact, the reported numbers for *GuAO*$^+$ are always precisely the cardinality of the largest relation in the query, as the execution using our method never introduces any new tuples (cf. Section 5). That is, this number can also inherently not be improved upon. Over the whole benchmark, we observe that the peak number of materialised tuples by *GuAO*$^+$ is *at least 10 times less* than that of standard Spark SQL query execution in 118 out of the 146 queries. In all other cases, the peak number of materialised tuples by *Ref* and *GuAO*$^+$ is exactly the same, i.e., *Ref* is never better.

7 CONCLUSION

In this work, we have introduced several optimisations for guarded aggregate queries, enabling significant reductions of the need to materialise intermediate results when evaluating analytical queries with aggregates over join or path queries. Our approach emphasises seamless integration into standard database systems, requiring only localised modifications to logical query plans. Additionally, we propose the use of new physical operators that extend semi-joins to manage frequencies and other aggregate information with the aim to completely eliminate the computation of intermediate joins and to facilitate straightforward integration into physical query plans.

We have implemented our optimisations into Spark SQL, which has been specifically designed as a powerful tool to deal with complex analytical queries. Our experimental evaluation confirms that the proposed techniques can provide significant performance improvements by avoiding costly materialisation, especially in larger queries. Furthermore, the integration of our optimisations into Spark SQL serves not only as a proof of concept but is already a practical enhancement of an important tool in data analytics – which is even more valuable given the limited applicability of advanced query optimisation techniques in Spark SQL due to its limited cost-model.

So far, we have applied our optimisation techniques to acyclic queries with piecewise-guarded aggregation. In principle, both acyclicity and piecewise-guardedness can be enforced by applying appropriate joins upfront. More specifically, if a query uses unguarded aggregation, we can create a guard by joining relations that cover all attributes involved in grouping and/or aggregation. Similarly, a cyclic query can be transformed into an acyclic one via (generalised or fractional) hypertree decompositions [3, 18, 22]. Again, this requires the computation of some joins upfront. Of course, carrying out such joins upfront to ensure guardedness and/or acyclicity comes at a cost – both in terms of space and time. We leave it as an interesting open question for future research to analyse when extending our approach to cyclic and/or unguarded queries actually pays off.

ACKNOWLEDGMENTS

This work has been supported by the Vienna Science and Technology Fund (WWTF) [10.47379/ICT2201, 10.47379/VRG18013]. We acknowledge the assistance of the TU Wien TU.it dataLAB team. We sincerely thank Katja Hose for many helpful tips on a previous version of this work, and we would also like to express our gratitude to the anonymous reviewers for their valuable comments and suggestions, which have greatly improved this work.

REFERENCES

[1] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4 (2017), 20:1–20:44. https://doi.org/10.1145/3129246

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley. http://webdam.inria.fr/Alice/

[3] Isolde Adler, Georg Gottlob, and Martin Grohe. 2007. Hypertree width and related hypergraph invariants. *Eur. J. Comb.* 28, 8 (2007), 2167–2181. https://doi.org/10.1016/j.ejc.2007.04.013

[4] Foto N. Afrati, Manas R. Joglekar, Christopher Ré, Semih Salihoglu, and Jeffrey D. Ullman. 2017. GYM: A Multiround Distributed Join Algorithm. In *Proceedings ICDT (LIPIcs)*, Vol. 68. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:18. https://doi.org/10.4230/LIPIcs.ICDT.2017.4

[5] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings SIGMOD*. ACM, 1383–1394. https://doi.org/10.1145/2723372.2742797

[6] Albert Atserias, Martin Grohe, and Dániel Marx. 2013. Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput.* 42, 4 (2013), 1737–1767. https://doi.org/10.1137/110859440

[7] Nurzhan Bakibayev, Tomás Kociský, Dan Olteanu, and Jakub Zavodny. 2013. Aggregation and Ordering in Factorised Databases. *Proc. VLDB Endow.* 6, 14 (2013), 1990–2001. https://doi.org/10.14778/2556549.2556579

[8] Lorenzo Baldacci and Matteo Golfarelli. 2019. A Cost Model for SPARK SQL. *IEEE Trans. Knowl. Data Eng.* 31, 5 (2019), 819–832. https://doi.org/10.1109/TKDE.2018.2850339

[9] Linus Bao, Emily Jin, Michael Bronstein, İsmail İlkan Ceylan, and Matthias Lanzinger. 2024. Homomorphism Counts as Structural Encodings for Graph Learning. arXiv:2410.18676 [cs.LG] https://arxiv.org/abs/2410.18676

[10] Pablo Barceló, Floris Geerts, Juan L. Reutter, and Maksimilian Ryschkov. 2021. Graph Neural Networks with Local Graph Parameters. In *Proceedings NeurIPS*. 25280–25293. https://proceedings.neurips.cc/paper/2021/hash/d4d8d1ac7e00e9105775a6b660dd3cbb-Abstract.html

[11] Hubie Chen and Víctor Dalmau. 2012. Decomposing Quantified Conjunctive (or Disjunctive) Formulas. In *Proceedings LICS*. IEEE Computer Society, 205–214. https://doi.org/10.1109/LICS.2012.31

[12] Binyang Dai, Qichen Wang, and Ke Yi. 2023. SparkSQL+: Next-generation Query Planning over Spark. In *Companion of the 2023 International Conference on Management of Data, SIGMOD/PODS 2023*. ACM, 115–118. https://doi.org/10.1145/3555041.3589715

[13] Arnaud Durand and Stefan Mengel. 2015. Structural Tractability of Counting of Solutions to Conjunctive Queries. *Theory Comput. Syst.* 57, 4 (2015), 1202–1249. https://doi.org/10.1007/S00224-014-9543-Y

[14] Georg Gottlob, Matthias Lanzinger, Davide Mario Longo, Cem Okulmus, Reinhard Pichler, and Alexander Selzer. 2023. Reaching Back to Move Forward: Using Old Ideas to Achieve a New Level of Query Optimization (short paper). In *Proceedings AMW (CEUR Workshop Proceedings)*, Vol. 3409. CEUR-WS.org. https://ceur-ws.org/Vol-3409/paper6.pdf

[15] Georg Gottlob, Matthias Lanzinger, Davide Mario Longo, Cem Okulmus, Reinhard Pichler, and Alexander Selzer. 2023. Structure-Guided Query Evaluation: Towards Bridging the Gap from Theory to Practice. *CoRR* abs/2303.02723 (2023). https://doi.org/10.48550/arXiv.2303.02723 arXiv:2303.02723

[16] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 2001. *J. ACM* 48, 3 (2001), 431–498. https://doi.org/10.1145/382780.382783

[17] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 2002. Hypertree Decompositions and Tractable Queries. *J. Comput. Syst. Sci.* 64, 3 (2002), 579–627. https://doi.org/10.1006/jcss.2001.1809

[18] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 2002. Hypertree Decompositions and Tractable Queries. *J. Comput. Syst. Sci.* 64, 3 (2002), 579–627. https://doi.org/10.1006/JCSS.2001.1809

[19] Marc H. Graham. 1979. *On The Universal Relation*. Technical Report. University of Toronto.

[20] GraphDB. (accessed July 2024). https://graphdb.ontotext.com/.

[21] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings PODS*. ACM, 31–40. https://doi.org/10.1145/1265530.1265535

[22] Martin Grohe and Dániel Marx. 2014. Constraint Solving via Fractional Edge Covers. *ACM Trans. Algorithms* 11, 1 (2014), 4:1–4:20.

[23] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *Proc. VLDB Endow.* 15, 4 (2021), 752–765. https://doi.org/10.14778/3503585.3503586

[24] Xiao Hu and Qichen Wang. 2023. Computing the Difference of Conjunctive Queries Efficiently. *Proc. ACM Manag. Data* 1, 2 (2023), 153:1–153:26. https://doi.org/10.1145/3589298

[25] Zeyuan Hu and Daniel P. Miranker. 2024. TreeTracker Join: Turning the Tide When a Tuple Fails to Join. *CoRR* abs/2403.01631 (2024). arXiv:2403.01631 http://arxiv.org/abs/2403.01631

[26] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. 2017. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *Proceedings SIGMOD*. ACM, 1259–1274. https://doi.org/10.1145/3035918.3064027

[27] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2020. General dynamic Yannakakis: conjunctive queries with theta joins under updates. *VLDB J.* 29, 2-3 (2020), 619–653. https://doi.org/10.1007/s00778-019-00590-9

[28] Xuechun Ji, Mao-Xian Zhao, Mingyu Zhai, and Qingxi Wu. 2020. Query Execution Optimization in Spark SQL. *Sci. Program.* 2020 (2020), 6364752:1–6364752:12. https://doi.org/10.1155/2020/6364752

[29] Emily Jin, Michael Bronstein, İsmail İlkan Ceylan, and Matthias Lanzinger. 2024. Homomorphism Counts for Graph Neural Networks: All About That Basis. *CoRR* abs/2402.08595 (2024). https://doi.org/10.48550/ARXIV.2402.08595 arXiv:2402.08595

[30] Guodong Jin, Xiyang Feng, Ziyi Chen, Chang Liu, and Semih Salihoglu. 2023. KÙZU Graph Database Management System. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p48-jin.pdf

[31] Manas R. Joglekar, Rohan Puttagunta, and Christopher Ré. 2016. AJAR: Aggregations and Joins over Annotated Relations. In *Proceedings PODS*. ACM, 91–106. https://doi.org/10.1145/2902251.2902293

[32] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions Asked Frequently. In *Proceedings PODS*. ACM, 13–28. https://doi.org/10.1145/2902251.2902280

[33] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. 2015. Lightning Fast and Space Efficient Inequality Joins. *Proc. VLDB Endow.* 8, 13 (2015), 2074–2085. https://doi.org/10.14778/2831360.2831362

[34] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. 2014. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.* 23, 2 (2014), 253–278. https://doi.org/10.1007/S00778-013-0348-4

[35] Matthias Lanzinger, Reinhard Pichler, and Alexander Selzer. 2024. Avoiding Materialisation for Guarded Aggregate Queries. *CoRR* abs/2406.17076 (2024). https://doi.org/10.48550/ARXIV.2406.17076 arXiv:2406.17076

[36] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[37] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[38] Riccardo Mancini, Srinivas Karthik, Bikash Chandra, Vasilis Mageirakos, and Anastasia Ailamaki. 2022. Efficient Massively Parallel Join Optimization for Large Queries. In *Proceedings SIGMOD*. ACM, 122–135. https://doi.org/10.1145/3514221.3517871

[39] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. 2021. LSQB: a large-scale subgraph query benchmark. In *Proceedings GRADES*. ACM, 8:1–8:11. https://doi.org/10.1145/3461837.3464516

[40] Michail Georgoulakis Misegiannis, Vasiliki Kantere, and Laurent d'Orazio. 2022. Multi-objective query optimization in Spark SQL. In *Proceedings IDEAS*. ACM, 70–74. https://doi.org/10.1145/3548785.3548800

[41] Neo4J. (accessed July 2024). https://neo4j.com/.

[42] Hoang Nguyen and Takanori Maehara. 2020. Graph Homomorphism Convolution. In *Proceedings ICML (Proceedings of Machine Learning Research)*, Vol. 119. PMLR, 7306–7316. http://proceedings.mlr.press/v119/nguyen20c.html

[43] Milos Nikolic and Dan Olteanu. 2018. Incremental View Maintenance with Triple Lock Factorization Benefits. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 365–380. https://doi.org/10.1145/3183713.3183758

[44] Dan Olteanu and Maximilian Schleich. 2016. Factorized Databases. *SIGMOD Rec.* 45, 2 (2016), 5–16. https://doi.org/10.1145/3003665.3003667

[45] Dan Olteanu and Jakub Závodný. 2015. Size Bounds for Factorised Representations of Query Results. *ACM Trans. Database Syst.* 40, 1 (2015), 2:1–2:44. https://doi.org/10.1145/2656335

[46] Adam Perelman and Christopher Ré. 2015. DunceCap: Compiling Worst-Case Optimal Query Plans. In *Proceedings SIGMOD*. ACM, 2075–2076. https://doi.org/10.1145/2723372.2764945

[47] Reinhard Pichler and Sebastian Skritek. 2013. Tractable counting of the answers to conjunctive queries. *J. Comput. Syst. Sci.* 79, 6 (2013), 984–1001. https://doi.org/10.1016/j.jcss.2013.01.012

[48] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. 2019. A Layered Aggregate Engine for Analytics Workloads.

In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1642–1659. https://doi.org/10.1145/3299869.3324961

[49] Praveen Seshadri, Hamid Pirahesh, and T. Y. Cliff Leung. 1996. Complex Query Decorrelation. In *Proceedings ICDE*. IEEE Computer Society, 450–458. https://doi.org/10.1109/ICDE.1996.492194

[50] Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. 2022. Functional collection programming with semi-ring dictionaries. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–33. https://doi.org/10.1145/3527333

[51] Yu Shen, Xinyuyang Ren, Yupeng Lu, Huaijun Jiang, Huanyong Xu, Di Peng, Yang Li, Wentao Zhang, and Bin Cui. 2023. Rover: An Online Spark SQL Tuning Service via Generalized Transfer Learning. In *Proceedings KDD*. ACM, 4800–4812. https://doi.org/10.1145/3580305.3599953

[52] TPC-DS. [n.d.]. TPC-DS Benchmark. https://www.tpc.org/tpcds/.

[53] TPC-H. [n.d.]. TPC-H Benchmark. https://www.tpc.org/tpch/.

[54] Susan Tu and Christopher Ré. 2015. DunceCap: Query Plans Using Generalized Hypertree Decompositions. In *Proceedings SIGMOD*. ACM, 2077–2078. https://doi.org/10.1145/2723372.2764946

[55] Qichen Wang, Xiao Hu, Binyang Dai, and Ke Yi. 2023. Change Propagation Without Joins. *CoRR* abs/2301.04003 (2023). https://doi.org/10.48550/arXiv.2301.04003 arXiv:2301.04003

[56] Qichen Wang and Ke Yi. 2022. Conjunctive Queries with Comparisons. In *SIGMOD '22: International Conference on Management of Data*. ACM, 108–121. https://doi.org/10.1145/3514221.3517830

[57] Yilei Wang and Ke Yi. 2021. Secure Yannakakis: Join-Aggregate Queries over Private Data. In *SIGMOD '21: International Conference on Management of Data*. ACM, 1969–1981. https://doi.org/10.1145/3448016.3452808

[58] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Proceedings VLDB*. VLDB, 82–94.

[59] C. T. Yu and M. Z. Özsoyoğlu. 1979. An algorithm for tree-query membership of a distributed query. In *The IEEE Computer Society's Third International Computer Software and Applications Conference, COMPSAC 1979*. 306–312.

[60] Mingyu Zhai, Aibo Song, Jingyi Qiu, Xuechun Ji, and Qingxi Wu. 2019. Query optimization Approach with Shuffle Intermediate Cache Layer for Spark SQL. In *Proceedings IPCCC*. IEEE, 1–6. https://doi.org/10.1109/IPCCC47392.2019.8958719

[61] Hao Zhang, Jeffrey Xu Yu, Yikai Zhang, Kangfei Zhao, and Hong Cheng. 2020. Distributed Subgraph Counting: A General Approach. *Proc. VLDB Endow.* 13, 11 (2020), 2493–2507. http://www.vldb.org/pvldb/vol13/p2493-zhang.pdf