Database Theory in Action: Evaluation of Aggregate Queries Without Materialisation

Matthias Lanzinger
TU Wien, Austria

Reinhard Pichler
TU Wien, Austria

Alexander Selzer
TU Wien, Austria

Abstract

Aggregate queries often require computing large intermediate joins despite producing only small outputs. We identify broad classes of acyclic aggregate queries that can be evaluated without materialising any join results, using a bottom-up, semi-join–based propagation of cardinalities and partial aggregates. An implementation in Spark SQL shows that this approach is widely applicable and yields substantial performance gains on standard benchmarks.

2012 ACM Subject Classification Information systems → Database query processing

Keywords and phrases Join Processing, Aggregate Queries, Acyclic Conjunctive Queries

Digital Object Identifier 10.4230/LIPIcs.ICDT.2026.24

Related Version *Full Version*: `https://www.vldb.org/pvldb/vol18/p1398-selzer.pdf` [6]

Supplementary Material *Software (Benchmarks)*: `https://github.com/dbai-tuw/spark-eval` archived at `swh:1:dir:2f5d396ae0c79a54a25a955db978c21a412c840e`

Funding This work has been supported by the Vienna Science and Technology Fund (WWTF) [10.47379/ICT2201, 10.47379/VRG18013].

1 Introduction

Conjunctive queries (CQs) are arguably the most fundamental type of queries on relational databases. However, as the number of relations to be joined grows, their evaluation gets more and more challenging. A major source of the computational complexity of CQs is the potential explosion of intermediate results. In case of acyclic conjunctive queries (ACQs), Yannakakis' algorithm [13] reduces this problem to some extent by eliminating dangling tuples via semi-joins before the actual join computation. However, for large ACQs, the (intermediate) join results may still become prohibitively big. This is particularly frustrating in case of aggregate queries where one first evaluates big joins but ultimately only outputs a small aggregated result. Advanced optimisation techniques like AJAR [4] and FAQs [5] do propose general techniques for aggregate queries, but are unfortunately incompatible with the architectures of the query evaluation engines of typical relational DBMS.

There are special cases of ACQs where the problem of exploding (intermediate) join results simply does not exist: above all, this applies to Boolean ACQs, which can be solved by a bottom-up traversal of the join tree via semi-joins. In [1, 2], a very restricted class of aggregate queries (so-called Zero-Materialisation Aggregate queries, 0MA for short) was presented, for which the result can be read off at the root node of the join tree after the bottom-up traversal via semi-joins. The goal of our work is to identify a big class of aggregate queries that can also be evaluated by a bottom-up traversal via semi-join-like operations without materializing any join results. Formally, we study queries of the following form:

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

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 $R_1, \ldots, R_n$.

The crucial concept to arrive at a class of aggregate queries that can be evaluated without the need to compute any joins is *guardedness*. More precisely, we call a query of the form in (1) a *guarded aggregate 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 may arbitrarily choose one as the guard. The key idea for an efficient (join-less) evaluation of guarded queries is to propagate *cardinalities* of join results rather than the join results themselves in a bottom-up traversal of the join tree. It is then possible to evaluate *any* SQL aggregate operator at the root node of the join tree by taking these cardinalities into account.

In fact, we can further relax the requirement of the existence of a single guard for all of the grouping and aggregation attributes. To this end we introduce the class of *piecewise guarded aggregate queries*. Here, we only require the existence of a guard for the grouping attributes and guards for the aggregate expressions; but, in case of the most common aggregate operators MIN, MAX, COUNT, SUM, and AVG, every aggregate expression may have a different guard. As we outline in Section 2, such queries can be evaluated by attaching aggregated information to tuples during the bottom-up traversal of the join tree, without computing joins. The number of tuples thus propagated is the same as in case of the semi-join bottom-up traversal for Boolean ACQs. This class of queries covers a large proportion of common benchmarks (e.g., 100% of JOB [7] and STATS-CEB [3]). We have implemented our optimisation in Spark SQL and our experiments (Section 3) on several standard benchmarks clearly demonstrate the efficacy of this approach.

2 Logical and Physical Optimisations

We evaluate *guarded aggregate queries* without propagating any join results up the join tree, by adapting the extension of Yannakakis' algorithm for `COUNT(*)` from [10]. To describe the main ideas, we introduce the following notation: let T be a join tree of the query such that the root node is labelled by the guard relation; let u denote a node in T and let T_u denote the set of all nodes in the subtree rooted at u. Moreover, let $R(u)$ denote the relation labelling node u and let $Att(u)$ denote the list of attributes of $R(u)$. The goal of the bottom-up propagation of cardinalities is to evaluate, for every node u in T, the following query:

$$\gamma[Att(u), \texttt{COUNT(*)}]\left(\bowtie_{v \in T_u} R(v)\right) \tag{2}$$

Conceptually, this is the result of joining all relations in the subtree rooted at u, grouped by the attributes occurring in the relation at u, and COUNT(*) for each group. Of course, we do not actually compute these joins. Instead, in the style of semi-joins, we compute the COUNT(*) values by a semi-join-like bottom-up traversal of the join tree. However, rather than just checking for each tuple $t \in R(u)$ the *existence of a join partner* in the subtree below, we compute the *number c_t of join partners of t*. This is done by initializing the cardinality attribute c_t to 1 for every t. For internal nodes u of T, we visit each child v of u (one at a time), sum up the cardinalities of all join partners of t in $R(v)$ and multiply c_t with this sum.

When the bottom-up propagation of cardinalities has reached the root node r of the join tree T, we can evaluate the group-by and aggregation based on the resulting relation (extended by the cardinality attribute) at r. By definition, the guard contains all grouping attributes. Hence, we can evaluate the group-by solely based on $R(r)$. For the evaluation of the aggregates, we take the additional information of the cardinality attribute into account. For instance, COUNT(*) is obtained as the sum over the c_t values for all tuples in the considered group. SUM(B) has to sum up the values $B \cdot c_t$ for all tuples t, taking NULLs into account. For details of the cardinality propagation and the evaluation of the remaining aggregate operators, see the full paper [6].

In case of *piecewise guarded aggregate queries*, we may have to deal with different guards for the grouping attributes and for the various aggregate expressions. We choose as root of the join tree a guard of the grouping attributes. As before, each relation is extended by a cardinality attribute, which is propagated bottom-up in the join tree T as described before. Aggregate expressions whose guard is the relation $R(r)$ at the root node r are processed as before. For an aggregate expression $A_j(a_j)$ which is not guarded by r, we choose as guard the node w that contains all attributes in the expression a_j and which is furthest up in T with this property. We extend the relations at all nodes u on the path between w and the root r by an attribute Agg_j that contains the required aggregated information to ultimately evaluate $A_j(a_j)$. Conceptually, for every such node u, we want to evaluate the query

$$\gamma[Att(u), Agg_j \leftarrow A_j(a_j)]\big(\bowtie_{v \in T_u}(R(v))\big), \tag{3}$$

That is, as with the cardinality attribute, we want to compute the join of all relations in the subtree rooted at u, group by the attributes occurring in the relation at u and evaluate $A_j(a_j)$ for each group. Again, we determine the concrete values of $A_j(a_j)$ by a semi-join-like operation in a bottom-up traversal of T rather than actually computing the joins. The bottom-up traversal for the aggregate expression $A_j(a_j)$ starts at the node w labelled by the guard of $A_j(a_j)$. The initialisation of Agg_j for each tuple $t \in R(w)$ depends on the concrete aggregate operator. For MIN and MAX, we simply evaluate the expression a_j for each tuple t and take the result as the initial value. Agg_j is initialised according to the operator (value of a_j for MIN/MAX, 0 or c_t for COUNT, $a_j \cdot c_t$ for SUM). The bottom-up propagation of the Agg_j attribute in a semi-join-like style (i.e., without computing any joins) is done similarly to the bottom-up propagation of the cardinalities. For details of the Agg_j-propagation and its efficient realisation by a dedicated physical operator, we refer the reader to the full paper [6].

3 Experimental Evaluation

To evaluate our approach for materialisation-free query evaluation, we implemented the methods presented here (for full details, see [6]) in Spark SQL. Notably our implementation naturally integrates into the existing logical and physical query planning steps and operates entirely transparent to the user. Our experiments cover a broad spectrum of standard benchmarks: the *Join Order Benchmark (JOB)* [7], *STATS-CEB* [3], TPC-H [12], TPC-DS [11], and the *Large-Scale Subgraph Query Benchmark (LSQB)* [9]. In addition, we evaluatethe performance on simple graph queries over two real-world graphs from the *SNAP (Stanford Network Analysis Project* [8]) dataset.

Table 3 summarises the overall performance of our optimisations on the applicable queries. We use *Ref* to denote unmodified Spark SQL, and *GuAO*$^+$ to denote our implementation with the new optimisations. Results for the SNAP graphs are reported separately in Table 2. In both tables, the fastest execution time is highlighted in bold. Each query is executed 6 times,

Table 1 Applicability of our method on standard benchmarks. We report the number of queries in benchmark (#), equi-join aggregate queries ($\bowtie$-*agg*), acyclic queries (*acyc*), piecewise-guarded queries (*pwg*), guarded queries (*g*).

Bench.	#	$\bowtie$-agg	acyc	pwg	g
JOB	113	113	113	113	19
STATS	146	146	146	146	146
TPC-H	22	15	14	7	3
LSQB	9	4	2	2	2
SNAP	18	18	18	18	18
TPC-DS	99	64	63	30	15

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$^+$	Spark	GuAO$^+$
path-03	27.97±1.5	6.08±0.65	6.32±1.1	1.59±0.12
path-04	449.14±26.9	**6.89**±0.30	50.97±9.8	**1.76**±0.16
path-05	o.o.m.	**7.53**±0.48	400.87±15.2	**2.03**±0.25
path-06	o.o.m.	**8.80**±0.25	o.o.m.	**2.18**±0.14
path-07	o.o.m.	**9.76**±1.21	o.o.m.	**2.38**±0.26
path-08	o.o.m.	**10.05**±1.49	o.o.m.	**2.53**±0.30
tree-01	539.11±22.4	**6.53**±1.11	25.96±4.5	**1.47**±0.28
tree-02	o.o.m.	**7.29**±0.73	328.88±11.5	**1.69**±0.16
tree-03	o.o.m.	**8.16**±0.66	o.o.m.	**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 ±.

Query/Queries	#joins (avg)	Ref	GuAO$^+$	GuAO$^+$ Speedup
STATS-CEB e2e	3.33	1558±7.3	**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±75	**3491.06**±27	1.08 x
TPC-H Ex.1 SF200	4	168.4±8.9	**105.11**±3.9	1.60 x
LSQB Q1 SF300	9	3096±232	**688**±23	4.57 x
LSQB Q4 SF300	3	602±37	**592**±9	1.02x
TPC-DS e2e SF100	2.52	5154.5	**5047.5**	1.02 x

with the first run being a warm-up run to avoid skew from initial table loads. We report statistics over the final five runs, including the mean execution time and standard deviation. 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).

Overall, the evaluation shows that our optimisation is broadly applicable and yields substantial performance improvements. On STATS-CEB, we reduce end-to-end runtime by more than 24×, and for LSQB (Q1) by over 4.5×. On JOB, TPC-H, and TPC-DS, the gains are more modest (1–1.6×) but still consistent across queries. Importantly even in these simple cases, very few joins, all on foreign keys, we *never observe performance degradation*. For SNAP graph queries, the difference is dramatic: GuAO$^+$ executes all path and tree queries in under 11 seconds, while Spark SQL frequently times out or runs out of memory.

4 Conclusion

We identified a widely applicable class of aggregate database queries that can be evaluated without materialising, or even computing, joins. Our Spark SQL implementation integrates seamlessly into existing systems, and experimental results demonstrate substantial performance gains. Future work includes lifting the restrictions of acyclicity and guardedness, for example via decomposition techniques and richer information propagation.

References

1 Georg Gottlob, Matthias Lanzinger, Davide Mario Longo, Cem Okulmus, Reinhard Pichler, and Alexander Selzer. Reaching back to move forward: Using old ideas to achieve a new level of query optimization (short paper). In *Proceedings AMW*, volume 3409 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2023. URL: https://ceur-ws.org/Vol-3409/paper6.pdf.

2 Georg Gottlob, Matthias Lanzinger, Davide Mario Longo, Cem Okulmus, Reinhard Pichler, and Alexander Selzer. Structure-guided query evaluation: Towards bridging the gap from theory to practice. *CoRR*, abs/2303.02723, 2023. doi:10.48550/arXiv.2303.02723.

3 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. Cardinality estimation in DBMS: A comprehensive benchmark evaluation. *Proceedings VLDB Endow.*, 15(4):752–765, 2021. doi:10.14778/3503585.3503586.

4 Manas R. Joglekar, Rohan Puttagunta, and Christopher Ré. AJAR: aggregations and joins over annotated relations. In *Proceedings PODS*, pages 91–106. ACM, 2016. doi:10.1145/2902251.2902293.

5 Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: questions asked frequently. In *Proceedings PODS*, pages 13–28. ACM, 2016. doi:10.1145/2902251.2902280.

6 Matthias Lanzinger, Reinhard Pichler, and Alexander Selzer. Avoiding materialisation for guarded aggregate queries. *Proc. VLDB Endow.*, 18(5):1398–1411, 2025. doi:10.14778/3718057.3718068.

7 Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, 2015. doi:10.14778/2850583.2850594.

8 Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

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

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

11 TPC-DS. TPC-DS benchmark. https://www.tpc.org/tpcds/.

12 TPC-H. TPC-H benchmark. https://www.tpc.org/tpch/.

13 Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings VLDB*, pages 82–94. VLDB, 1981.