# Avoiding Materialisation for Guarded Aggregate Queries

Matthias Lanzinger[1], Reinhard Pichler[1] and Alexander Selzer[1]

[1]TU Wien, Vienna, Austria

## 1. Introduction

Despite significant progress in query optimization techniques over the past decades and the advent of powerful computer infrastructure (including cluster environments), DBMSs still struggle with the potential explosion of intermediate results, even if the final output is small. This is particularly glaring in the context of analytical queries which often combine data from many tables to ultimately only produce comparatively tiny aggregate results.

Traditionally, database engines try to avoid expensive intermediate blow-up by searching for an optimal join order. More recently, worst-case optimal joins, which at least guarantee to 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 (see, e.g., [1]). Furthermore, the problem of big intermediate results holds all the same even if joins are made only along foreign-key relationships [2].

By a landmark result of Yannakakis [3], we know that materialisation of unnecessary intermediate results can be avoided for acyclic conjunctive queries (ACQs) by eliminating all dangling tuples (i.e., tuples not contributing to the final join result) via semi-joins. However, even if dangling tuples have been eliminated the intermediate results produced in the join phase may still become prohibitively big. This situation is highly unsatisfactory. Especially in aggregate queries, where only a restricted amount of information is ultimately extracted from the result of a join query, we would like to avoid the materialisation of the join result altogether.

Indeed, it is well known that, in case of Boolean 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. In [4, 5], it was investigated how variations of the same algorithmic idea also apply to counting the answers to conjunctive queries (i.e., join queries with COUNT aggregates). Subsequently, these ideas were extended to more general aggregate queries in the FAQ-framework (Functional Aggregate Queries) [6] and, similarly, under the name AJAR (Aggregations and Joins over Annotated Relations) in [7].

The algorithmic results in these works generally rely on the avoidance of exponential intermediate blow-up. However, these methods, analogous to Yannakakis' algorithm, are incompatible with the execution engines of typical relational DBMSs. **In this work, we show that, for**

*AMW 2024: 16th Alberto Mendelzon International Workshop on Foundations of Data Management, September 30th–October 4th, 2024, Mexico City, Mexico*

matthias.lanzinger@tuwien.ac.at (M. Lanzinger); reinhard.pichler@tuwien.ac.at (R. Pichler); alexander.selzer@tuwien.ac.at (A. Selzer)
0000-0002-7601-3727 (M. Lanzinger); 0000-0002-1760-122X (R. Pichler); 0000-0002-6867-5448 (A. Selzer)

***guarded* acyclic aggregate queries, Yannakakis-style query execution can actually be naturally integrated into standard SQL execution engines.** As a result, intermediate materialisation of join results can often be drastically reduced or even avoided entirely when executing aggregate queries in relational DBMSs. We have implemented our approach in Spark SQL and tested it on various standard benchmarks. The experimental results reported in Section 3 show significant improvements can be achieved for queries that fall into our targeted class. Full details of this work are provided in [8].

## 2. Guarded Aggregate Queries

Recently, in [9] and [10], 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 $\mathrm{Q} = \gamma_{g_1,\dots,g_k,\ A_1(a_1),\dots,A_m(a_m)}(Q_C)$ with $Q_C = \pi_U(R_1 \bowtie \cdots \bowtie R_n))$ is 0MA if it satisfies the following conditions:

- *Guardedness*, meaning that all grouping attributes $g_1, \dots, g_k$ and all attributes $a_1, \dots, a_m$ occurring in the aggregate expressions $A_1(a_1), \dots, A_m(a_m)$ occur in some relation $R_i$.
- *Set-safety* of the aggregate functions $A_1 \dots, A_m$, meaning that duplicate elimination applied to the inner expression $\pi_U(R_1 \bowtie \cdots \bowtie R_n)$ does not alter the result of the grouping and aggregate expression $\gamma_{g_1,\dots,g_n,\ A_1(a_1),\dots,A_m(a_m)}$.

It is known (see, [9]) that for queries in the class *0MA*, materialisation of join results can be completely avoided. However, it is limited to a subset of the available SQL aggregate functions. Heavily used analytical functions such as `COUNT` without `DISTINCT` as well as `SUM`, `AVG`, or `MEDIAN` depend on full joins. In [4], it was shown how Yannakakis' algorithm can be extended to ACQs with a `COUNT(*)` aggregate on top. We adapt this approach to integrate it into the logical QEP of relational DBMSs and we further extend it to related aggregates such as `SUM`, `AVG`, and `MEDIAN`. To this end, we keep the requirements of acyclicity and guardedness, while the set-safety requirement is now dropped. Our technique also works for `COUNT(*)`, which we can view as counting the number of non-null entries grouped by the empty set of attributes; hence, it is trivially guarded. As in the 0MA-case, we start by checking acyclicity and, simultaneously, computing a join tree of the join query $Q_C$. If $Q_C$ is acyclic and guarded, we take the node labelled by the "guard" as the root of the join tree. Of course, in case of `COUNT(*)`, we may choose any node as the root of the join tree.

The key idea is to propagate frequencies up the join tree rather than duplicating tuples (cf. [4]). This propagation is realised by recursively constructing extended Relational Algebra expressions $Freq(u)$ for every node $u$ of the join tree as follows: Every relation of the join query $Q_C$ is extended by an additional attribute where we store frequency information for each tuple. We write $c_u$ to denote this additional attribute for the relation at node $u$ and we write $\bar{A}_u$ for the remaining attributes of that relation. If $u$ is a leaf node of the join tree labelled by relation $R$, then we initialise the attribute $c_u$ to 1. Formally, we thus have $Freq(u) = R \times \{(1)\}$.

Now consider an internal node $u$ of the join tree with child nodes $u_1, \dots, u_k$. Again, we assume that $u$ is labelled by some relation $R$ with attributes $\bar{A}_u$ and we write $c_u$ for the additional attribute used for keeping track of frequencies. The extended Relational Algebra expression $\mathit{Freq}(u)$ is constructed iteratively by defining subexpressions $\mathit{Freq}_i(u)$ with $i \in \{0, \dots, k\}$. To avoid confusion, we refer to the frequency attribute of such a subexpression $\mathit{Freq}_i(u)$ as $c_u^i$. That is, each relation $\mathit{Freq}_i(u)$ consists of the same attributes $\bar{A}_u$ plus the additional frequency attribute $c_u^i$. Then we define $\mathit{Freq}_i(u)$ for every $i \in \{0, \dots, k\}$ and, ultimately, $\mathit{Freq}(u)$ as follows:

$$\mathit{Freq}_0(u) := R \times \{(1)\}$$
$$\mathit{Freq}_i(u) := \gamma_{\bar{A}_u, c_u^i \leftarrow \mathrm{SUM}(c_u^{i-1} \cdot c_{u_i})}(\mathit{Freq}_{i-1}(u) \bowtie \mathit{Freq}(u_i))$$
$$\mathit{Freq}(u) := \rho_{c_u \leftarrow c_u^k}(\mathit{Freq}_k(u))$$

Intuitively, after initialising $c_u^0$ to 1 in $\mathit{Freq}_0(u)$, the frequency values $c_u^1, \dots, c_u^k$ are obtained by grouping over the attributes $\bar{A}_u$ of $R$ and computing the number of possible extensions of each tuple $\bar{a}$ in $R$ to the relations labelling the nodes in the subtrees rooted at $u_1, \dots, u_k$. By the connectedness condition of join trees, these extensions are independent of each other, i.e., they share no attributes outside $\bar{A}_u$. Moreover, the frequency attributes $c_u^1, \dots, c_u^k$ are functionally dependent on the attributes $\bar{A}_u$. Hence, by distributivity, the value of $c_u^k$ obtained by iterated summation and multiplication for given tuple $\bar{a}$ of $R$ is equal to computing, for every $i \in \{1, \dots, k\}$ the sum $s_i$ of the frequencies of all join partners of $\bar{a}$ in $\mathit{Freq}(u_i)$ and then computing their product, i.e., $c_u = c_u^k = \Pi_{i=1}^k s_i$.

It remains to modify the aggregate functions COUNT($*$), COUNT($A$), SUM($A$), and AVG($A$) so that they can operate on tuples with frequencies. For instance, let $c_r$ denote the frequency attribute in $\mathit{Freq}(r)$ and let $A$ be an attribute of the relation $R$ labelling the root node $r$ of the join tree. Then, in SQL-notation, we can rewrite common aggregate expressions as follows. Note that the PERCENTILE function is not part of the ANSI SQL standard, but it is provided in Spark SQL, which we used for our prototype implementation.

- COUNT($*$) $\rightarrow$ SUM($c_r$)
- COUNT($A$) $\rightarrow$ SUM(IF(ISNULL($A$), 0, $c_r$))
- SUM($A$) $\rightarrow$ SUM($A \cdot c_r$)
- AVG($A$) $\rightarrow$ SUM($A \cdot c_r$)/COUNT($A$)
- MEDIAN($A$) $\rightarrow$ PERCENTILE(0.5, $A$, $c_r$)

## 3. Implementation and Results

The approach outlined in Section 2 can be fully realised by rewriting logical query plans, even in systems that only execute query plans based on classical two-way join trees. We have implemented these optimisations for guarded aggregate queries in the form of standard logical optimisation rules in Spark SQL. Recall that the $\mathit{Freq}_i$ operator allowed us to drastically reduce the number of joins needed for aggregate evaluation. In our Spark SQL implementation, we have managed to completely eliminate the need for joins in case of guarded aggregation by introducing a new physical operator *FreqJoin*. Intuitively, it does the work of $\mathit{Freq}_i$ in the style of semi-joins by first (say in relation $S$) summing up the frequencies of all tuples in $S$ with the same join partner $(r, c_r)$ in the other relation (say $R$) and then multiplying the frequency $c_r$ of $r$ with this sum.

| | wiki-topcats | | com-DBLP | |
|---|---|---|---|---|
| Query | Ref | Opt$^+$ | Ref | Opt$^+$ |
| path-03 | X | **23.71**±0.54 | 6.32±1.1 | **1.59**±0.12 |
| path-04 | X | **25.94**±1.12 | 50.97±9.8 | **1.76**±0.16 |
| path-05 | X | **27.46**±0.64 | 400.87±15.2 | **2.03**±0.25 |
| path-06 | X | **30.16**±1.00 | X | **2.18**±0.14 |
| path-07 | X | **33.32**±1.62 | X | **2.38**±0.26 |
| path-08 | X | **34.49**±0.68 | X | **2.53**±0.30 |
| tree-01 | X | **25.44**±0.43 | 25.96±4.5 | **1.47**±0.28 |
| tree-02 | X | **27.64**±0.57 | 328.88±11.5 | **1.69**±0.16 |
| tree-03 | X | **30.70**±1.01 | X | **1.99**±0.16 |

(a) Performance on SNAP graphs (X means Spark SQL fails by running out of memory).

| Query | Ref | Opt$^+$ | Speedup |
|---|---|---|---|
| STATS-CEB e2e | 1558±7.3 | **64.8**±7.9 | 24.04 x |
| JOB Q 2 | 5.6±1.21 | **4.72**±0.57 | 1.19 x |
| JOB Q 3 | 5.2±0.32 | **4.70**±0.22 | 1.11 x |
| JOB Q 5 | 1.23±1.22 | **1.00**±0.98 | 1.23 x |
| JOB Q 17 | 118.5±32 | **35.69**±0.59 | 3.32 x |
| JOB Q 20 | 23.98±1.60 | **21.72**±1.92 | 1.1 x |
| TPC-H Q 2 SF200 | 179.4±6.5 | **160.6**±3.7 | 1.12 x |
| TPC-H Q 11 SF200 | 361.0±13.3 | **341.6**±19.2 | 1.06 x |
| TPC-H V.1 SF200 | 168.4±4.4 | **105.11**±3.9 | 1.6 x |
| LSQB Q 1 SF300 | 3096±232 | **688**±23 | 4.57 x |
| LSQB Q 4 SF300 | 602±37 | **592**±9 | 1.02x |

(b) Summary of the impact of aggregate optimisation on execution times (seconds).

**Figure 1:** Summary of Experimental Results.

We evaluate the performance of the optimisation over 5 benchmark databases / datasets with different characteristics: *Join Order Benchmark (JOB)* [11], *STATS / STATS-CEB* [12], *Large-Scale Subgraph Query Benchmark (LSQB)* [13], *SNAP (Stanford Network Analysis Project* [14]), and *TPC-H* [15]. For further details on the experiments, see [8].

The overall performance of our proposed optimisations on the applicable queries is summarised in Table 1b. The numbers reported there are mean times over 5 runs of the same query with standard deviations given after ±. Our experiments on the SNAP graphs specifically are summarised in Table 1a. The fastest execution time achieved for each case is printed in boldface. In both tables, we refer to the reference performance of Spark SQL without any alterations as *Ref*. The results obtained by applications of the logical QEP optimisations are referred to as *Opt*$^+$. The speed-up achieved by *Opt*$^+$ over *Ref* is explicitly stated in Table 1b in the column *Opt*$^+$ *Speedup*. Since our optimisations apply to all 146 queries of STATS-CEB, we report the end-to-end time of executing *all* queries in the benchmark.

*Avoiding Materialisation.* To validate that the significant speedups are a result of decreased materialisation we perform additional experiments for the STATS-CEB queries where we record the peak number of materialised tuples during query execution. We see that the reduction in materialisation is immense, especially in the queries that are most challenging for original SparkSQL. In particular, in the 20 hardest queries (by runtime) for SparkSQL, we observe a reduction of peak materialised tuples by at least 3 orders of magnitude when using *Opt*$^+$.

## 4. Conclusion

We propose the integration of optimisation techniques for certain kinds of aggregate queries into relational DBMSs to eliminate the materialisation of intermediate join results. We have shown how queries with `COUNT` and related aggregates (such as `SUM`, `AVG`, and `MEDIAN`) can be evaluated more efficiently while operating fully within the standard two-way join plan paradigm of query evaluation engines. We have implemented these optimisations into Spark SQL and our experimental evaluation confirms that these techniques can provide significant performance improvements and avoid large amounts of unnecessary materialisation in a variety of settings.

## Acknowledgments

This work has been supported by the Vienna Science and Technology Fund (WWTF) [10.47379/ICT2201, 10.47379/VRG18013, 10.47379/NXT22018]; and the Christian Doppler Research Association (CDG) JRC LIVE. We also acknowledge the assistance of the TU.it dataLAB Big Data team at TU-Wien.

## References

[1] A. Atserias, M. Grohe, D. Marx, Size bounds and query plans for relational joins, SIAM J. Comput. 42 (2013) 1737–1767. URL: https://doi.org/10.1137/110859440. doi:10.1137/110859440.

[2] R. Mancini, S. Karthik, B. Chandra, V. Mageirakos, A. Ailamaki, Efficient Massively Parallel Join Optimization for Large Queries, in: Proceedings SIGMOD, ACM, 2022, pp. 122–135. URL: https://doi.org/10.1145/3514221.3517871. doi:10.1145/3514221.3517871.

[3] M. Yannakakis, Algorithms for acyclic database schemes, in: Proceedings VLDB, VLDB, 1981, pp. 82–94.

[4] R. Pichler, S. Skritek, Tractable counting of the answers to conjunctive queries, J. Comput. Syst. Sci. 79 (2013) 984–1001. URL: https://doi.org/10.1016/j.jcss.2013.01.012. doi:10.1016/j.jcss.2013.01.012.

[5] A. Durand, S. Mengel, Structural tractability of counting of solutions to conjunctive queries, Theory Comput. Syst. 57 (2015) 1202–1249. URL: https://doi.org/10.1007/s00224-014-9543-y. doi:10.1007/S00224-014-9543-Y.

[6] M. A. Khamis, H. Q. Ngo, A. Rudra, FAQ: questions asked frequently, in: Proceedings PODS, ACM, 2016, pp. 13–28. URL: https://doi.org/10.1145/2902251.2902280. doi:10.1145/2902251.2902280.

[7] M. R. Joglekar, R. Puttagunta, C. Ré, AJAR: aggregations and joins over annotated relations, in: Proceedings PODS, ACM, 2016, pp. 91–106. URL: https://doi.org/10.1145/2902251.2902293. doi:10.1145/2902251.2902293.

[8] M. Lanzinger, R. Pichler, A. Selzer, Avoiding materialisation for guarded aggregate queries, CoRR abs/2406.17076 (2024). URL: https://doi.org/10.48550/arXiv.2406.17076. doi:10.48550/ARXIV.2406.17076. arXiv:2406.17076.

[9] G. Gottlob, M. Lanzinger, D. M. Longo, C. Okulmus, R. Pichler, A. Selzer, Structure-guided query evaluation: Towards bridging the gap from theory to practice, CoRR abs/2303.02723 (2023). URL: https://doi.org/10.48550/arXiv.2303.02723. doi:10.48550/arXiv.2303.02723. arXiv:2303.02723.

[10] G. Gottlob, M. Lanzinger, D. M. Longo, C. Okulmus, R. Pichler, A. 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.

[11] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, T. Neumann, How good are query optimizers, really?, Proc. VLDB Endow. 9 (2015) 204–215. URL: http://www.vldb.org/pvldb/vol9/p204-leis.pdf. doi:10.14778/2850583.2850594.

[12] Y. Han, Z. Wu, P. Wu, R. Zhu, J. Yang, L. W. Tan, K. Zeng, G. Cong, Y. Qin, A. Pfadler, Z. Qian, J. Zhou, J. Li, B. Cui, Cardinality estimation in DBMS: A comprehensive benchmark evaluation, Proc. VLDB Endow. 15 (2021) 752–765. URL: https://www.vldb.org/pvldb/vol15/p752-zhu.pdf. doi:10.14778/3503585.3503586.

[13] A. Mhedhbi, M. Lissandrini, L. Kuiper, J. Waudby, G. Szárnyas, LSQB: a large-scale subgraph query benchmark, in: Proceedings GRADES, ACM, 2021, pp. 8:1–8:11. URL: https://doi.org/10.1145/3461837.3464516. doi:10.1145/3461837.3464516.

[14] J. Leskovec, A. Krevl, SNAP Datasets: Stanford large network dataset collection, http://snap.stanford.edu/data, 2014.

[15] TPC-H, Tpc-h benchmark, https://www.tpc.org/tpch/, 2022.