Avoiding Materialisation for Guarded Aggregate Queries[†]

Matthias Lanzinger, Reinhard Pichler, and Alexander Selzer

TU Wien

[†] talk mainly based on: M. Lanzinger, P., A. Selzer:
Avoiding Materialisation for Guarded Aggregate Queries.
CoRR abs/2406.17076 (2024).
accepted for publication at VLDB 2025.

Acyclic Conjunctive Queries

The cost of joins.

- Processing (not necessarily large) join queries remains a challenge, even for modern DBMSs: *explosion of intermediate results*
- However, the vast majority of queries from benchmarks and query logs are acyclic (*ACQs*) or almost-acyclic.
- Yannakakis' algorithm allows us to answer ACQs without any "useless" intermediate results.

Definition.

- An *Acyclic Conjunctive Query (ACQ)* is a CQ that has a *join tree*.
- A *join tree* is a rooted, labelled tree $\langle T, r, \lambda \rangle$ with root r, such that
  - λ is a bijection that assigns to each node of T one of the relations in $\{R_1, \ldots, R_n\}$ and
  - λ satisfies the so-called *connectedness condition*, i.e., if some attribute A occurs in both relations $\lambda(u_i)$ and $\lambda(u_j)$ for two nodes u_i and u_j, then A occurs in the relation $\lambda(u)$ for every node u along the path between u_i and u_j.

Yannakakis' algorithm

Theorem.

ACQs can be evaluated in time $O((||D|| + ||Q(D)||) \cdot ||Q||)$ using *Yannakakis' algorithm*, i.e., linear w.r.t. the size of the input and output data and w.r.t. the size of the query

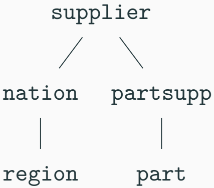
Yannakakis' algorithm.

involves 3 traversals of the join tree T which consist of

1. a bottom-up traversal of semi-joins
2. a top-down traversal of semi-joins
3. a traversal of full joins.

Running Example: Yannakakis' Algorithm

```
SELECT s_suppkey, s_nationkey, 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')
```

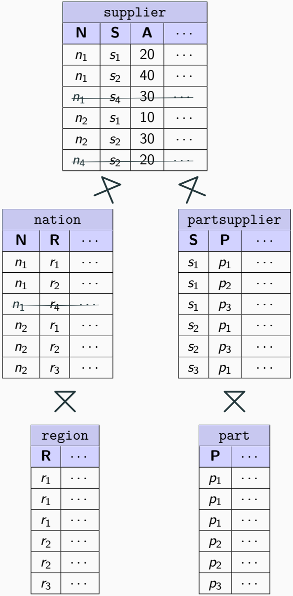


supplier

nation partsupp

region part

Bottom-up Traversal of Semi-Joins



supplier			
N	**S**	**A**	···
n_1	s_1	20	···
n_1	s_2	40	···
~~n_1~~	~~s_4~~	~~30~~	~~···~~
n_2	s_1	10	···
n_2	s_2	30	···
~~n_4~~	~~s_2~~	~~20~~	~~···~~

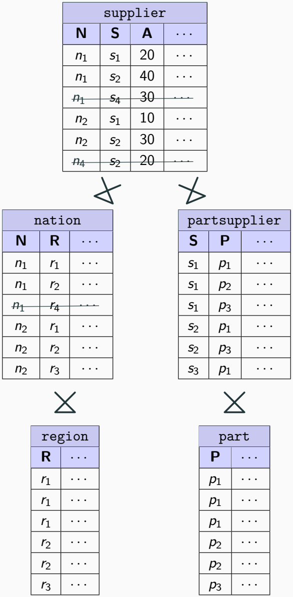
nation		
N	**R**	···
n_1	r_1	···
n_1	r_2	···
~~n_1~~	~~r_4~~	~~···~~
n_2	r_1	···
n_2	r_2	···
n_2	r_3	···

partsupplier		
S	**P**	···
s_1	p_1	···
s_1	p_2	···
s_1	p_3	···
s_2	p_1	···
s_2	p_3	···
s_3	p_1	···

region	
R	···
r_1	···
r_1	···
r_1	···
r_2	···
r_2	···
r_3	···

part	
P	···
p_1	···
p_1	···
p_1	···
p_2	···
p_2	···
p_3	···

Top-Down Traversal of Semi-Joins



`supplier`

N	**S**	**A**	⋯
n_1	s_1	20	⋯
n_1	s_2	40	⋯
~~n_1~~	~~s_4~~	~~30~~	~~⋯~~
n_2	s_1	10	⋯
n_2	s_2	30	⋯
~~n_4~~	~~s_2~~	~~20~~	~~⋯~~

`nation`

N	**R**	⋯
n_1	r_1	⋯
n_1	r_2	⋯
~~n_1~~	~~r_4~~	~~⋯~~
n_2	r_1	⋯
n_2	r_2	⋯
n_2	r_3	⋯

`partsupplier`

S	**P**	⋯
s_1	p_1	⋯
s_1	p_2	⋯
s_1	p_3	⋯
s_2	p_1	⋯
s_2	p_3	⋯
s_3	p_1	⋯

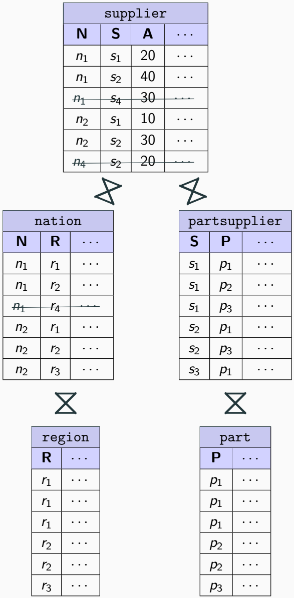
`region`

R	⋯
r_1	⋯
r_1	⋯
r_1	⋯
r_2	⋯
r_2	⋯
r_3	⋯

`part`

P	⋯
p_1	⋯
p_1	⋯
p_1	⋯
p_2	⋯
p_2	⋯
p_3	⋯

Bottm-Up Traversal of Joins



supplier

N	S	A	⋯
n_1	s_1	20	⋯
n_1	s_2	40	⋯
~~n_1~~	~~s_4~~	~~30~~	~~⋯~~
n_2	s_1	10	⋯
n_2	s_2	30	⋯
~~n_4~~	~~s_2~~	~~20~~	~~⋯~~

$\Join$ $\Join$

nation

N	R	⋯
n_1	r_1	⋯
n_1	r_2	⋯
~~n_1~~	~~r_4~~	~~⋯~~
n_2	r_1	⋯
n_2	r_2	⋯
n_2	r_3	⋯

partsupplier

S	P	⋯
s_1	p_1	⋯
s_1	p_2	⋯
s_1	p_3	⋯
s_2	p_1	⋯
s_2	p_3	⋯
s_3	p_1	⋯

$\Join$ $\Join$

region

R	⋯
r_1	⋯
r_1	⋯
r_1	⋯
r_2	⋯
r_2	⋯
r_3	⋯

part

P	⋯
p_1	⋯
p_1	⋯
p_1	⋯
p_2	⋯
p_2	⋯
p_3	⋯

=

result

N	S	A	⋯
n_1	s_1	20	⋯
⋯	⋯	⋯	⋯
n_1	s_2	40	⋯
⋯	⋯	⋯	⋯
n_2	s_1	10	⋯
⋯	⋯	⋯	⋯
n_2	s_2	30	⋯

Summary

Correctness of Yannakakis' algorithm.

Let $R_{i_1}, \ldots, R_{i_k}$ be the relations at the subtree T_u rooted at node u.
Let $R'(u)$ be the relation at node u after each traversal of the join tree.
Let (1), (2), (3) denote the 3 traversals of the join tree.
Then it holds:

- after (1), we have $R'(u) = \pi_{Att(u)}(R_{i_1} \bowtie \ldots \bowtie R_{i_\ell})$,
- after (2), we have $R'(u) = \pi_{Att(u)}(R_1 \bowtie \ldots \bowtie R_n)$,
- after (3), we have $R'(u) = \pi_{Att(T_u)}(R_1 \bowtie \ldots \bowtie R_n)$.

Advantage of Yannakakis' algorithm.

- The semi-joins remove all dangling tuples.
- All intermediate results of the joins end up in the final result.

Aggregate Queries

Cost of the joins.

- The joins are cheap if they are via foreign keys from the parent node to the child nodes.
- However, in general, despite the deletion of dangling tuples, the join step may still be expensive.

Analytical queries.

- Analytical queries tend to combine several tables, but output only a comparatively small aggregated final result.
- Usual strategy: computing the aggregates as post-processing (after the evaluation of the joins query).
- **Question.** Can we do better?
  That is: *evaluate the query without computing the joins!*

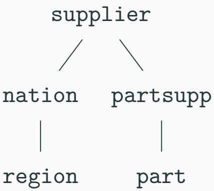
Joinless Evaluation of Queries

Roadmap.

- Boolean ACQs
- Zero Materialization Aggregate (0MA) Queries
- Guarded Aggregate Queries
- Piecewise Guarded Aggregate Queries

Example: Boolean ACQ

```sql
SELECT ... WHERE EXISTS
(SELECT * FROM
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')
  )
```

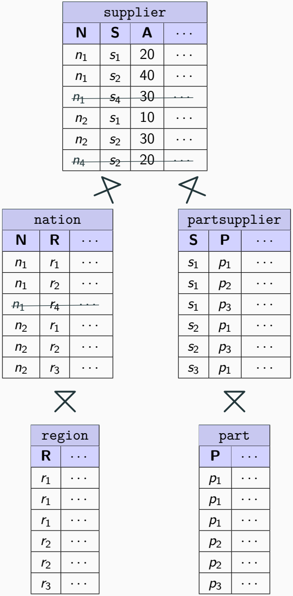


supplier

nation partsupp

region part

Bottom-up Traversal of Semi-Joins



supplier

N	**S**	**A**	···
n_1	s_1	20	···
n_1	s_2	40	···
~~n_1~~	~~s_4~~	~~30~~	~~···~~
n_2	s_1	10	···
n_2	s_2	30	···
~~n_4~~	~~s_2~~	~~20~~	~~···~~

nation

N	**R**	···
n_1	r_1	···
n_1	r_2	···
~~n_1~~	~~r_4~~	~~···~~
n_2	r_1	···
n_2	r_2	···
n_2	r_3	···

partsupplier

S	**P**	···
s_1	p_1	···
s_1	p_2	···
s_1	p_3	···
s_2	p_1	···
s_2	p_3	···
s_3	p_1	···

region

R	···
r_1	···
r_1	···
r_1	···
r_2	···
r_2	···
r_3	···

part

P	···
p_1	···
p_1	···
p_1	···
p_2	···
p_2	···
p_3	···

Aggregate Queries Considered Here

Acyclic Conjunctive Queries with aggregation, i.e.
Extended Relational Algebra-expressions of the following form:

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

(or SQL `SELECT-FROM-WHERE-GROUP BY` queries), where:

- $R_1 \bowtie \cdots \bowtie R_n$ is an ACQ
- $\gamma[g_1, \ldots, g_\ell,\ A_1(a_1), \ldots, A_m(a_m)]$ denotes the grouping operation
- $g_1, \ldots, g_\ell$ are attributes occurring in the relations $R_1, \ldots, R_n$,
- $A_1, \ldots, A_m$ are (standard SQL) aggregate functions such as `MIN`, `MAX`, `COUNT`, `SUM`, `AVG`, `MEDIAN`, etc.,
- $a_1, \ldots, a_m$ are expressions over attributes from $R_1, \ldots, R_n$.

Zero Materialization Aggregate (0MA) Queries

Definition [Zero Materialization Aggregate (0MA) Queries][1]

Aggregate Queries $\gamma[g_1, \ldots, g_\ell,\ A_1(a_1), \ldots, A_m(a_m)]\big(R_1 \bowtie \cdots \bowtie R_n\big)$, with the following properties:

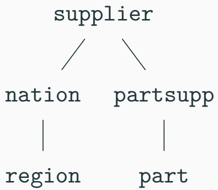
- *Set-safety*: an aggregate function is *set-safe*, if its value is invariant under duplicate elimination. A query is set-safe, if all aggregates are.
- *Guardedness*: a query is guarded, if there exists a *single* relation R_i that contains all grouping attributes $g_1, \ldots, g_\ell$ and all attributes occurring in the aggregate expressions $A_1(a_1), \ldots, A_m(a_m)$.

---

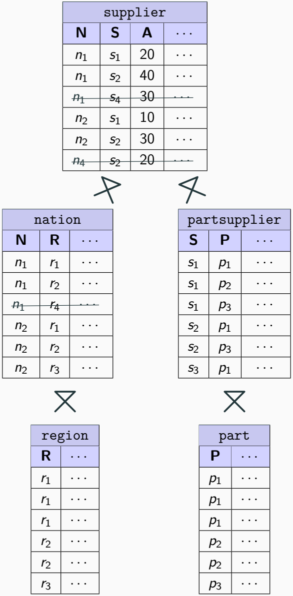
[1] G. Gottlob, M. Lanzinger, D. Longo, C. Okulmus, P., A. Selzer:
Structure-Guided Query Evaluation: Towards Bridging the Gap from Theory to Practice. CoRR abs/2303.02723 (2023).

Example: 0MA Query

```
SELECT MIN(s_acctbal), MAX(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')
GROUP BY s_nationkey
```



Bottom-Up Traversal of Semi-Joins



supplier			
N	**S**	**A**	$\cdots$
n_1	s_1	20	$\cdots$
n_1	s_2	40	$\cdots$
~~n_1~~	~~s_4~~	~~30~~	~~$\cdots$~~
n_2	s_1	10	$\cdots$
n_2	s_2	30	$\cdots$
~~n_4~~	~~s_2~~	~~20~~	~~$\cdots$~~

nation		
N	**R**	$\cdots$
n_1	r_1	$\cdots$
n_1	r_2	$\cdots$
~~n_1~~	~~r_4~~	~~$\cdots$~~
n_2	r_1	$\cdots$
n_2	r_2	$\cdots$
n_2	r_3	$\cdots$

partsupplier		
S	**P**	$\cdots$
s_1	p_1	$\cdots$
s_1	p_2	$\cdots$
s_1	p_3	$\cdots$
s_2	p_1	$\cdots$
s_2	p_3	$\cdots$
s_3	p_1	$\cdots$

region	
R	$\cdots$
r_1	$\cdots$
r_1	$\cdots$
r_1	$\cdots$
r_2	$\cdots$
r_2	$\cdots$
r_3	$\cdots$

part	
P	$\cdots$
p_1	$\cdots$
p_1	$\cdots$
p_1	$\cdots$
p_2	$\cdots$
p_2	$\cdots$
p_3	$\cdots$

Guarded Aggregate Queries

Motivation and Definition.

- 0MA queries are very restricted.
- *Guarded Aggregate Queries*: lift the set-safety condition. That is: we only require guardedness.
- This means: we allow arbitrary (standard SQL) aggregate functions; in particular, `COUNT`, SUM, etc.

Idea. Efficient frequency propagation[2]

Compute $Freq(u)$ (i.e., original relation extended by a row count) at node u with child nodes $u_1, \ldots, u_k$ in a bottom-up traversal of the join tree.

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

---

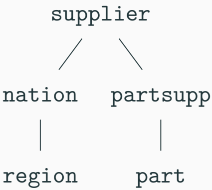
[2] P., S. Skritek: Tractable counting of the answers to conjunctive queries. J. Comput. Syst. Sci. 79, 6 (2013).

Efficient Counting and Aggregation

- Frequencies can be propagated up the join tree efficiently (essentially by an extension of the semi-joins)
- Using these frequencies, we can reconstruct the original aggregates without actually evaluating the join query.
- Let c_r denote the count-attribute at the root node of a join tree.
- We can rewrite all aggregate expressions, e.g. (in SQL notation):
    - $\texttt{COUNT}(*) \rightarrow \texttt{SUM}(c_r)$
    - $\texttt{COUNT}(B) \rightarrow \texttt{SUM}(\texttt{IF}(\texttt{ISNULL}(B), 0, c_r))$
    - $\texttt{SUM}(B) \rightarrow \texttt{SUM}(B \cdot c_r)$
    - $\texttt{AVG}(B) \rightarrow \texttt{SUM}(B \cdot c_r)/\texttt{COUNT}(B)$

Example: Guarded Aggregate Query

```
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')
GROUP BY s_nationkey
```

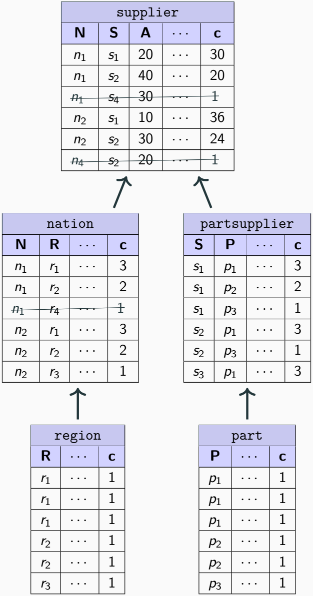


supplier

nation partsupp

region part

Bottom-Up Traversal with Frequency Propagation



supplier

N	S	A	⋯	c
n_1	s_1	20	⋯	30
n_1	s_2	40	⋯	20
~~n_1~~	~~s_4~~	~~30~~	~~⋯~~	~~1~~
n_2	s_1	10	⋯	36
n_2	s_2	30	⋯	24
~~n_4~~	~~s_2~~	~~20~~	~~⋯~~	~~1~~

nation

N	R	⋯	c
n_1	r_1	⋯	3
n_1	r_2	⋯	2
~~n_1~~	~~r_4~~	~~⋯~~	~~1~~
n_2	r_1	⋯	3
n_2	r_2	⋯	2
n_2	r_3	⋯	1

partsupplier

S	P	⋯	c
s_1	p_1	⋯	3
s_1	p_2	⋯	2
s_1	p_3	⋯	1
s_2	p_1	⋯	3
s_2	p_3	⋯	1
s_3	p_1	⋯	3

region

R	⋯	c
r_1	⋯	1
r_1	⋯	1
r_1	⋯	1
r_2	⋯	1
r_2	⋯	1
r_3	⋯	1

part

P	⋯	c
p_1	⋯	1
p_1	⋯	1
p_1	⋯	1
p_2	⋯	1
p_2	⋯	1
p_3	⋯	1

Piecewise Guarded Aggregate Queries

Motivation.

- Requiring a single guard for the grouping attributes *and all* attributes used in aggregate expressions is still very restrictive.
- Relax this condition for the most common aggregate functions, namely `MIN`, `MAX`, `COUNT`, `SUM`, and `AVG`.

Definition [Piecewise Guarded Aggregate Query].

Aggregate Query $\gamma[g_1, \ldots, g_\ell,\ A_1(a_1), \ldots, A_m(a_m)]\big(R_1 \bowtie \cdots \bowtie R_n\big)$, s.t. 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 *some* 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 $A_j(a_j)$.

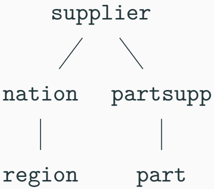
Efficient Propagation of Aggregates

Idea. Choose the guard of the grouping attributes as root of the join tree T and handle an aggregate expression $A_j(a_j)$ with $A_j \in \{\mathtt{MIN}, \mathtt{MAX}, \mathtt{SUM}, \mathtt{COUNT}\}$ that is not guarded by the root of T as follows:

- as guard, choose node w highest up in T with $Att(a_j) \subseteq Att(w)$.
- add attribute Agg_j to every node u from w up to the root r; intended meaning of the resulting relation at node u: $\gamma[Att(u), Agg_j \leftarrow A_j(a_j)]\big(\bowtie_{v \in T_u}(R(v))\big)$
- initialize Agg_j at node w: for `MIN`, `MAX` simply take the value of a_j; for `SUM`, `COUNT` also take the frequency of the tuple into account.
- propagate Agg_j to every ancestor u of w:
    - by connectedness of T: only one child v of u has attribute Agg_j;
    - propagate Agg_j from all tuples $t[v]$ in $R(v)$ to all tuples $t[u]$ in $R(u)$ which have identical values on the common attributes;
    - for `SUM`, `COUNT` also take the frequencies of the join partners of $t[u]$ in the siblings of v into account.

Example: Piecewise Guarded Aggregate Query

```
SELECT MIN(region.X), SUM(part.Y)
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')
GROUP BY s_nationkey
```



supplier

nation partsupp

region part

Bottom-Up Traversal with Aggregate Propagation

supplier

N	**S**	⋯	**c**	Agg_1	Agg_2
n_1	s_1	⋯	30	10	625
n_1	s_2	⋯	20	10	375
~~n_1~~	~~s_4~~	~~⋯~~	~~1~~	~~-~~	~~-~~
n_2	s_1	⋯	36	5	750
n_2	s_2	⋯	24	5	450
~~n_4~~	~~s_2~~	~~⋯~~	~~1~~	~~-~~	~~-~~

nation

N	**R**	⋯	**c**	Agg_1
n_1	r_1	⋯	3	10
n_1	r_2	⋯	2	20
~~n_1~~	~~r_4~~	~~⋯~~	~~1~~	~~-~~
n_2	r_1	⋯	3	10
n_2	r_2	⋯	2	20
n_2	r_3	⋯	1	5

partsupplier

S	**P**	⋯	**c**	Agg_2
s_1	p_1	⋯	3	45
s_1	p_2	⋯	2	50
s_1	p_3	⋯	1	30
s_2	p_1	⋯	3	45
s_2	p_3	⋯	1	30
s_3	p_1	⋯	3	45

region

R	**X**	⋯	**c**	Agg_1
r_1	10	⋯	1	10
r_1	20	⋯	1	20
r_1	15	⋯	1	15
r_2	20	⋯	1	20
r_2	25	⋯	1	25
r_3	5	⋯	1	5

part

P	**Y**	⋯	**c**	Agg_2
p_1	20	⋯	1	20
p_1	15	⋯	1	15
p_1	10	⋯	1	10
p_2	30	⋯	1	30
p_2	20	⋯	1	20
p_3	30	⋯	1	30

Coverage

Many applicable queries in 5 standard benchmarks:

- *JOB (Join Order Benchmark)*
- *STATS / STATS-CEB*
- *TPC-H*
- *LSQB (Large-Scale Subgraph Query Benchmark)*
- *SNAP (Stanford Network Analysis Project)* (web-Google & com-DBLP)

Benchmark	**#**	**$\bowtie$-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

Implementation and Evaluation

Implementation.

- in Spark SQL
- logical optimization: exchange the subtree in the query plan
- physical optimization: new physical operator “AggJoin”, that combines join (relation at parent and child node) followed by aggregate propagation into a semi-join-like operation.
- `https://github.com/dbai-tuw/spark-eval`

End-to-end results.

Query	# joins (mean)	Ref	GuAO	GuAO$^+$	GuAO$^+$ Speedup
STATS-CEB e2e	3.33	1558$_{\pm 7.3}$	97.9$_{\pm 6.1}$	**64.8**$_{\pm 7.9}$	24.04 x
JOB e2e	7.65	3217.84$_{\pm 106}$	-	**2189.46**$_{\pm 76}$	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$_{\pm 232}$	**677**$_{\pm 23}$	688$_{\pm 23}$	4.57 x
LSQB Q4 SF300	3	602$_{\pm 37}$	593$_{\pm 15}$	**592**$_{\pm 9}$	1.02x
TPC-DS e2e SF100	2.52	5154.5	-	5047.5	1.02 x

More Detailed Results: SNAP

	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

Conclusion

Summary of Results.

- (Piecewise) Guarded Aggregate Queries
- Physical Operator AggJoin
- Implementation in Spark SQL
- Promising empirical results

Next steps.

- Extension to cyclic queries
- Extension to unguarded queries,
  e.g., `SUM (X*Y)` for attributes from different relations