

Yannakakis⁺

Practical Acyclic Query Evaluation

CS 848 · Indexing for Modern Data Processing
October 6, 2026

Nowadays

- Big data: Analytical Processing
- Complex analytical queries: SPJA queries
- ❑ Why is efficient acyclic query processing important?
- Backbone of various analytical workloads.

Conjunctive Queries

Conjunctive Query

Definition (Conjunctive Query). We consider conjunctive queries (CQs) of the following form:

$$Q = \pi_o(R_1(A_1) \bowtie R_2(A_2) \bowtie \cdots \bowtie R_n(A_n))$$

where each $R_i(A_i)$ is a relation with a set of attributes A_i , for $i = 1, 2, \dots, n$.

Conjunctive Query (cont.)

Definition (Commutative semiring). Aggregations can be formalized using a commutative semiring $(S, \oplus, \otimes)$. S is the ground set; $\oplus$ combines alternatives and $\otimes$ combines joined contributions.

Common commutative semirings:

- Boolean: $(\{0,1\}, \vee, \wedge)$
- Sum-product: $(\mathbb{R}, +, \times)$, e.g. $\text{SUM}(A \times C)$
- Max-plus: $(\mathbb{R} \cup \{-\infty\}, \max, +)$, e.g. $\text{MAX}(A + C)$

Conjunctive Query (cont.)

Definition(annotation). Each input tuple $t \in R_i$ is associated with an *annotation* $v_i(t) \in S$. These annotations are propagated through the join and projection, as follows. The annotation for any tuple t in the join results $J = R_1(A_1) \bowtie R_2(A_2) \bowtie \cdots \bowtie R_n(A_n)$ is the $\otimes$ -aggregate of all the tuples, one from each relation, that make up t :

$$v(t) := \otimes_{R_i(A_i) \in Q} v_i(\pi_{A_i} t)$$

Then π_O performs a $\oplus$ -aggregation grouped by O , i.e., the annotation of each tuple t in the final query results is:

$$v(t) := \oplus_{\forall t' \in J, \pi_O t' = t} v'(t')$$

Conjunctive Query (cont.)

Such a conjunctive query with properly defined annotations is equivalent to the following SQL query:

SELECT	$0, \oplus (v_1 \otimes \dots \otimes v_n)$
FROM	$R_1 \text{ NATURAL JOIN } \dots \text{ NATURAL JOIN } R_n$
WHERE	$\langle \text{predicate 1} \rangle \langle \text{predicate 2} \rangle \dots$
GROUP BY	$0;$

Conjunctive Query (cont.)

- Summary of relation operators and the corresponding SQL queries, where v represents the annotation

Operator	SQL Query	Complexity
Selection($\sigma_f(R)$)	<code>SELECT * FROM R WHERE f;</code>	$O(R)$
Projection($\pi_E(R)$)	<code>SELECT E, $\oplus(v)$ AS v FROM R GROUP BY E;</code>	$O(R)$
Join($R_1 \bowtie R_2$)	<code>SELECT *, $R_1.v \otimes R_2.v$ AS v FROM R_1 NATURAL JOIN R_2;</code>	$O(R_1 + R_2 + R_1 \bowtie R_2)$
SemiJoin($R_1 \bowtie R_2$)	<code>SELECT * FROM R_1 WHERE $R_1.key$ in (SELECT DISTINCT $R_2.key$ FROM R_2);</code>	$O(R_1 + R_2)$

Conjunctive Query (cont.)

Example. TPC-H Q9-style query

```
SELECT  n_name, o_orderkey, l_returnflag, SUM(ps_supplycost * l_quantity) AS part_cost
FROM    nation, supplier, part, orders, lineitem, partsupp
WHERE   o_orderdate < DATE '1996-12-31'
AND     o_orderdate > DATE '1996-01-01'
AND     p_name LIKE '%blue%'
AND     o_orderkey = l_orderkey
AND     ps_suppkey = l_suppkey
AND     ps_partkey = l_partkey
AND     p_partkey = l_partkey
AND     s_suppkey = l_suppkey
AND     s_nationkey = n_nationkey
GROUP BY n_name, o_orderkey, l_returnflag;
```

Conjunctive Query (cont.)

Example. TPC-H Q9-style query can be represented as the following conjunctive query over the semiring $(R, +, \cdot)$:

$$Q_1 = \pi_{x_1, x_2, x_8} ((R_1(x_1, x_2, x_3, x_4) \bowtie R_2(x_2, x_5) \bowtie R_3(x_3, x_4) \\ \bowtie R_4(x_3, x_6) \bowtie R_5(x_4, x_7) \bowtie R_6(x_7, x_8))$$

$R_1, R_2, R_3, R_4, R_5, R_6$ correspond to the relations *lineitem*, *orders*, *partsupp*, *part*, *supplier*, and *nation*, respectively, while $x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8$ correspond to the attributes $l_{\text{returnflag}}$, $orderkey$, $partkey$, $supplierkey$, $o_{\text{orderdate}}$, p_{name} , $n_{\text{nationkey}}$, n_{name} .

For all tuples in R_2, R_4, R_5, R_6 , their annotations are set to 1. For each tuple $t \in R_3$, set $v(t) := p_{\text{supplycost}}$; for each tuple $t \in R_1$, set $v(t) := l_{\text{quantity}}$

Acyclic Conjunctive Queries

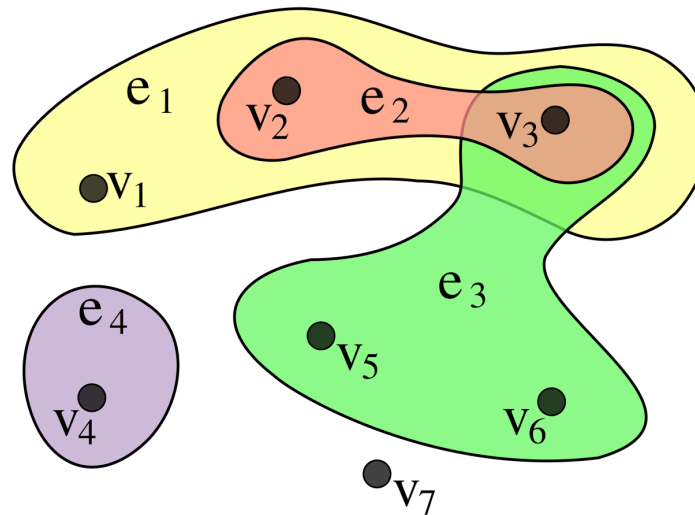
Query Hypergraph

Definition (Query Hypergraph). The hypergraph $H(q) = (V, E)$ associated to a conjunctive query q is defined as follows. The set of vertices V consists of all variables in the body of q , $\text{vars}(q)$, while the set E of hyperedges contains, for each atom in the query, the set of variables that appear in this atom.

Query Hypergraph

Example (Query Hypergraph). Figure1 below is an hypergraph example for:

$$q_0(v_1, v_2, v_3, v_4, v_5, v_6) : \neg R(v_1, v_2, v_3), S(v_2, v_3), T(v_3, v_5, v_6), G(v_4)$$



To formally define acyclicity, we will apply the following methods:

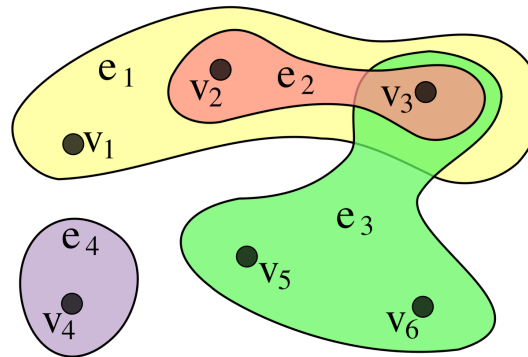
GYO

GYO Algorithm

Definition (Ear) [1]. An ear in a hypergraph H is a hyperedge e for which we can divide its nodes into two groups:

- a. those that appear exclusively in e , and
- b. those that are contained in another hyperedge f of H .

Example. For query $q_0 : \neg R(v_1, v_2, v_3), S(v_2, v_3), T(v_3, v_5, v_6), G(v_4)$ in the Figure, e_2, e_3, e_4 are ears.



[1] Clement Tak Yu and Meral Z Ozsoyoglu. 1979. An algorithm for tree-query membership of a distributed query. In COMPSAC 79. Proceedings. Computer Software and The IEEE Computer Society's Third International Applications Conference, 1979. IEEE, 306–312.

GYO recognizes α -acyclicity

Apply the two reductions repeatedly

Remove a vertex appearing in only one edge; remove an edge contained in another.

If all edges disappear

The hypergraph is α -acyclic and a join tree can be constructed.

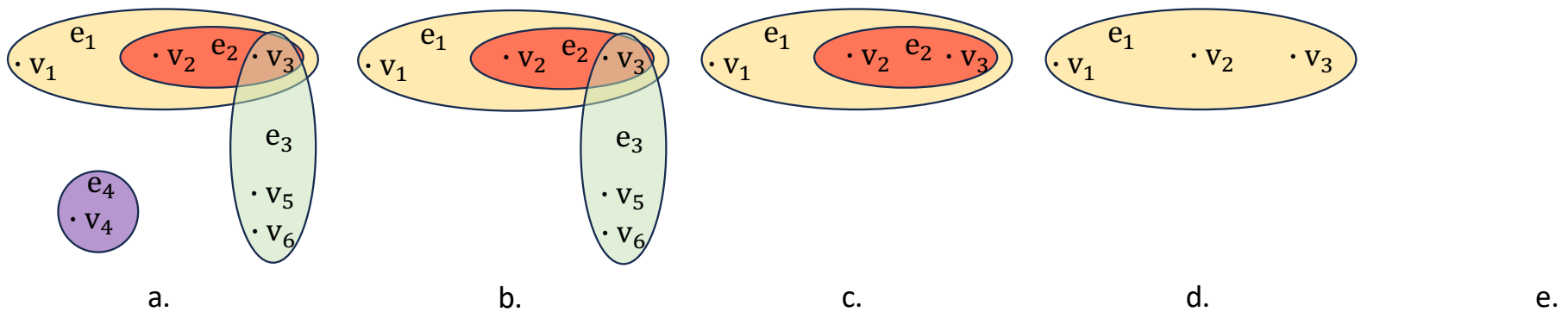
If a nonempty core remains

The hypergraph is cyclic under this notion of acyclicity.

Hypergraph α -acyclicity is the structural notion used throughout this tutorial.

GYO Algorithm (cont.)

Example. For q_0 , the GYO algorithm will run as follows:



Since the resulting hypergraph has no vertices, the algorithm returns that the query is acyclic.

Join Tree

Join Tree

Definition (Join Tree) [1]. A join tree $JT(H)$ for a hypergraph H is a tree whose vertices are the hyperedges of H such that, whenever the same node $X \in V$ occurs in two hyperedges h_1 and h_2 of H , then X occurs in each vertex on the unique path linking h_1 and h_2 in $JT(H)$ (connectedness condition for X).

Lemma (Acyclicity). A hypergraph H is acyclic iff it has a join tree.

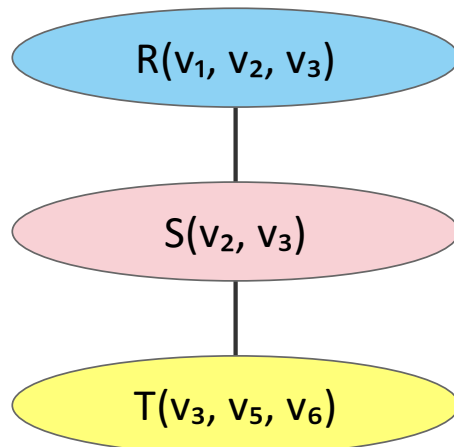
[1] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. 1983. On the desirability of acyclic database schemes. JACM 30, 3 (1983), 479–513.

Join trees: valid and invalid examples

$R(v_1, v_2, v_3)$, $S(v_2, v_3)$, $T(v_3, v_5, v_6)$

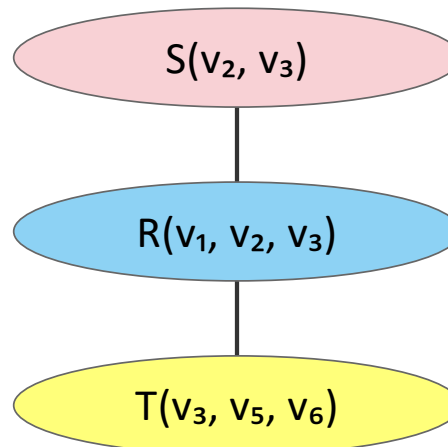
Unrooted trees: edges are undirected; no node is designated as the root.

Valid: S in the middle



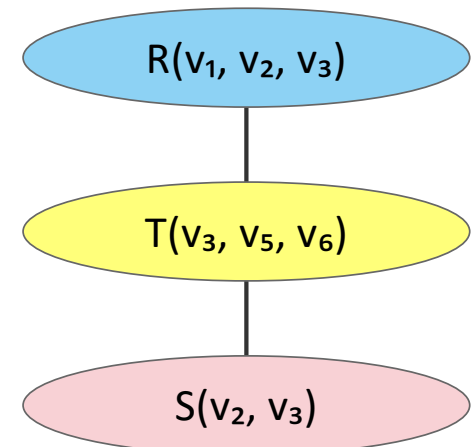
Edges: RS, ST

Valid: R in the middle



Edges: RS, RT

Invalid: T in the middle



Edges: RT, ST

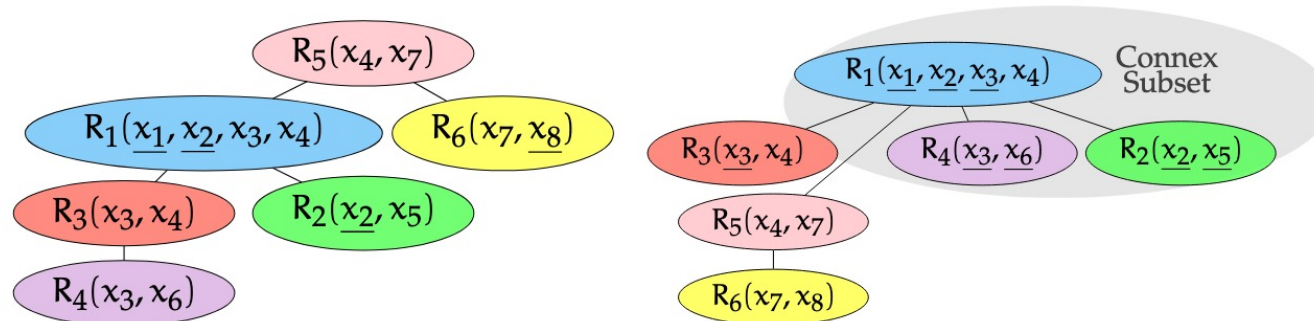
R and S contain v_2 . Their path must stay inside nodes containing v_2 .

In R–T–S, T has no v_2 , so running intersection fails for every root choice.

Free-connex queries

Definition (free-connex queries) [1]. A CQ Q is *free-connex* if and only if it has a join tree T with a subtree T_n containing the root node that satisfies two conditions:

- (1) $\emptyset \subseteq A(T_n)$, where $A(T_n)$ represents the set of all attributes present in T_n
- (2) For any non-root node $R(A) \in T_n$ with parent $R_p(A_p)$, $A \cap A_p \subseteq \emptyset$.



(a) Non-Free-connex query with output attributes x_1, x_2, x_8 (b) Free-connex query with output attributes x_1, x_2, x_3, x_5, x_6

Such a T is called a free-connex join tree of Q , and T_n is referred to as its *connex subset*.

[1] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In Computer Science Logic. Springer Berlin Heidelberg, Berlin, Heidelberg, 208–222.

GHD

Generalized hypertree decomposition

A tree of bags, with edge covers

A GHD is (T, χ, λ) : $\chi(t)$ is a set of attributes; $\lambda(t)$ is a set of relation edges.

Coverage and connectedness

Each relation schema fits in some bag; bags containing any attribute form a connected subtree.

Each bag has a guard set

$\chi(t) \subseteq \bigcup_{e \in \lambda(t)} e$. Its cost depends on the relations needed to cover the bag.

GHD adds a relation-cover measure to a tree decomposition.

Generalized Hypertree Decomposition (GHD)

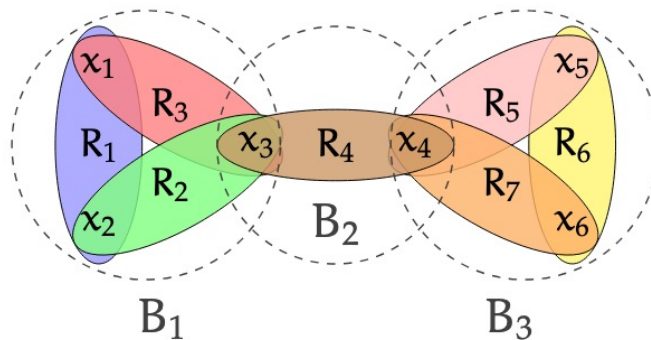
Example See Figure2 below as an example of GHD on a natural join of 7 relations:

$$q_2: - R_1(x_1, x_2), R_2(x_2, x_3), R_3(x_3, x_1), R_4(x_3, x_4), R_5(x_4, x_5), R_6(x_5, x_6), R_7(x_6, x_4)$$

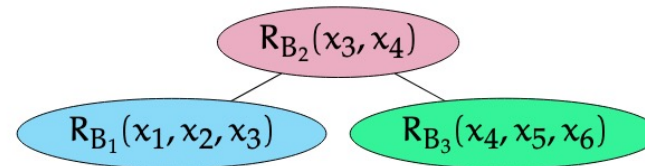
$$R_{B_1} \leftarrow R_1(x_1, x_2) \bowtie R_2(x_2, x_3) \bowtie R_3(x_3, x_1);$$

$$R_{B_2} \leftarrow R_4(x_3, x_4);$$

$$R_{B_3} \leftarrow R_5(x_4, x_5) \bowtie R_6(x_5, x_6) \bowtie R_7(x_6, x_4);$$



(a) Generalized Hypertree Decomposition



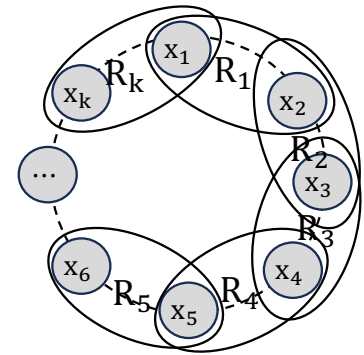
(b) Join tree

Generalized and fractional hypertree width

Measure	Definition
Width of a given GHD	$\max_t \lambda(t) $
Generalized hypertree width $\text{ghw}(Q)$	Minimum width over valid GHDs
Equivalent bag-cost view	Use the minimum integral edge cover $\rho(\chi(t))$ for each bag
Fractional hypertree width	Use fractional edge covers $\rho^*(\chi(t))$ instead
Acyclicity	A nonempty CQ is α -acyclic iff $\text{ghw}(Q) = 1$

For a triangle, $\text{ghw} = 2$ while fractional width = $3/2$.

Generalized Hypertree Width



Example. Consider the Boolean cycle query of length k :

$$C^k() : - R_1(x_1, x_2), R_2(x_2, x_3), \dots, R_k(x_k, x_1)$$

Define $\chi(v_i) = \{x_1, x_i, x_{i+1}\}$, two conditions satisfied. The ghw here is 2, since each bag can be covered by two atoms.

Also not possible to get construct a tree decomposition for C_k with ghw less than 2 then ghw of C_k is 2.

Running Example

Example. Consider the simple 3-way join (possibly with selection predicates):

$$Q = R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(C, D)$$

R ₁		R ₂		R ₃	
A	B	B	C	C	D
1	1	1	3	1	1
2	1	1	4	1	2
...	1	1	...	1	...
N	1	1	N + 1	1	N
2	2	2	2	2	1
3	2	3	1	3	2
		4	1		
		...	1		
		N + 1	1		

Two Versions of the Query

Results for **SELECT *** of Q :

SELECT *			
A	B	C	D
1	1	3	2
2	1	3	2
...	1	3	2
N	1	3	2
2	2	2	1
3	2	2	1

Results for **SUM(A*C) GROUP BY D** of Q :

D	SUM(A*C)
1	10
2	$3N(N+1)/2$

Intermediate result has size $O(N^2)$

- $R_1 \bowtie R_2$
- $R_2 \bowtie R_3$

Yannakakis

Yannakakis

Yannakakis Algorithm.[1] On a given acyclic query Q with a join tree T , works as follows:

1. Traverse the tree in the post-order; for each visited tree node R_i and its parent node R_p , replace R_p with $R_p \bowtie R_i$;
2. Traverse the tree in the pre-order; for each visited non-leaf node R_i , for each $R_c \in C_i$, replace R_c with $R_c \bowtie R_i$;
3. Traverse the tree in the post-order again; for each visited tree node R_i , replace R_p with $\left(\pi_{A_p \cup O}\right) R_i \bowtie R_p$ and remove R_i from the tree;

[1] M. Yannakakis, "Algorithms for acyclic database schemes," in VLDB, vol. 81, 1981, pp. 82–94.

Yannakakis

Plan for SELECT * of Q

$$(1) R_2 \leftarrow R_2 \bowtie R_1;$$

R_1		R_2		R_3	
A	B	B	C	C	D
1	1	1	3	1	1
2	1	1	4	1	2
...	1	1	...	1	...
N	1	1	N + 1	1	N
2	2	2	2	2	1
3	2	~~3~~	~~1~~	3	2
		4	~~1~~		
		...	~~1~~		
		~~N + 1~~	~~1~~		

Yannakakis

Plan for SELECT * of Q

$$(2) R_3 \leftarrow R_3 \bowtie R_2;$$

R ₁		R ₂		R ₃	
A	B	B	C	C	D
1	1	1	3	~~1~~	~~1~~
2	1	1	4	~~1~~	~~2~~
...	1	1	...	~~1~~	~~...~~
N	1	1	N + 1	~~1~~	~~N~~
2	2	2	2	2	1
3	2			3	2

Yannakakis

Plan for SELECT * of Q

$$(3) R_2 \leftarrow R_2 \bowtie R_3;$$

R_1		R_2		R_3	
A	B	B	C	C	D
1	1	1	3	2	1
2	1	~~1~~	~~4~~	3	2
...	1	~~1~~	~~...~~		
N	1	~~1~~	~~N+1~~		
2	2	2	2		
3	2				

Yannakakis

Plan for SELECT * of Q

$$(4) R_1 \leftarrow R_1 \bowtie R_2;$$

R_1		R_2		R_3	
A	B	B	C	C	D
1	1	1	3	2	1
2	1	2	2	3	2
...	1				
N	1				
2	2				
3	2				

Yannakakis

Plan for SELECT * of Q

(5) $J_1 \leftarrow R_1 \bowtie R_2$;

R_1		R_2		R_3	
A	B	B	C	C	D
1	1	1	3	2	1
2	1	2	2	3	2
...	1				
N	1				
2	2				
3	2				

J_1		
A	B	C
1	1	3
2	1	3
...	1	3
N	1	3
2	2	2
3	2	2

Yannakakis

Plan for SELECT * of Q

$$(6) Q \leftarrow J_1 \bowtie R_3 ;$$

R_1		R_2		R_3		J_1		
A	B	B	C	C	D	A	B	C
1	1	1	3	2	1	1	1	3
2	1	2	2	3	2	2	1	3
...	1					...	1	3
N	1					N	1	3
2	2					2	2	2
3	2					3	2	2

Q			
A	B	C	D
1	1	3	2
2	1	3	2
...	1	3	2
N	1	3	2
2	2	2	1
3	2	2	1

Yannakakis⁺

Yannakakis⁺

Yannakakis⁺ Algorithm.[1] On a given acyclic query Q with a join tree T , works as follows:

1. First round: The first round performs a post-order traversal on T with projection/aggregation or semi-join.
2. Second round: Iteratively perform join & projection/aggregation

[1] Q. Wang, B. Chen, B. Dai, K. Yi, F. Li, and L. Lin, "Yannakakis+: Practical acyclic query evaluation with theoretical guarantees," Proc. ACM Manag. Data, vol. 3, no. 3, Jun. 2025

Yannakakis+: first-round choices

Condition at a visited relation R_i	Operation	Effect
R_i is now a leaf, and its output attributes occur in parent R_p	Group/project R_i on $A_i \cap A_p$, then join into R_p	Remove R_i from the tree
Otherwise	Keep output and still-needed join attributes, then $R_p \leftarrow R_p \bowtie R_i$	Reduce the parent and retain R_i

Post-order traversal. Early aggregation removes a leaf when its required outputs survive in the parent.

Yannakakis⁺

First round plan for SELECT * of Q

$$(1) R_2 \leftarrow R_2 \bowtie R_1;$$

R ₁		R ₂		R ₃	
A	B	B	C	C	D
1	1	1	3	1	1
2	1	1	4	1	2
...	1	1	...	1	...
N	1	1	N + 1	1	N
2	2	2	2	2	1
3	2	~~3~~	~~1~~	3	2
		~~4~~	~~1~~		
		~~...~~	~~1~~		
		~~N + 1~~	~~1~~		

Yannakakis⁺

First round plan for SELECT * of Q

$$(2) R_3 \leftarrow R_3 \bowtie R_2;$$

R ₁		R ₂		R ₃	
A	B	B	C	C	D
1	1	1	3	~~1~~	~~1~~
2	1	1	4	~~1~~	~~2~~
...	1	1	...	~~1~~	~~...~~
N	1	1	N + 1	~~1~~	~~N~~
2	2	2	2	2	1
3	2			3	2

Yannakakis⁺

Second round plan for SELECT * of Q

$$(3) J_1 \leftarrow R_2 \bowtie R_3;$$

R ₁		R ₂		R ₃	
A	B	B	C	C	D
1	1	1	3	2	1
2	1	1	4	3	2
...	1	1	...		
N	1	1	N + 1		
2	2	2	2		
3	2				

J ₁		
B	C	D
1	3	2
2	2	1

R_3 has no dangling tuples $\Rightarrow$

J_1 has no dangling tuples $\Rightarrow$

$$|J_1| \leq M$$

Yannakakis⁺

Second round plan for SELECT * of Q

(4) $Q \leftarrow J_1 \bowtie R_1$;

R_1		J_1		
A	B	B	C	D
1	1	1	3	2
2	1	2	2	1
...	1			
N	1			
2	2			
3	2			

Q			
A	B	C	D
1	1	3	2
2	1	3	2
...	1	3	2
N	1	3	2
2	2	2	1
3	2	2	1

J_1 is dangling-free
 $\Rightarrow Q$ is dangling-free
 $\Rightarrow |Q| \leq M$

Example. $\text{SUM}(A * C) \text{ GROUP BY } D \text{ FROM } R_1 \bowtie R_2 \bowtie R_3$

R_1			R_2			R_3		
A	B	Annot	B	C	Annot	C	D	Annot
1	1	1	1	3	3	1	1	1
2	1	2	1	4	4	1	2	1
...	1	...	1	...	...	1	...	1
N	1	N	1	N + 1	N + 1	1	N	1
2	2	2	2	2	2	2	1	1
3	2	3	3	1	1	3	2	1
			4	1	1			
			...	1	1			
			N + 1	1	1			

$$(1) R_1 \leftarrow \pi_B R_1;$$

R ₁			R ₂			R ₃		
A	B	Annot	B	C	Annot	C	D	Annot
1	1	1	1	3	3	1	1	1
2	1	2	1	4	4	1	2	1
...	1	...	1	...	...	1	...	1
N	1	N	1	N + 1	N + 1	1	N	1
2	2	2	2	2	2	2	1	1
3	2	3	3	1	1	3	2	1
			4	1	1			
			...	1	1			
			N + 1	1	1			

R' ₁	
B	Annot
1	$N(N+1) / 2$
2	5

Yannakakis⁺

Join + Aggregation Queries

$$(2) R_2 \leftarrow R_2 \bowtie R_1';$$

R ₁		R ₁ '		R ₂			R ₃	
A	B	B	annot	B	C	annot	C	D
1	1	1	$N(N+1)/2$	1	3	3	1	1
2	1	2	5	1	4	4	1	2
...	1			1	...	...	1	...
N	1			1	N+1	N+1	1	N
2	2			2	2	2	2	1
3	2			3	1	1	3	2
				4	1	1		
				...	1	1		
				N+1	1	1		

R ₂		
B	C	annot
1	3	$3N(N+1)/2$
1	4	$4N(N+1)/2$
1	...	...
1	N+1	$N(N+1)^2/2$
2	2	10

Yannakakis⁺

Join + Aggregation Queries

$$(3) R_2' \leftarrow \pi_C R_2;$$

R ₁		R ₁ '		R ₂			R ₃	
A	B	B	annot	B	C	annot	C	D
1	1	1	$N(N+1)/2$	1	3	$3N(N+1)/2$	1	1
2	1	2	5	1	4	$4N(N+1)/2$	1	2
...	1			1	...	...	1	...
N	1			1	N + 1	$N(N+1)^2/2$	1	N
2	2			2	2	10	2	1
3	2						3	2

R ₂ '	
C	annot
3	$3N(N+1)/2$
4	$4N(N+1)/2$
...	...
N + 1	$N(N+1)^2/2$
2	10

Yannakakis⁺

Join + Aggregation Queries

$$(4) R_3 \leftarrow R_3 \bowtie R_2';$$

R ₁		R ₁ '		R ₂			R ₂ '		R ₃	
A	B	B	annot	B	C	annot	C	annot	C	D
1	1	1	$N(N+1)/2$	1	3	$3N(N+1)/2$	3	$3N(N+1)/2$	1	1
2	1	2	5	1	4	$4N(N+1)/2$	4	$4N(N+1)/2$	1	2
...	1			1	...	...	...	...	1	...
N	1			1	N + 1	$N(N+1)^2/2$	N + 1	$N(N+1)^2/2$	1	N
2	2			2	2	10	2	10	2	1
3	2								3	2

R ₃		
C	D	Annot
2	1	10
3	2	$3N(N+1)/2$

Yannakakis⁺

Join + Aggregation Queries

$$(5) Q \leftarrow \pi_D R_3$$

R ₁		R ₁ '		R ₂			R ₂ '		R ₃		
A	B	B	annot	B	C	annot	C	annot	C	D	annot
1	1	1	$N(N+1)/2$	1	3	$3N(N+1)/2$	3	$3N(N+1)/2$	2	1	10
2	1	2	5	1	4	$4N(N+1)/2$	4	$4N(N+1)/2$	3	2	$3N(N+1)/2$
...	1			1	...	...	...	...			
N	1			1	N + 1	$N(N+1)^2/2$	N + 1	$N(N+1)^2/2$			
2	2			2	2	10	2	10			
3	2										

Yannakakis⁺ second-round invariant

A supported starting point

The root after the first round is dangling-free.

A safe merge: a reducible neighbor

For dangling-free R_i , choose adjacent R_j such that $A_i \cap A_k \subseteq O$ for every other neighbor R_k of R_i .

When no such neighbor is available

Use a semijoin to make a child dangling-free, and continue toward a leaf.

For free-connex queries the reduced query is full, so every adjacent relation is reducible.

Yannakakis⁺ theoretical guarantees

Query class	After the first round	Total time
Relation-dominated: $O \subseteq A_r$	One relation, using a dominating root	$O(N)$
Free-connex	A full query, using a free-connex tree	$O(N + M)$
General acyclic	Outputs and necessary join attributes remain	$O(N + \min(NM, F))$

N: input. M: output. F: full join. Fixed acyclic query, exact operators, expected hashing costs.

Optimizer

Cardinality Estimation

Definition. Cardinality estimation (CE) predicts the number of tuples produced by each operator in a query plan.

Assumptions. Common simplifying assumptions in basic estimation models include:

- a. Uniformity: values in a column are uniformly distributed within the global min/max.
- b. Independence: values in different columns are uncorrelated.
- c. Inclusion: probe-side join-key values are assumed to occur on the build side.

Plan Enumeration

Definition. Plan enumeration searches equivalent join orders and ranks them using a cost model. General join-order optimization is NP-hard under common formulations.

Algorithms.

- Dynamic programming.[1,2,3]
- GYO[4] & GHD[5]

- [1] Guido Moerkotte and Thomas Neumann. 2006. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In Proceedings of the 32nd International Conference on Very Large Data Bases (Seoul, Korea) (VLDB '06). VLDB Endowment, 930–941.
- [2] Guido Moerkotte and Thomas Neumann. 2008. Dynamic programming strikes back. In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (Vancouver, Canada) (SIGMOD '08). Association for Computing Machinery, New York, NY, USA, 539–552.
- [3] M. Stoian and A. Kipf, “Dpconv: Super-polynomially faster join ordering,” Proc. ACM Manag. Data, vol. 2, no. 6, Dec. 2024.
- [4] Clement Tak Yu and Meral Z Ozsoyoglu. 1979. An algorithm for tree-query membership of a distributed query. In COMPSAC 79. Proceedings. Computer Software and The IEEE Computer Society's Third International Applications Conference, 1979. IEEE, 306–312.
- [5] G. Gottlob, N. Leone, and F. Scarcello, “Hypertree decompositions and tractable queries,” in Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, ser. PODS '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 21–32

Cost Model

Definition. Cost Model (CM) translates the database state (which relations are in memory, availability of indexes, etc.) and cardinality estimates into execution costs, with traditional models defined by experts and modern, adaptive learning-based methods.

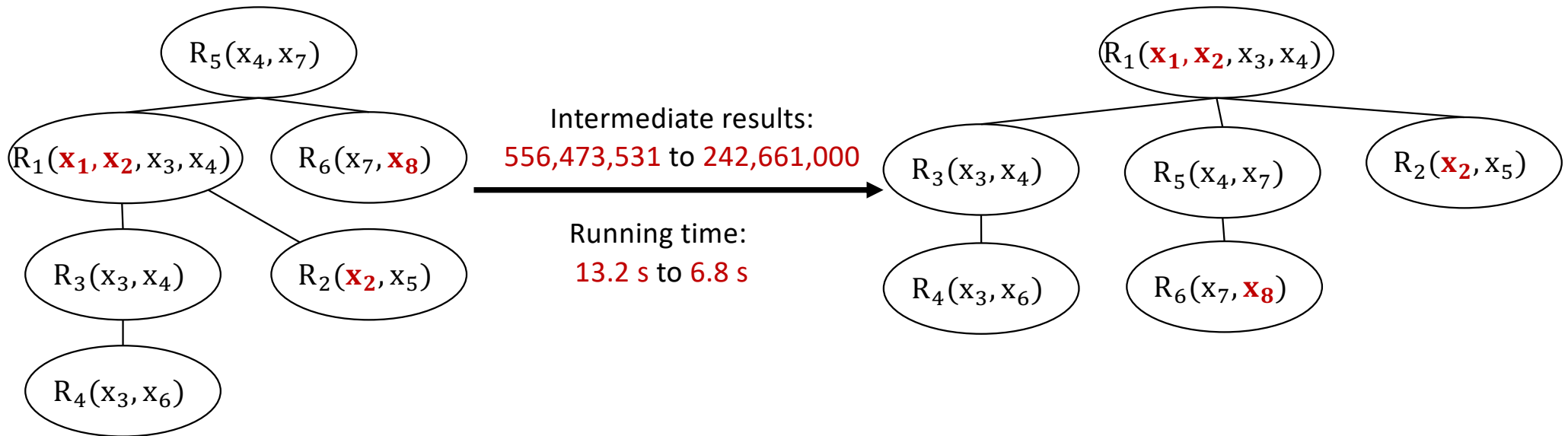
Example. Join cost model in DuckDB.

```
double CostModel::ComputeCost(DPJoinNode &left, DPJoinNode &right) {  
    auto &combination = query_graph_manager.set_manager.Union(left.set, right.set);  
    auto join_card = cardinality_estimator.EstimateCardinalityWithSet<double>(combination);  
    auto join_cost = join_card;  
    return join_cost + left.cost + right.cost;  
}
```

Cost-Based Optimization: Example

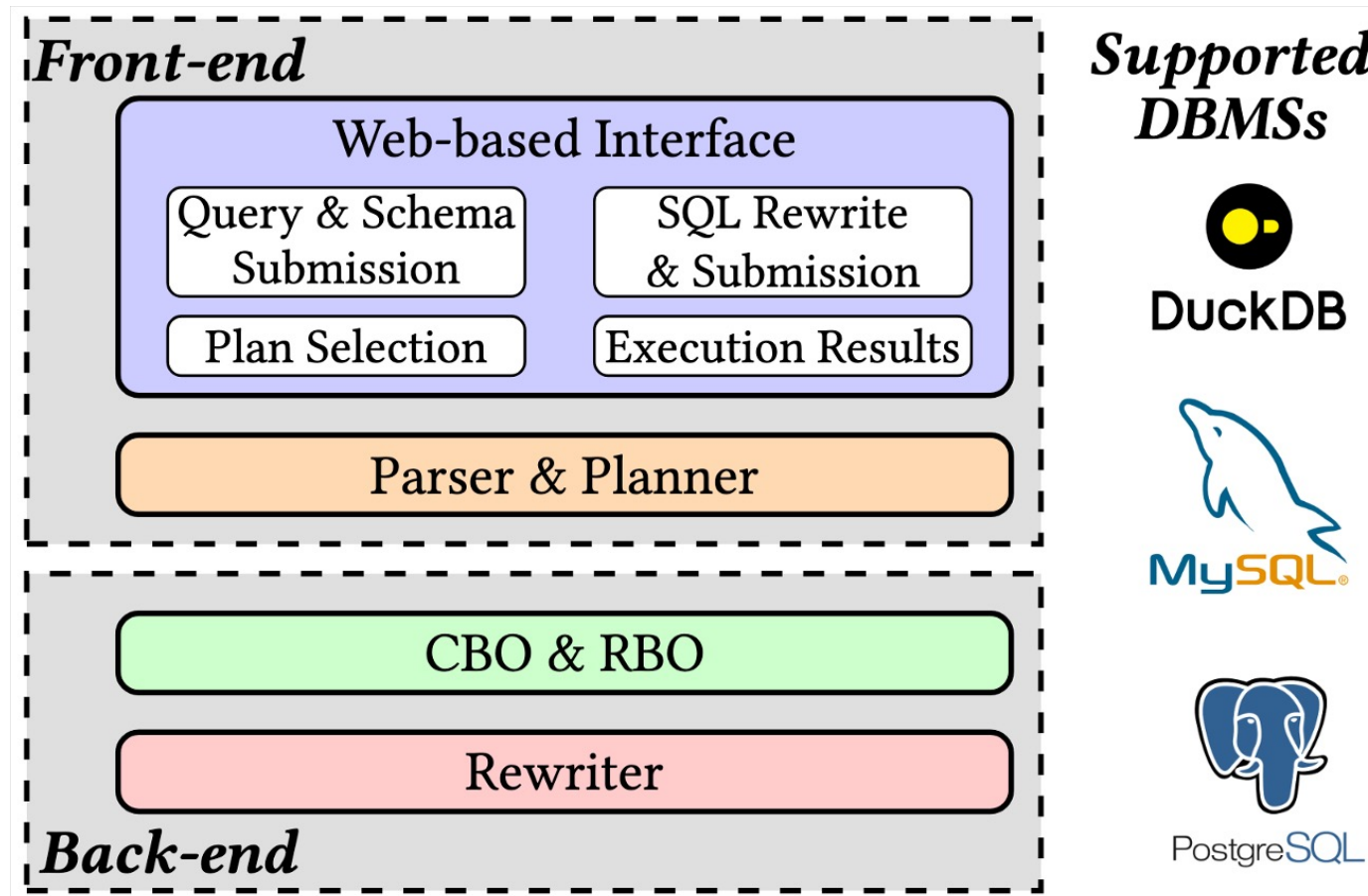
Example. TPC-H query 9:

$$Q_9 = \pi_{x_1, x_2, x_8} (R_1(x_1, x_2, x_3, x_4) \bowtie R_2(x_2, x_5) \bowtie R_3(x_3, x_4) \bowtie R_4(x_3, x_6) \bowtie R_5(x_4, x_7) \bowtie R_6(x_7, x_8))$$



For a fixed query, valid plans share the stated worst-case bound; costs still differ in practice.

Our rewriter system



Dumb-bell query: original SQL

```
SELECT count(*)  
FROM Graph AS g1, Graph AS g2, Graph AS g3,  
      Graph AS g4, Graph AS g5, Graph AS g6, Graph AS g7  
WHERE g1.dst = g2.src AND g2.dst = g3.src AND g3.dst = g1.src  
      AND g4.dst = g5.src AND g5.dst = g6.src AND g6.dst = g4.src  
      AND g1.dst = g7.src AND g7.dst = g4.src
```

Two directed triangles connected by g7. COUNT(*) counts all matching combinations.

Rewritten SQL: right triangle and bridge

```
create or replace TEMP view bag187 as
select g5.dst as v10, g4.src as v12, g4.dst as v8
from Graph as g4, Graph as g5, Graph as g6
where g4.dst=g5.src and g5.dst=g6.src and g6.dst=g4.src;

create or replace TEMP view aggView2342847404648505626 as
select v12, COUNT(*) as annot from bag187 group by v12;

create or replace TEMP view aggJoin8909182364670701114 as
select src as v2, annot
from Graph as g7, aggView2342847404648505626
where g7.dst=aggView2342847404648505626.v12;
```

Count right-triangle matches by v12, then carry each count across g7 to v2.

Rewritten SQL: left triangle

```
create or replace TEMP view bag188 as
select g1.dst as v2, g2.dst as v4, g1.src as v6
from Graph as g1, Graph as g2, Graph as g3
where g1.dst=g2.src and g2.dst=g3.src and g3.dst=g1.src;

create or replace TEMP view aggView6906784260504084903 as
select v2, COUNT(*) as annot from bag188 group by v2;
```

Count left-triangle matches by v2. The final join can now combine counts.

Rewritten SQL: multiply and sum

```
create or replace TEMP view aggJoin1657488435551201938 as
select aggJoin8909182364670701114.annot
      * aggView6906784260504084903.annot as annot
from aggJoin8909182364670701114
join aggView6906784260504084903 using(v2);

select SUM(annot) as v15 from aggJoin1657488435551201938;
```

COUNT(*) semantics on empty input require COALESCE(SUM(annot), 0).

Yannakakis⁺ in DuckDB

DuckDB: fast OLAP



github.com/duckdb/duckdb

41.9k

GitHub stars

Snapshot: 6 October 2026

Fast analytics on one machine

Columnar vectors and multicore execution accelerate large scans, joins, and aggregates.

SQL on the files you already have

Query Parquet, CSV, JSON, and data frames directly from your analytical workflow.

No database server to manage

Run SQL inside Python, R, a notebook, the CLI, or your own application.

Open, portable, extensible

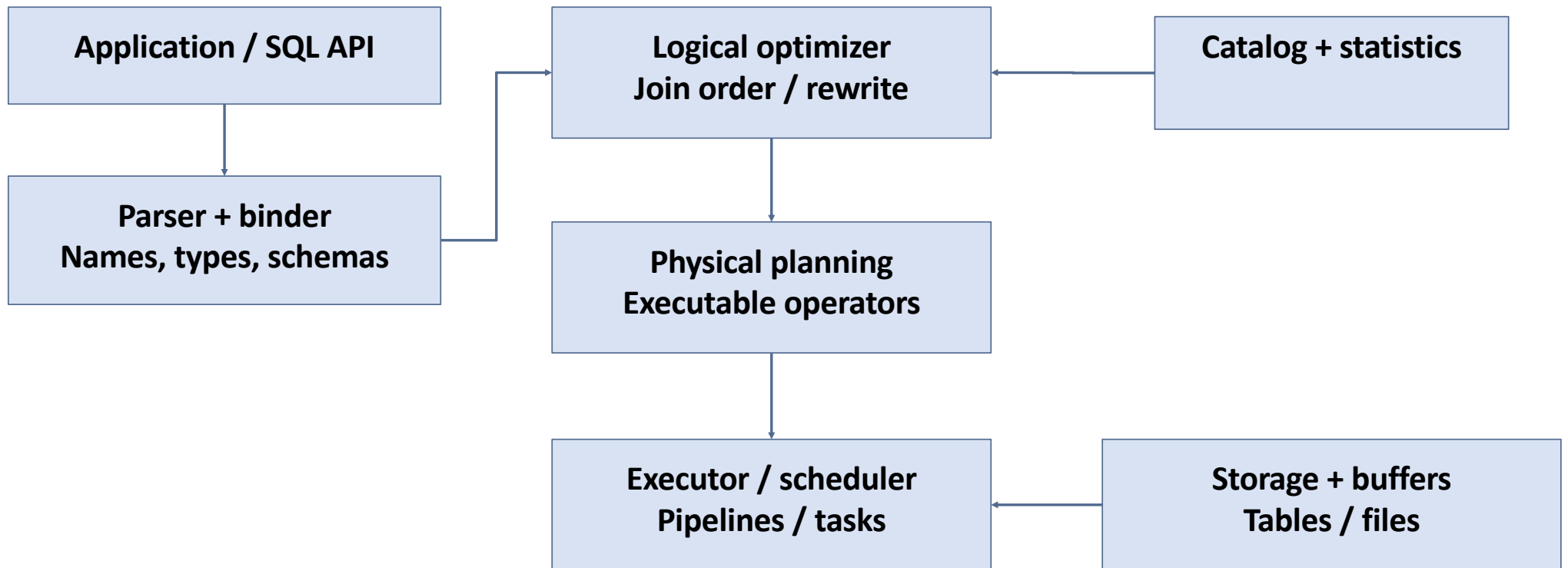
MIT license, portable database files, and an ecosystem of extensions.

DuckDB: execution model

Scale	Engine mechanism	Why it matters
Worker task / morsel	Workers obtain pieces of source work	Balances work across CPU threads
Pipeline	Source → operators → sink	Streams data until a blocking boundary
DataChunk / vector	Column vectors; normally $\leq 2,048$ rows	Amortizes per-row operator overhead
CPU instruction	SIMD processes several lanes	May accelerate kernels within a batch

Morsel-driven scheduling + vectorized pipelines provide the execution framework for our filters.

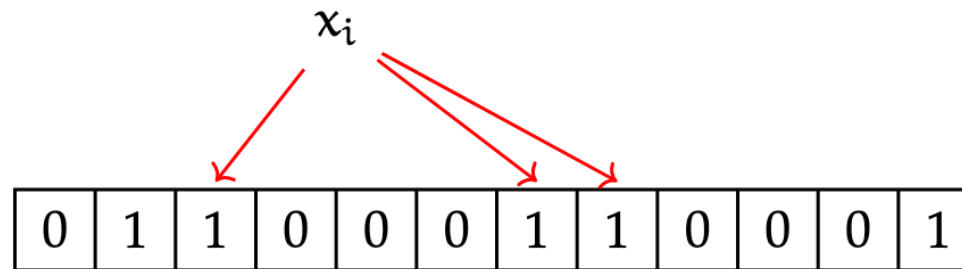
DuckDB system architecture



SQL planning prepares the operators. The executor runs vectorized pipelines over shared storage services.

Bloom Filter

Definition (Bloom filter) [1]. A sketch for testing set membership. It utilizes a bit array B of length m and k independent hash functions $h_1, h_2, \dots, h_k$, each mapping elements to $\{1, 2, \dots, m\}$. Upon receiving an element x_i , it sets $B[h_j(x_i)] = 1$ for all $j \in [k]$. To check if an element x belongs to the set, it verifies whether $B[h_j(x)] = 1$ for all $j \in [k]$, returning true if satisfied, or false otherwise.



Example of a Bloom Filter with $k = 3$ and $m = 12$

Predicate Transfer. [2,3] Inspired by the Yannakakis algorithm, it proposes to speed up the above semi-joins by building bloom filters mentioned above instead of full hash filters.

[1] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," Communications of the ACM, vol. 13, no. 7, pp. 422–426, 1970.

[2] Y. Yang, H. Zhao, X. Yu, and P. Koutris, "Predicate transfer: Efficient pre-filtering on multi-join queries," in 14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024.

[3] J. Zhao, K. Su, Y. Yang, X. Yu, P. Koutris, and H. Zhang, "Debunking the myth of join ordering: Toward robust sql analytics," Proc. ACM Manag. Data, vol. 3, no. 3, Jun. 2025.

False positives and useful filtering

Classical Bloom-filter model	Expression
m bits, n inserted keys, k hash functions	$p \approx (1 - e^{(-kn/m)})^k$
Best k in this idealized model	$k \approx (m/n) \ln 2$
Probe survival with true-match fraction s	$q = s + (1 - s)p$
Example: s = 0.10 and p = 0.01	q = 0.109, so about 89.1% are rejected

Illustrative calculation. Blocked layouts and skew can change the actual false-positive rate.

Bloom filter or exact hash membership?

Design choice	Blocked Bloom filter	Exact hash set
Stored information	Bit masks	Keys, or equivalent exact representation
Membership answer	Definitely absent / possibly present	Present / absent
Memory access	Few nearby words per probe	Depends on table layout and occupancy
Parallel insertion	Atomic bit updates in this fork	Implementation-dependent coordination
SIMD opportunities	Mask construction and batched tests	Also possible in modern hash tables

Measure total avoided work, including false positives and build overhead.

Our choice: a blocked Bloom filter

One aligned 64-byte block

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		3 bits									4 bits				

Example selected sectors (positions vary by hash)

A 64-byte aligned block

16 sectors, each a 32-bit word

Two selected sectors per key

Three candidate bits in the first, four in the second

Nearby memory accesses

$\text{sector}_2 = \text{sector}_1 \text{ XOR } (8 + (\text{key_hi} \& 7))$

The XOR changes only low address bits, keeping both sectors inside the same 64-byte block.

Concrete filter settings in the code

Setting	Value / behavior
Storage layout	64-byte aligned blocks; 16 × 32-bit sectors
Sizing target	16 bits per collected build row; round to a power of two
Minimum / cap	512 bits minimum; 2 ²⁶ sectors cap (256 MiB bit array)
Mask work per hash	3 candidate bits in one sector, 4 in another
Processing batch	SIMD_BATCH_SIZE = 16 hashes

Pinned source: DuckDBYanPlus 38d165e. These are implementation constants, not benchmark settings.

Bloom filters: operator-to-kernel map

Build operator	PhysicalCreateBF::Sink → Finalize Buffers input, initializes the filter, schedules build tasks
Build task	CreateBFFinalizeTask::ExecuteTask FetchChunk → BloomFilter::Insert → HashColumns → BloomFilterInsert
Build kernel	BloomFilter::BloomFilterInsert 16-key batches use atomic OR; InsertOne handles the tail
Probe operator	PhysicalUseBF::ExecuteInternal BloomFilterUsage::Lookup → BloomFilter::Lookup → HashColumns
Probe kernel	BloomFilter::BloomFilterLookup 16-key tests; LookupOne handles the tail; flags become a selection

The build event completes before dependent probes run. These are DuckDBYanPlus fork functions.

Build: scheduled tasks consume chunks

PhysicalCreateBF::Sink buffers rows. Finalize initializes filters and schedules these tasks.

CreateBFFinalizeTask::ExecuteTask — complete function

```
TaskExecutionResult ExecuteTask(TaskExecutionMode mode) override {
    DataChunk chunk;
    sink.data_collection->InitializeScanChunk(chunk);
    for (idx_t i = chunk_idx_from; i < chunk_idx_to; i++) {
        sink.data_collection->FetchChunk(i, chunk);
        for (auto &pair : sink.op.unique_bloom_filters) {
            auto &cols_build = pair.first;
            auto &bf = pair.second;
            bf->Insert(chunk, cols_build);
        }
    }
    event->FinishTask();
    return TaskExecutionResult::TASK_FINISHED;
}
```

Each task inserts its assigned chunks. CreateBFFinalizeEvent::FinishEvent marks filters finalized.

Build: public entry and key hashing

BloomFilter::Insert — complete public method

```
void BloomFilter::Insert(DataChunk &chunk, const vector<idx_t> &bound_cols_built) {
    int count = static_cast<int>(chunk.size());
    Vector hashes = HashColumns(chunk, bound_cols_built);
    BloomFilterInsert(count, reinterpret_cast<uint64_t *>(hashes.GetData()), blocks);
}
```

HashColumns — complete file-local helper (anonymous namespace)

```
static Vector HashColumns(DataChunk &chunk, const vector<idx_t> &cols) {
    auto count = chunk.size();
    Vector hashes(LogicalType::HASH);
    VectorOperations::Hash(chunk.data[cols[0]], hashes, count);
    for (size_t j = 1; j < cols.size(); j++) {
        VectorOperations::CombineHash(hashes, chunk.data[cols[j]], count);
    }
    if (hashes.GetVectorType() == VectorType::CONSTANT_VECTOR) {
        hashes.Flatten(count);
    }
    return hashes;
}
```

The probe entry, `BloomFilter::Lookup`, uses this same `HashColumns` helper on its corresponding key columns.

Blocked layout: exact address functions

BloomFilter::GetSector1 / GetSector2 — complete private methods

```
inline uint32_t GetSector1(uint32_t key_lo, uint32_t key_hi) const {  
    // block: 13 bits in key_lo and 9 bits in key_hi  
    // sector 1: 4 bits in key_lo  
    return ((key_lo & ((1 << 17) - 1)) +  
            ((key_hi << 14) & (((1 << 9) - 1) << 17))) & (num_sectors - 1);  
}  
inline uint32_t GetSector2(uint32_t key_hi, uint32_t block1) const {  
    // sector 2: 3 bits in key_hi  
    return block1 ^ (8 + (key_hi & 7));  
}
```

The XOR changes only the lowest four sector bits: both 32-bit sectors stay inside one 64-byte block.

Blocked layout: exact bit-mask functions

Each 64-bit key supplies two 32-bit masks: three selected bit positions in one sector, four in the other.

BloomFilter::GetMask1 / GetMask2 — complete private methods

```
inline uint32_t GetMask1(uint32_t key_lo) const {
    // 3 bits in key_lo
    return (1u << ((key_lo >> 17) & 31)) |
           (1u << ((key_lo >> 22) & 31)) | (1u << ((key_lo >> 27) & 31));
}
inline uint32_t GetMask2(uint32_t key_hi) const {
    // 4 bits in key_hi
    return (1u << ((key_hi >> 12) & 31)) | (1u << ((key_hi >> 17) & 31)) |
           (1u << ((key_hi >> 22) & 31)) |
           (1u << ((key_hi >> 27) & 31));
}
```

Repeated bit positions can coincide. Insert ORs these masks into the filter; probe checks that every bit is set.

Build kernel: atomic bit insertion

Called by BloomFilterInsert for the final 0–15 keys. Full batches perform the same two atomic OR updates.

BloomFilter::InsertOne — complete private method

```
inline void InsertOne(uint32_t key_lo, uint32_t key_hi, uint32_t *BF_RESTRICT bf) const {
    uint32_t sector1 = GetSector1(key_lo, key_hi);
    uint32_t mask1 = GetMask1(key_lo);
    uint32_t sector2 = GetSector2(key_hi, sector1);
    uint32_t mask2 = GetMask2(key_hi);
    // Perform atomic OR operation on the bf array elements using std::atomic
    std::atomic<uint32_t>& atomic_bf1 =
        *reinterpret_cast<std::atomic<uint32_t>*>(&bf[sector1]);
    std::atomic<uint32_t>& atomic_bf2 =
        *reinterpret_cast<std::atomic<uint32_t>*>(&bf[sector2]);
    atomic_bf1.fetch_or(mask1, std::memory_order_relaxed);
    atomic_bf2.fetch_or(mask2, std::memory_order_relaxed);
}
```

Atomic OR combines concurrent insertions without losing bits. Event completion separates build and probe.

Probe: public entry and membership test

BloomFilter::Lookup — complete public method

```
int BloomFilter::Lookup(DataChunk &chunk, vector<uint32_t> &results,
                        const vector<idx_t> &bound_cols_applied) const {
    int count = static_cast<int>(chunk.size());
    Vector hashes = HashColumns(chunk, bound_cols_applied);
    BloomFilterLookup(count, reinterpret_cast<uint64_t *>(hashes.GetData()),
                      blocks, results.data());
    return count;
}
```

BloomFilter::LookupOne — complete private method

```
inline bool LookupOne(uint32_t key_lo, uint32_t key_hi,
                      const uint32_t *BF_RESTRICT bf) const {
    uint32_t sector1 = GetSector1(key_lo, key_hi);
    uint32_t mask1 = GetMask1(key_lo);
    uint32_t sector2 = GetSector2(key_hi, sector1);
    uint32_t mask2 = GetMask2(key_hi);
    return ((bf[sector1] & mask1) == mask1) & ((bf[sector2] & mask2) == mask2);
}
```

Lookup returns the input count and fills 0/1 flags. LookupOne handles the tail; both masks must match.

Probe operator: flags select input rows

bf_to_use is a BloomFilterUsage. Its Lookup forwards to BloomFilter::Lookup.

PhysicalUseBF::ExecuteInternal — excerpt, lines 83–98

```
// 1. Lookup the BloomFilter
bf_to_use->Lookup(input, state.lookup_results);
// 2. Fill results
idx_t result_count = 0;
auto &sel = state.sel_vector;
for (size_t i = 0; i < input.size(); i++) {
    sel.set_index(result_count, i);
    result_count += state.lookup_results[i];
}
if (result_count == input.size()) {
    // nothing was filtered: skip adding any selection vectors
    chunk.Reference(input);
} else {
    chunk.Slice(input, sel, result_count);
}
```

Reference reuses the whole input. Slice retains selected rows. Bypass and statistics code are omitted.

Probe kernel: 16-key software batches

BloomFilter::BloomFilterLookup — batch-loop excerpt, lines 106–118

```
for (int j = 0; j < SIMD_BATCH_SIZE; j++) {
    int p = i + j;
    uint32_t key_lo = key[p + p];
    uint32_t key_hi = key[p + p + 1];
    block1[j] = GetSector1(key_lo, key_hi);
    mask1[j] = GetMask1(key_lo);
    block2[j] = GetSector2(key_hi, block1[j]);
    mask2[j] = GetMask2(key_hi);
}
for (int j = 0; j < SIMD_BATCH_SIZE; j++) {
    out[i + j] = ((bf[block1[j]] & mask1[j]) == mask1[j]) &
                ((bf[block2[j]] & mask2[j]) == mask2[j]);
}
```

Same function — scalar-tail excerpt, lines 122–124

```
for (int i = num & ~(SIMD_BATCH_SIZE - 1); i < num; i++) {
    out[i] = LookupOne(key[i + i], key[i + i + 1], bf);
}
```

`SIMD_BATCH_SIZE = 16` is a software batch size. This source contains no explicit AVX gather/test intrinsic.

A concrete two-sector membership example

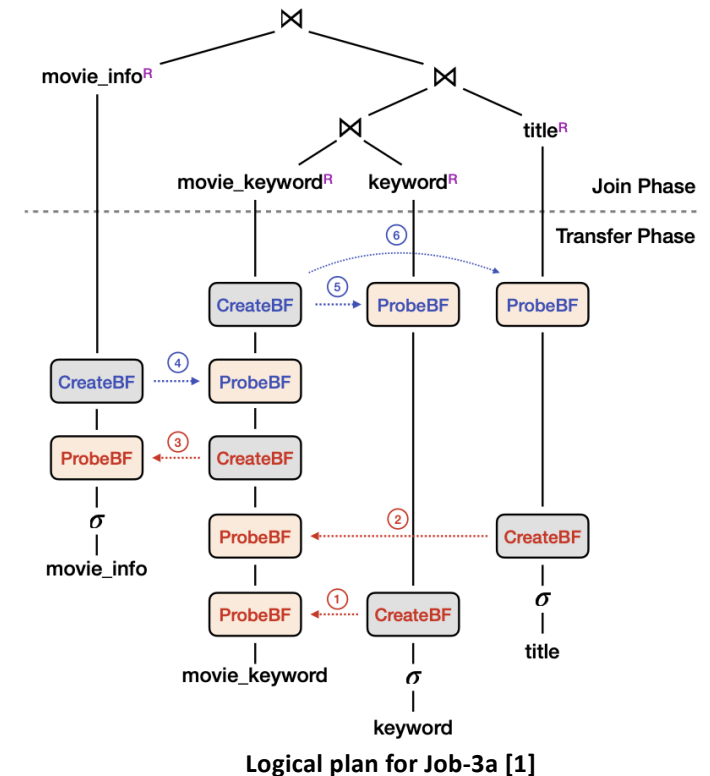
Step	Value in this example
One 64-byte block (16 sectors)	$\text{sector}_1 = 2; \text{key_hi} \& 7 = 1$
Second sector	$\text{sector}_2 = 2 \text{ XOR } (8 + 1) = 11$
Mask in sector 2	Bits {1,6,12}: 0x00001042
Mask in sector 11	Bits {3,7,13,25}: 0x02002088
Build / probe	OR both masks to insert; require both masks to probe

Cache locality comes from sector_2 staying in the same block; collisions explain false positives.

Accelerating Yannakakis with Bloom Filters

Example. JOB-3a Query Plan, using bloom filters to execute Yannakakis algorithm.

```
SELECT MIN(t.title) AS movie_title
FROM keyword AS k, movie_info AS mi, movie_keyword AS mk,
title AS t
WHERE k.keyword LIKE '%sequel%'
AND mi.info IN ('Sweden','Norway','Germany','Denmark',
'Swedish','Denish','Norwegian','German')
AND t.production_year > 2005
AND t.id = mi.movie_id
AND t.id = mk.movie_id
AND mk.movie_id = mi.movie_id
AND k.id = mk.keyword_id;
```



[1] J. Zhao, K. Su, Y. Yang, X. Yu, P. Koutris, and H. Zhang, "Debunking the myth of join ordering: Toward robust sql analytics," Proc. ACM Manag. Data, vol. 3, no. 3, Jun. 2025.

Running time: summary comparison

Exact algorithms on acyclic queries: $O(N + \min(NM, F))$.

Free-connex queries: $O(N + M)$.

$$J = R_1(A,B) \bowtie R_2(B,C) \bowtie R_3(C,D)$$

$$Q_1 = J \quad (M = F) \qquad Q_2 = \pi_{(A,B)} J \quad (M \leq F)$$

	Native binary plan	Yannakakis	PT	RPT	Yannakakis+
Q₁: full join	$O(N^2)$ possible	$O(N + M)$	Filter-dependent*	Filter-dependent*	$O(N + M)$
Q₂: projection	$O(N^2)$ possible	$O(N + M)$	May emit F^*	May emit F^*	$O(N + M)$
Rounds	One plan	3	Plan-specific	Plan-specific	1 or 2
Reduction invariant	None required	Full reduction	False positives	False positives	Supported merges
Early aggregation	Plan-dependent	Yes	Separate rewrite	Separate rewrite	Yes

N: input size. M: final output size. F: full join size. Fixed query, expected hashing costs.

*Bloom prefilters preserve exact answers with exact joins, but do not by themselves establish the exact bound.

Yannakakis+ in DuckDB: reported results

Intel Xeon Gold 6354 CPU @ 3.00GHz (36 cores, 72 threads), 1TB RAM, running Ubuntu 20.04.

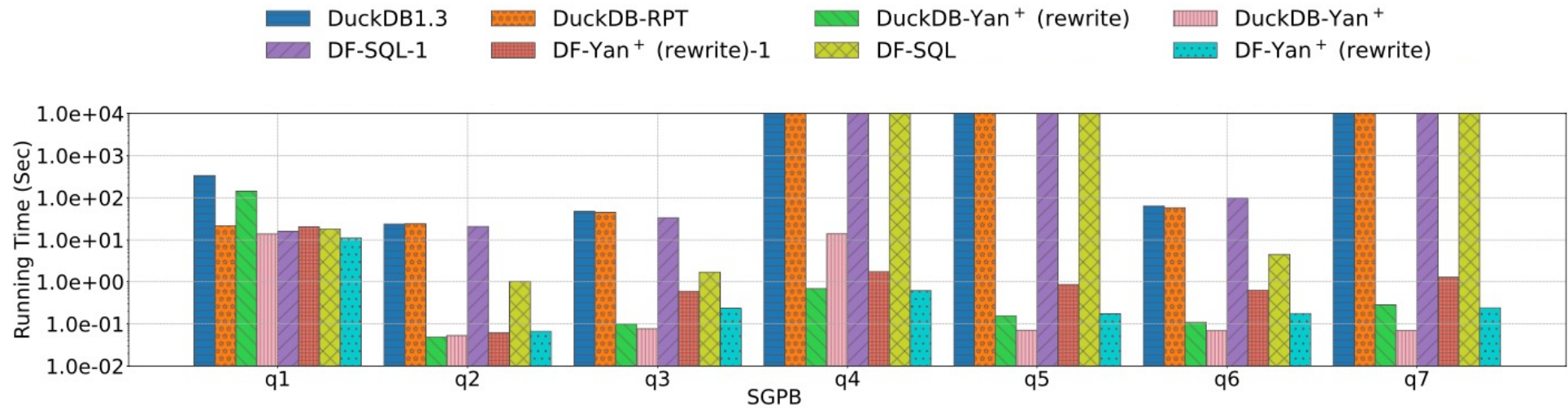
DuckDB v1.3 & v1.5, Apache DataFusion v0.34

Total **136/145** queries.

Yannakakis+ vs. Native query plans: Achieve an average speedup of **1.95x** and a maximum speedup of **104,347.83x**.

Yannakakis+ vs. Yannakakis: Achieve an average speedup of **2.2x**.

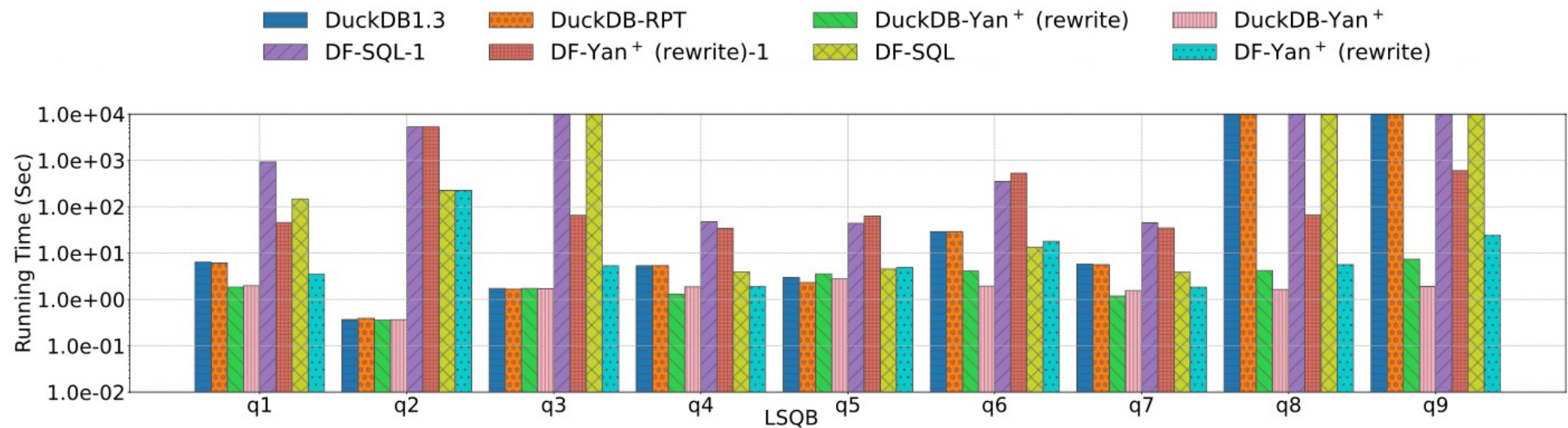
Experiments: SGPB



Running time (seconds, log scale) · Lower is better · Source legend: DuckDB1.3

Large gaps appear on q4, q5, and q7; the log scale spans several orders of magnitude.

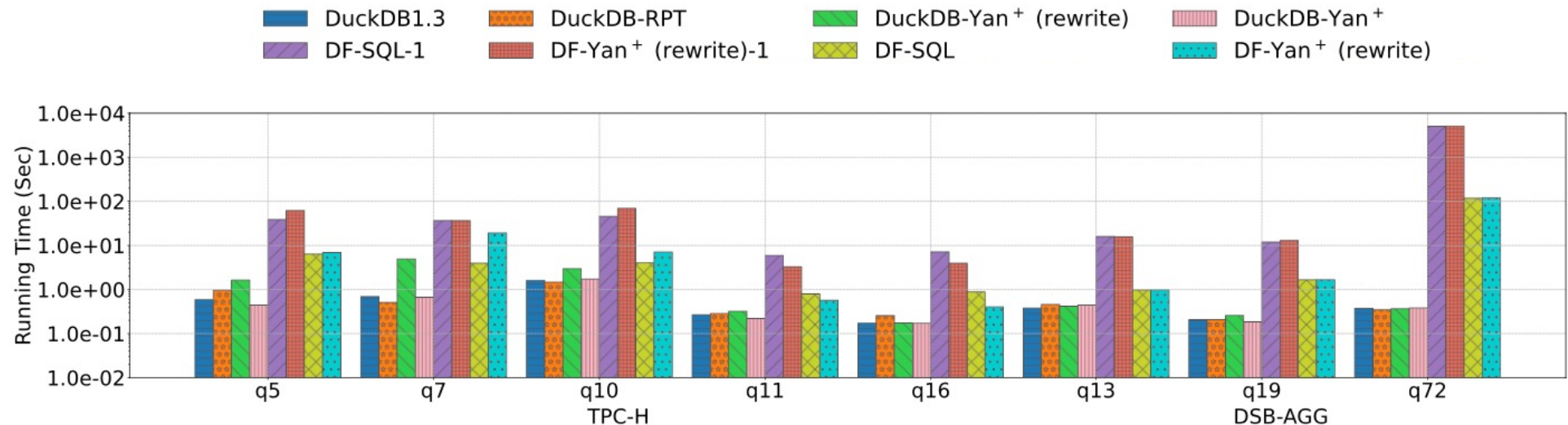
Experiments: LSQB



Running time (seconds, log scale) · Lower is better · Source legend: DuckDB1.3

The benefit varies by query: compare near-ties with the larger gaps on q8 and q9.

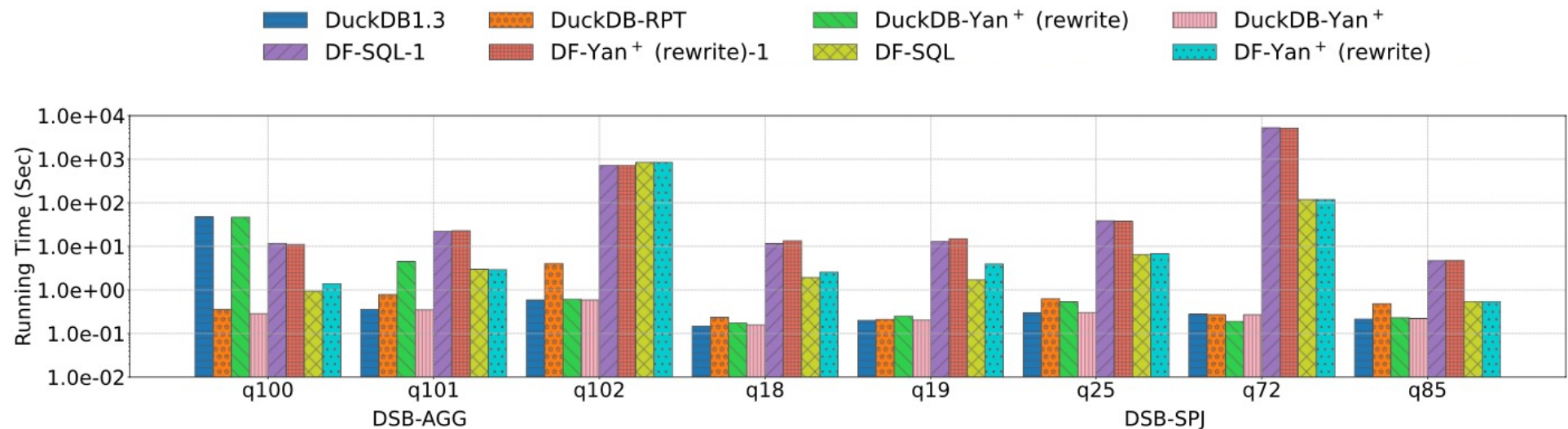
Experiments: TPC-H and DSB-AGG



Running time (seconds, log scale) · Lower is better · Source legend: DuckDB1.3

Already fast native plans leave less room for improvement; rewrite overhead can be visible.

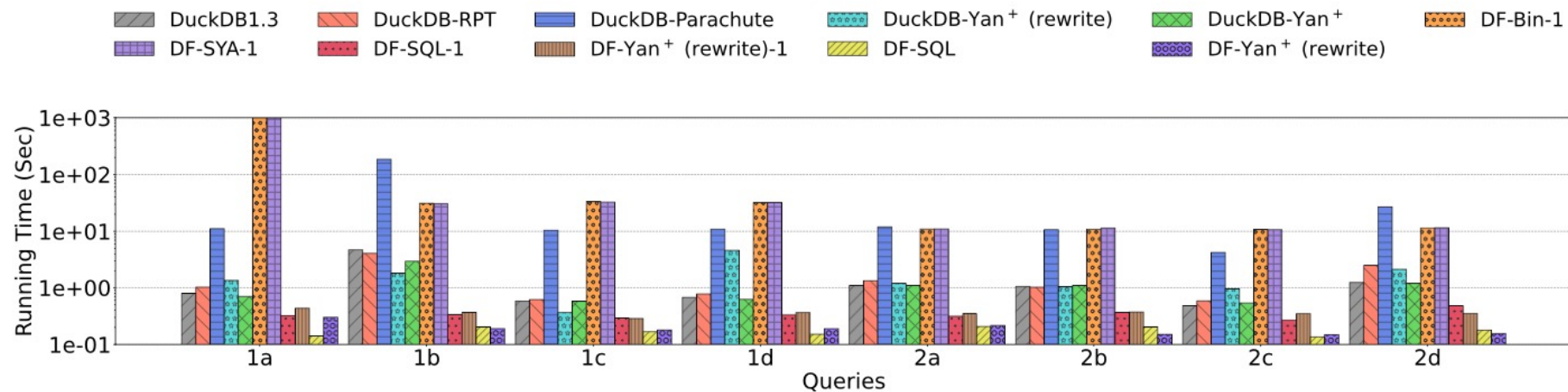
Experiments: DSB-AGG and DSB-SPJ



Running time (seconds, log scale) · Lower is better · Source legend: DuckDB1.3

Some queries gain substantially; others cluster near the same subsecond runtime.

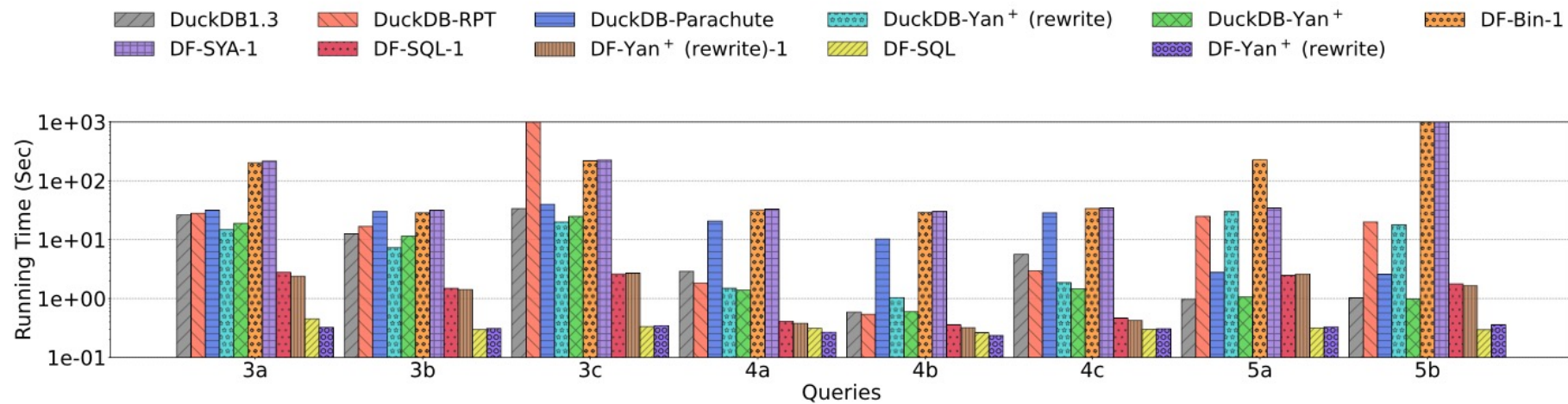
Experiments: JOB



Running time (seconds, log scale) · Lower is better · Source legend: DuckDB1.3

Compare integrated execution and SQL rewrite against the labeled native and alternative baselines.

Experiments: JOB



Running time (seconds, log scale) · Lower is better · Source legend: DuckDB1.3

Changing query structure changes the relative ranking and the size of the benefit.

Thank you!
Questions and discussion