CSE599d: Advanced Query Processing

Lecture 12: Modern Approaches to Yannakakis' Algorithm

Dan Suciu

University of Washington

Announcements

- HW3 due: Friday Feb. 27

Review of YA

Acyclic Query

Q is acyclic if it admits a join tree, which is a tree T where:

- The nodes in T are in 1-1 correspondence with the atoms in Q.
- T satisfies the running intersection property: for any variable, the set of nodes that contain it forms a connected component.

YA for a Boolean Query

Choose a root of the join tree.

- Traverse the tree bottom-up and set $R_i := R_i \ltimes R_{\texttt{child}(i)}$.

$R_{\texttt{root}} = \emptyset$ then return FALSE. Otherwise return TRUE.

Runtime: $O(|\texttt{IN}|)$

Semijoin Reduction Algorithm

Yannakakis' algorithm for full semi-join reduction of an acyclic query Q.

- Traverse the tree bottom-up and set $R_i := R_i \ltimes R_{\texttt{child}(i)}$.
- Traverse the tree top-down and set $R_i := R_i \ltimes R_{\texttt{parent}(i)}$.

Runtime: $O(|\texttt{IN}|)$

YA for a Full Conjunctive Query

Phase 1: Semijoin Reduction.

- Traverse the tree bottom-up and set $R_i := R_i \ltimes R_{\texttt{child}(i)}$.
- Traverse the tree top-down and set $R_i := R_i \ltimes R_{\texttt{parent}(i)}$.

YA for a Full Conjunctive Query

Phase 1: Semijoin Reduction.

- Traverse the tree bottom-up and set $R_i := R_i \ltimes R_{\texttt{child}(i)}$.
- Traverse the tree top-down and set $R_i := R_i \ltimes R_{\texttt{parent}(i)}$.

Phase 2: Join Computation. Initialize $\text{Out}_0 := \{()\}$ (empty tuple).

- Traverse the tree top-down and set $\text{Out}_i := \text{Out}_{i-1} \bowtie R_i$.

Return Out_n.

Runtime: $O(|\texttt{IN}| + |\texttt{OUT}|)$

YA for a Full Conjunctive Query

Phase 1: Semijoin Reduction.

- Traverse the tree bottom-up and set $R_i := R_i \ltimes R_{\texttt{child}(i)}$.
- ~~Traverse the tree top-down and set $R_i := R_i \ltimes R_{\texttt{parent}(i)}$.~~

Phase 2: Join Computation. Initialize $\text{Out}_0 := \{()\}$ (empty tuple).

- Traverse the tree top-down and set $\text{Out}_i := \text{Out}_{i-1} \bowtie R_i$.

Return Out_n.

Runtime: $O(|\texttt{IN}| + |\texttt{OUT}|)$

GYO Acyclicity Test (Graham and Yu-Oszoyoglu)

Repeat:

- Remove an isolated variable (i.e. occurs in only one atom).
- Remove an ear (i.e. atom contain in another atom).

Q is a acyclic iff result is one empty edge.

Summary

- YA runs in time $O(|\texttt{IN}| + |\texttt{OUT}|)$.
- Semijoin reduction: first up, then down.
- Only for acyclic queries.
- Most SQL queries in practice are acyclic.

Main attraction today: Query Robustness

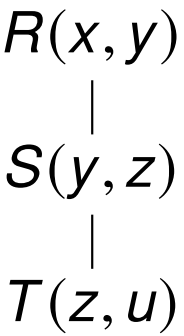
Limitations of Yannakakis' Algorithm

Limitations

- Requires additional (semi)-joins: joins are expensive!

- Requires a rigid join order.

Additional Joins



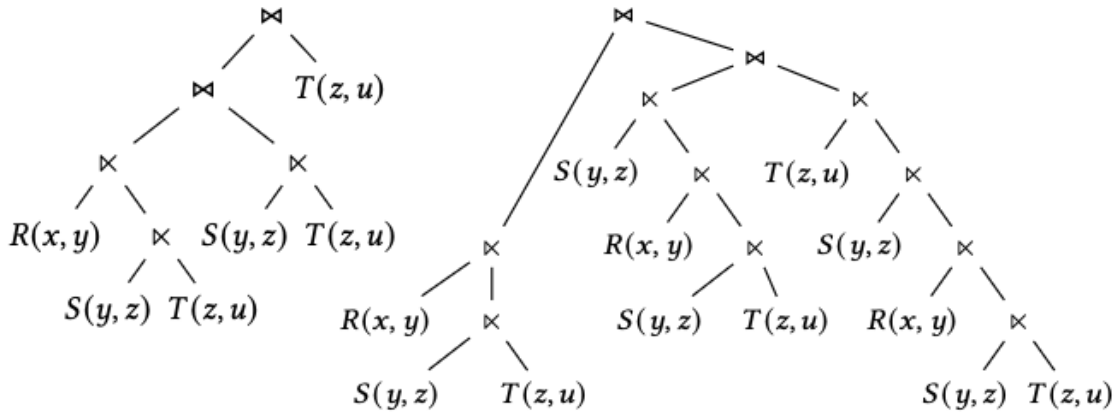
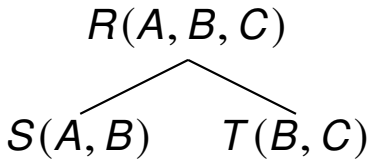


Figure 2: Semijoin plans induced by YA on join tree J_3 (Fig. 1a). Left: pass two and three combined. Right: all three passes.

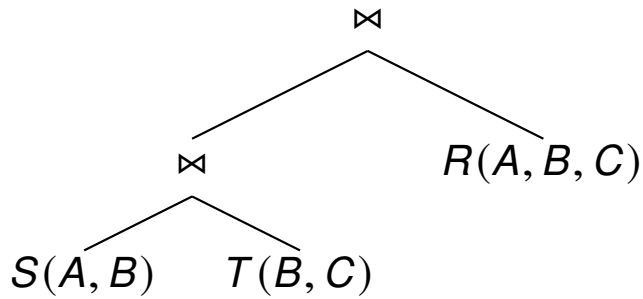
[Bekkers et al., 2025]

Rigid Join Order

Not every query plan without cross-products ensures optimality:



A join tree (acyclic query)



Not optimal [Zhao et al., 2025]:

$R = \{(1,1,1), (2,1,2), \ldots, (n,1,n)\}$
$S = \{(1,1), (2,1), \ldots, (n,1)\}$
$T = \{(1,1), (1,2), \ldots, (1,n)\}$

For optimality: every join must be an edge in the join tree

Approaches to Make YA Practical

- Use Bloom filters [Zhao et al., 2025]
- New physical operators: Lookup and Expand [Birler et al., 2024]
- Nested relations (factorized databases) [Bekkers et al., 2025]

Many others that we won't discuss.

Bloom Filters

Overview

We follow [Zhao et al., 2025]

- Replace semi-joins with a Bloom filter.
- Allow more flexibility for choosing a query plan.

Quick Review

A = array of m bits; $h_1, \ldots, h_k$ independent hash functions.

- Insert(v): set $A[h_i(v)] := 1$, for $i = 1, k$.
- Test(v): return $\bigwedge_{i=1,k} A[h_i(v)]$.
- One-sided guarantee: if Test(v) $= 0$ then v was not inserted.

For optimal FPR ε : $k = \frac{m}{n} \ln 2$ $\varepsilon = \exp\left(-\frac{m}{n} \ln^2 2\right)$

Main Heuristics

Instead of a semijoin: $\boxed{R := R \ltimes S}$

compute a Bloom filter on S, use it to probe R: $\boxed{R := R \ltimes_b S}$

If $|R| \geq |S|$ then $\boxed{R := R \ltimes_b S}$ is better than $\boxed{S := S \ltimes_b R}$

because $\varepsilon = \exp\left(-\frac{m}{n} \ln^2 2\right)$

Choosing a Join Trees

Ideal: all semi-joins are from small to large. Not always possible:

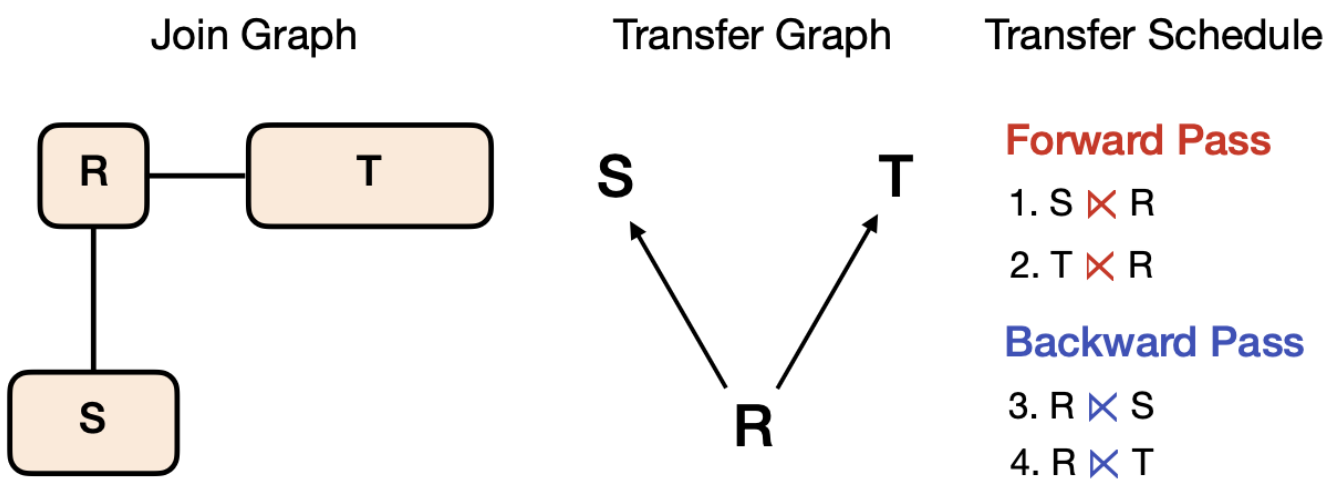


Fig. 2. An example of the `Small2Large` algorithm in the original Predicate Transfer

Alternative Characterization of Join Trees

Let G be the query graph of Q

what is that?

Alternative Characterization of Join Trees

Let G be the query graph of Q

what is that?

The weight of an edge (R, S) in G
is the number of common attributes:

$$\boxed{w(R,S) = |\text{attrs}(R) \cap \text{attrs}(S)|}$$

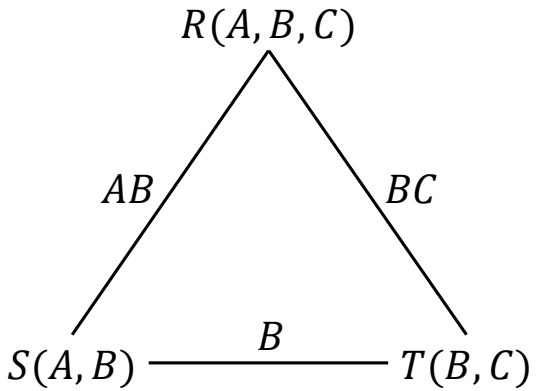
Alternative Characterization of Join Trees

Let G be the query graph of Q

what is that?

The weight of an edge (R, S) in G is the number of common attributes:

$$w(R,S) = |\text{attrs}(R) \cap \text{attrs}(S)|$$



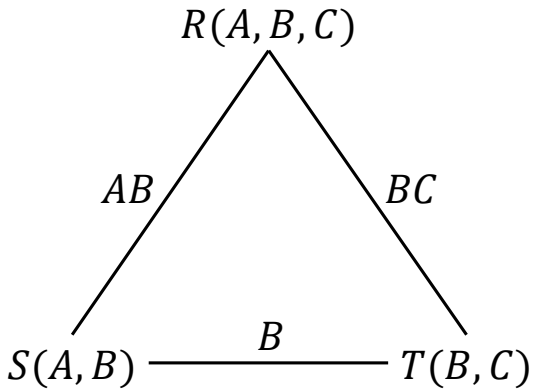
Alternative Characterization of Join Trees

Let G be the query graph of Q

what is that?

The weight of an edge (R, S) in G is the number of common attributes:

$$w(R,S) = |\text{attrs}(R) \cap \text{attrs}(S)|$$

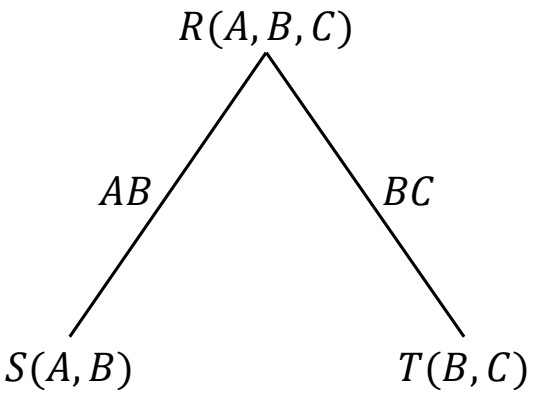


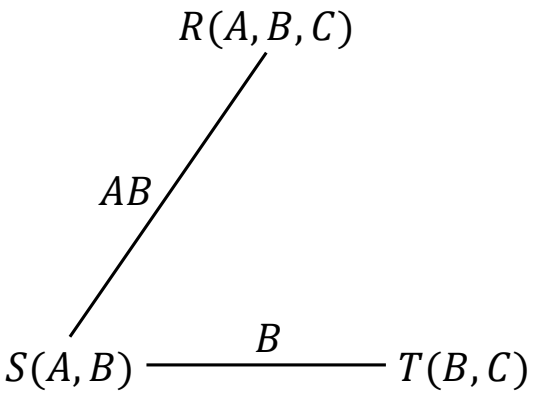
Theorem A tree $T \subseteq G$ is a join tree iff T is a maximal spanning tree of G.

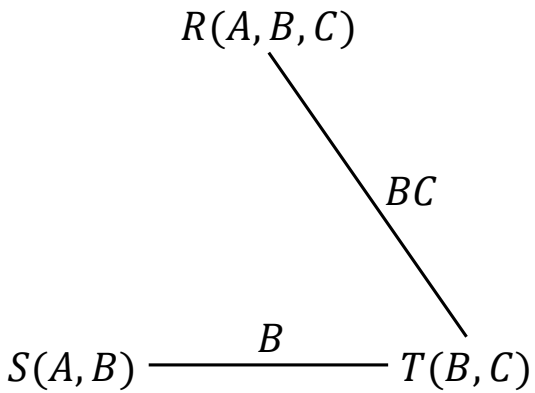
(Proof on next slides)

Proof (in class)

Which trees are join trees, and what are their weights?

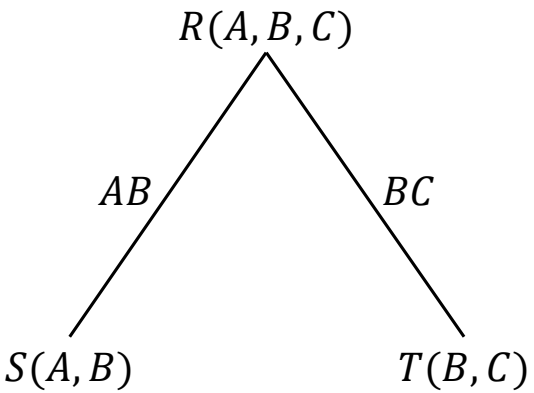


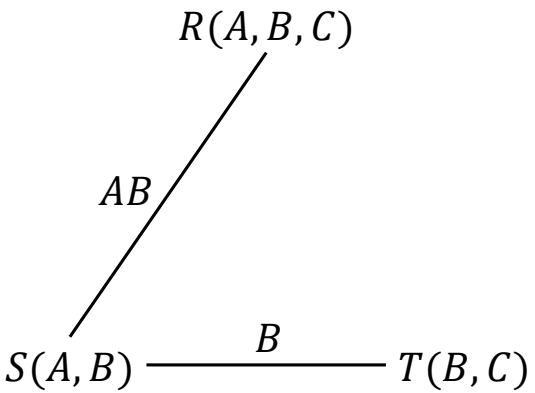


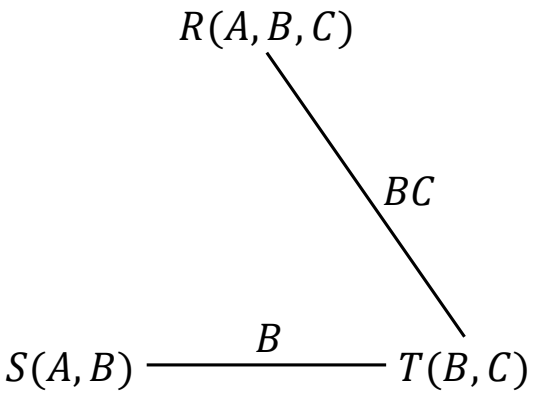


Proof (in class)

Which trees are join trees, and what are their weights?



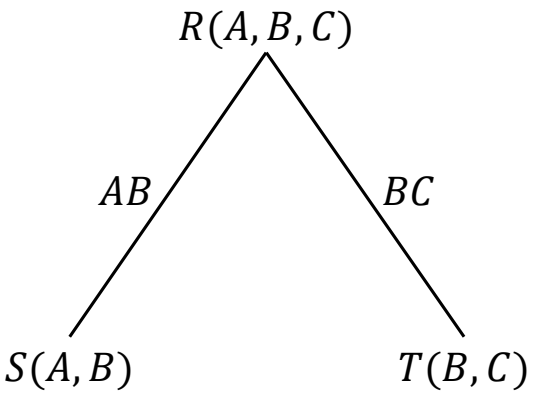


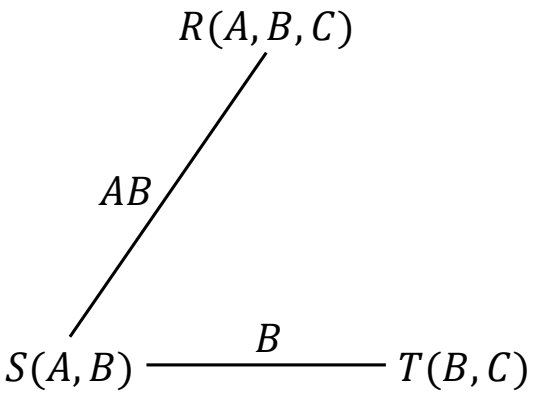


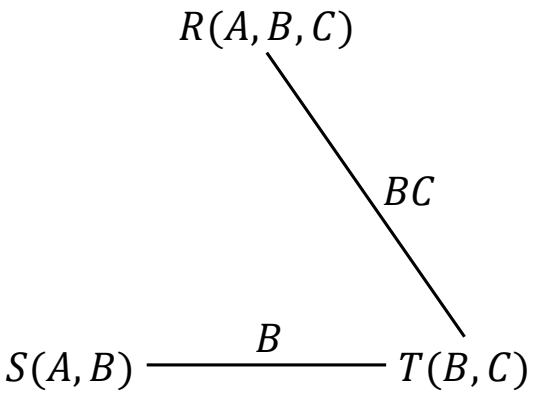
What is the weight of a join tree? of a non-join tree?

Proof (in class)

Which trees are join trees, and what are their weights?







What is the weight of a join tree? of a non-join tree?

$\sum_i \texttt{arity}(R_i) - |\text{Vars}(Q)|$ for JT, strictly smaller otherwise

Choosing a Join Tree

Run Prim's algorithm for computing a maximal spanning tree.
Break ties in favor of larger relations.

Algorithm 1: `LargestRoot`

Input: join graph G_q
Output: tree T

$T \leftarrow \emptyset$; $\mathcal{R} \leftarrow$ all relations; $\mathcal{R}' \leftarrow \{R_{max}\}$;
while $\mathcal{R}' \neq \mathcal{R}$ **do**
  Find an edge $e = \{R, S\} \in E(G_q)$ with the largest weight such that $R \in \mathcal{R} \setminus \mathcal{R}', S \in \mathcal{R}'$. Choose the edge with the largest R to break ties;
  Add e to T with direction from R to S;
  $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{R\}$;
end
return T;

At each tie we can choose between multiple join trees.

Choosing a Query Plan

The standard optimizer searches for an optimal bushy plan.

Problem: not every bushy plan guarantees optimality in the sense of YA.

Need to restrict all subplans to be safe.

Safe Join Order

Q' = a subquery (subset of atoms) of Q.

Q' is safe if for every fully reduced DB, $\boxed{Q'(DB) = \Pi_{\text{attrs}(Q')}(Q(DB))}$

This implies $|Q'(DB)| \leq |Q(DB)|$.

Safe Join Order

Q' = a subquery (subset of atoms) of Q.

Q' is safe if for every fully reduced DB, $\boxed{Q'(DB) = \Pi_{\text{attrs}(Q')}(Q(DB))}$

This implies $|Q'(DB)| \leq |Q(DB)|$.

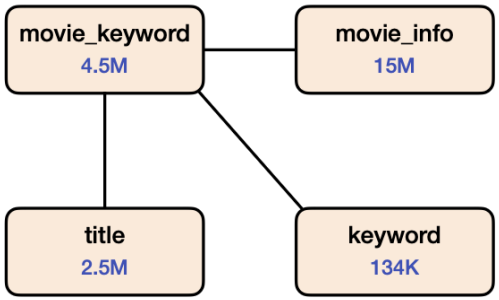
$Q = R(A,B,C) \bowtie S(A,B) \bowtie T(B,C)$, $Q' = S(A,B) \bowtie T(B,C)$ unsafe

We want every subplan to be safe

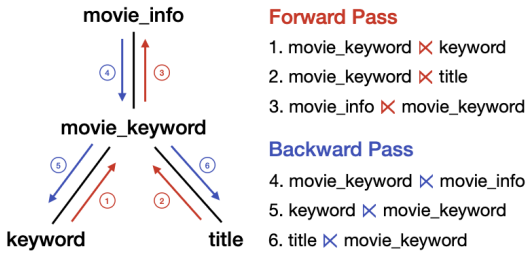
Testing for Safety

- Compute a maximal spanning tree T_0 for Q.
- Compute a maximal spanning tree T' for Q'.
- Extending T' to maximal spanning tree T for Q.
- If weight(T_0) = weight(T) then Q' is safe.

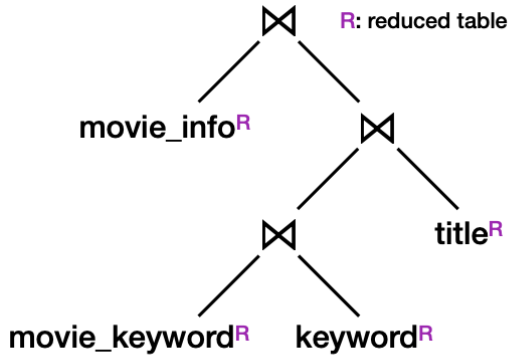
Putting it Together (by Example Only)



(a) Join Graph

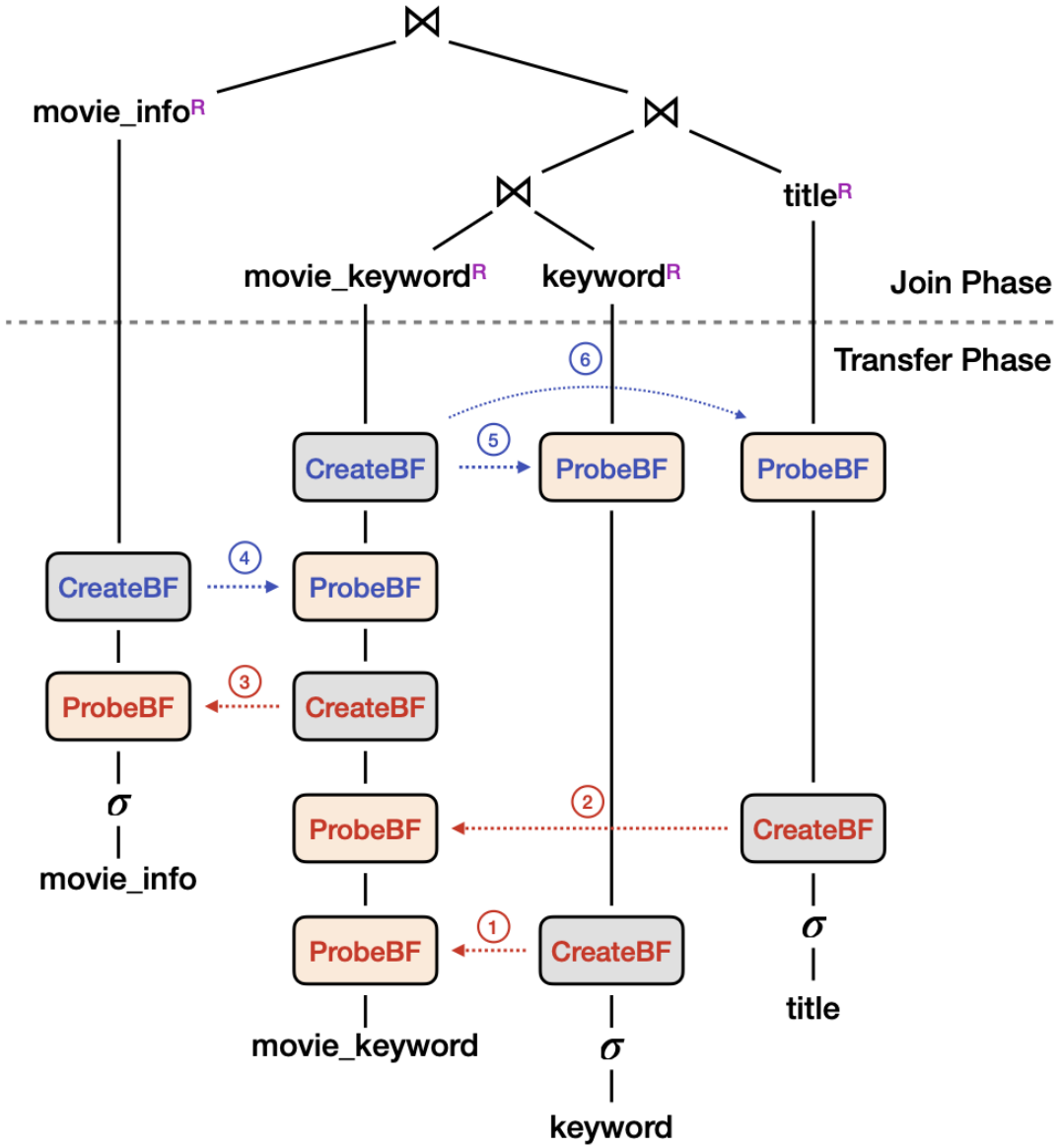


(b) Semi-Join Phase



(c) Join Phase

Putting it Together (by Example Only)



Discussion

- Bloom filter approach to YA: simple idea, more efficient than semijoins.
- But requires heavy engineering to incorporate in existing query engines.

Lookup and Expand

Overview

Goal: implement YA with minimum changes to an existing engine

[Birler et al., 2024]

Review: Hash-Join

```
1  # Hash table build
2  ht = {}
3  for k, v in buildInput:
4      ht[k] = v
5
6  # Hash table probe
7  for k, v in probeInput:
8      iterator = ht.find(k) # Lookup
9      if not iterator.done():
10         do: # Expand
11             produce(k, *iterator)
12             iterator.step()
13         while not iterator.done()
```

The probe phase has two steps: Lookup and Expand

Make them separate operators!

Lookup and Expand

Lookup returns pointer to the hash table entry, or NULL

Expand traverses the list at that entry.

```
1  def lookup_consume(produce, tuple, hashTable):
2    # Look for the key in the hash table
3    iterator = hashTable.find(tuple.key)
4    if iterator.done():
5      return
6    # Return tuple and iterator over matches
7    produce(tuple, iterator)
8
9  def expand_consume(produce, tuple):
10   # Extract the iterator from the tuple
11   currentMatch = tuple.iterator
12   do: # Loop over all matches
13     produce(tuple, *currentMatch)
14     currentMatch.step()
15   while not currentMatch.done()
```

(This is the push-based API from the Query Compiler paper in 544)

Examples

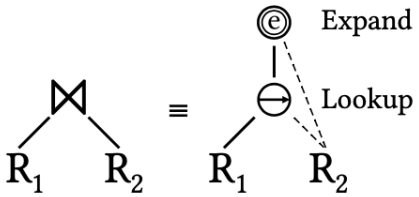


Figure 2: Binary hash join and equivalent Lookup & Expand.

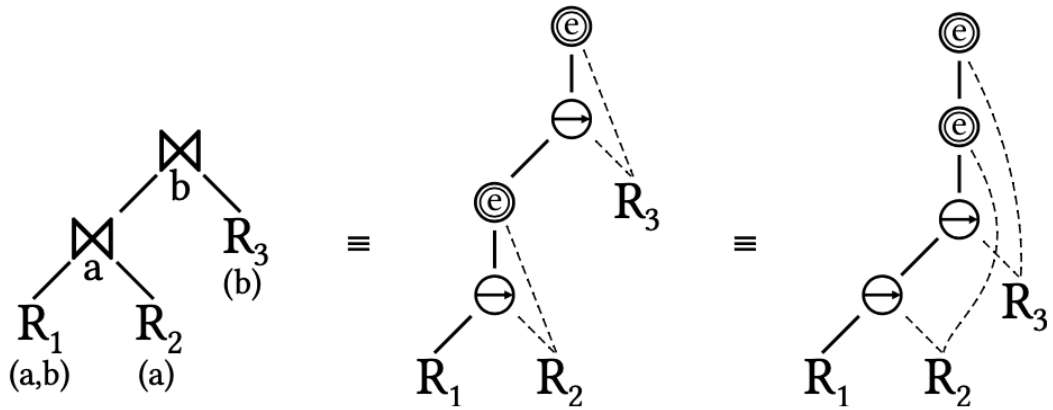


Figure 3: Threeway join, equivalent Lookup & Expand, and Lookup pushdown.

Example

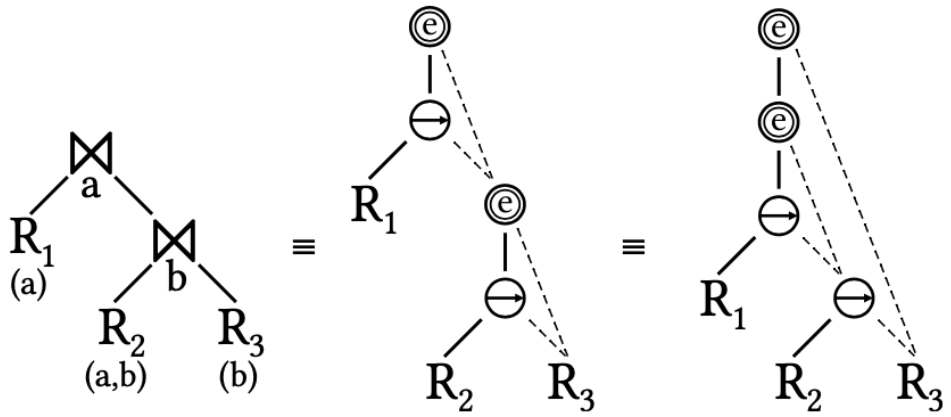


Figure 4: Threeway join, equivalent Lookup & Expand, and Expand pullup.

Discussion

- Lookup/Expand become independent operators, allows most freedom for the query optimizer.

- May still need to impose restrictions if we want the guarantees of YA.

Nested Relations

Main Idea

Represent the result of a join in a nested relations.

This idea is called factorized databases [Olteanu and Závodný, 2015]

This postpones the cross products inherent in a join, potentially leading to the benefit of YA.

Nested Relations

Nested schema: set of attributes and/or nested schemas.

Nested relation with schema $\{x, \{y\}, \{u, \{v\}\}\}$

x	$\{y\}$	$\{u, \{v\}\}$
1	10 20	a | m n b | k n p
2	...	

Nested Relational Algebra is Relational Algebra plus two operators:

$\texttt{nest}_x : \{x, y\} \rightarrow \{x, \{y\}\}$ a.k.a. `group-by` or γ.

$\texttt{unnest}_y : \{x, \{y\}\} \rightarrow \{x, y\}$ a.k.a. μ

Example: Using NRA for YA

$$R(x,y) \bowtie S(y,z) \bowtie T(z,u)$$

Several factorized representations are possible for the output:

$$\begin{aligned} P(y,z,\{u\}) &:= S(y,z) \ltimes_z \mathtt{nest}_z T(z,u) \\ F(x,\{y,\{z,\{u\}\}\}) &:= R(x,y) \ltimes_y \mathtt{nest}_y(P) \end{aligned}$$

For the query answer, unnest F.

Example: Using NRA for YA

$R(x,y) \bowtie S(y,z) \bowtie T(z,u)$

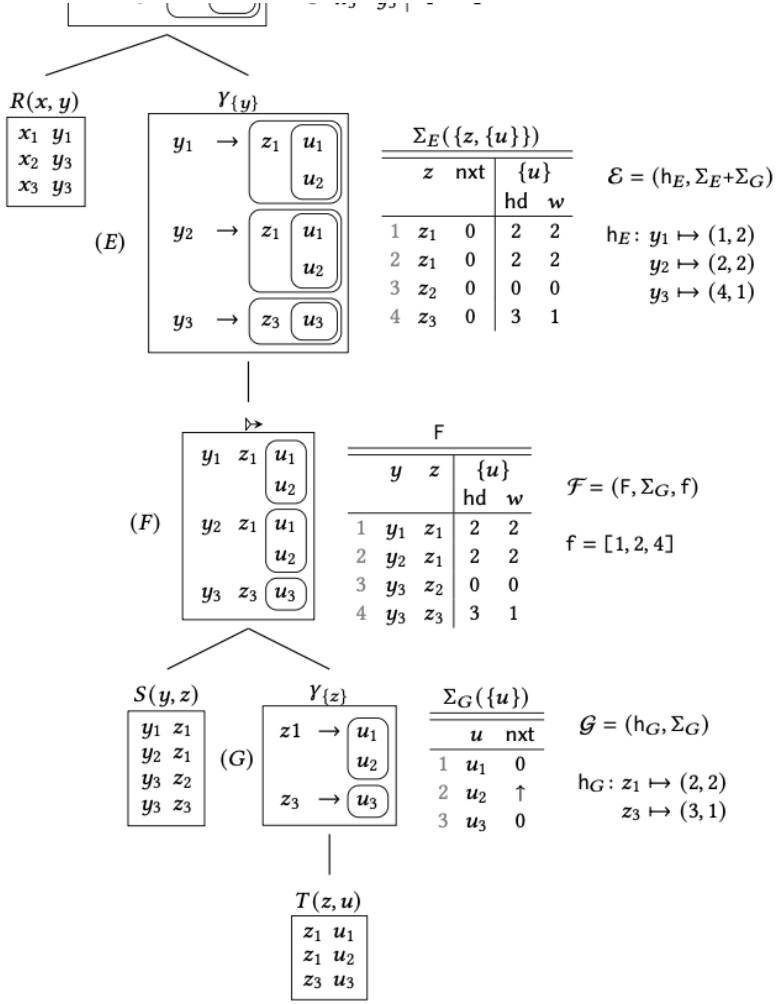
Several factorized representations are possible for the output:

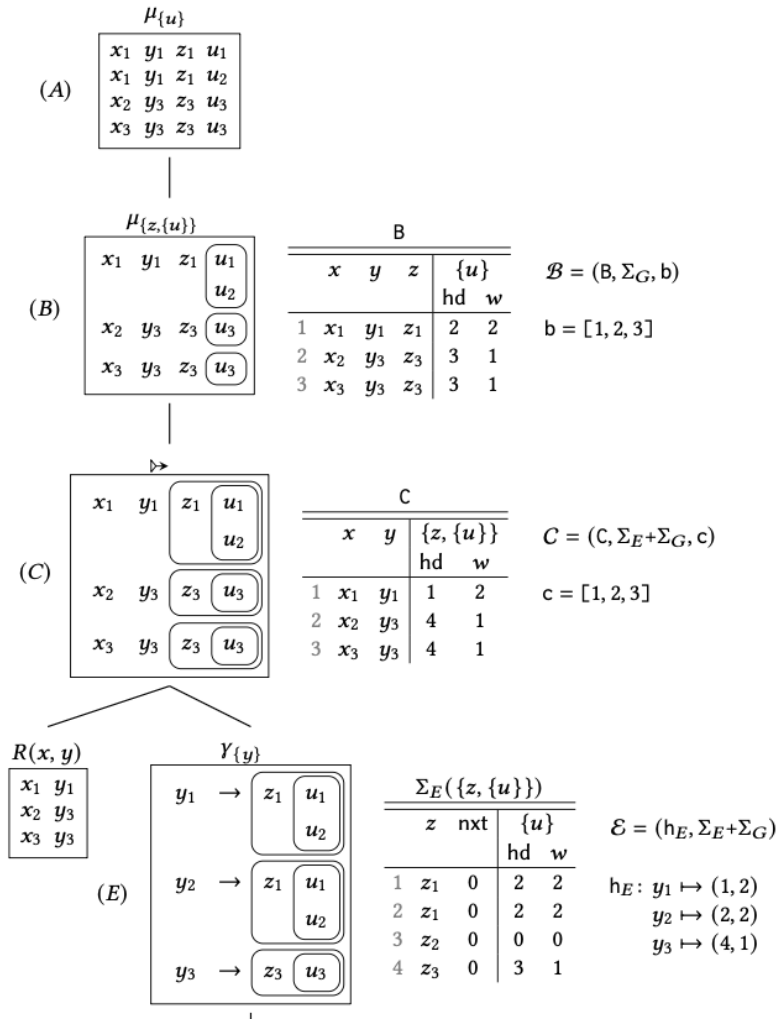
$$P(y,z,\{u\}) := S(y,z) \ltimes_z \texttt{nest}_z T(z,u)$$
$$F(x,\{y,\{z,\{u\}\}\}) := R(x,y) \ltimes_y \texttt{nest}_y(P)$$

$$P(y,z,\{u\}) := [\text{same}]$$
$$F(\{x\},y,\{z,\{u\}\}) := \texttt{nest}_y R(x,y) \ltimes_y \texttt{nest}_y P$$

For the query answer, unnest F.

Example: Using NRA for YA





Main Takeaways

- Renewed interest in YA comes from the quest for robust query processing: less sensitive to cardinality estimation erros.

- Surprisingly, SQL Server already supports YA, without knowing!
  hfill[Zhao et al., 2026]


Bekkers, L., Neven, F., Vansummeren, S., and Wang, Y. R. (2025).
Instance-optimal acyclic join processing without regret: Engineering the yannakakis algorithm in column stores.
Proc. VLDB Endow., 18(8):2413–2426.


Birler, A., Kemper, A., and Neumann, T. (2024).
Robust join processing with diamond hardened joins.
Proc. VLDB Endow., 17(11):3215–3228.


Olteanu, D. and Závodný, J. (2015).
Size bounds for factorised representations of query results.
ACM Trans. Database Syst., 40(1):2:1–2:44.


Zhao, H., Tian, Y., Alotaibi, R., Ding, B., Bruno, N., Camacho-Rodríguez, J., Papadimos, V., Juárez, E. C., Galindo-Legaria, C. A., and Curino, C. (2026).
I can't believe it's not yannakakis: Pragmatic bitmap filters in microsoft SQL server.
In 16th Conference on Innovative Data Systems Research, CIDR 2026, Chaminade, CA, USA, January 18-21, 2026. www.cidrdb.org.


Zhao, J., Su, K., Yang, Y., Yu, X., Koutris, P., and Zhang, H. (2025).
Debunking the myth of join ordering: Toward robust SQL analytics.
Proc. ACM Manag. Data, 3(3):146:1–146:28.