Engineering the Yannakakis Algorithm in Column Stores

Liese Bekkers

22 November 2024

DSI DATA SCIENCE INSTITUTE
UHASSELT
WWW.UHASSELT.BE/DSI

Team Work

Liese Bekkers
UHasselt

Frank Neven
UHasselt

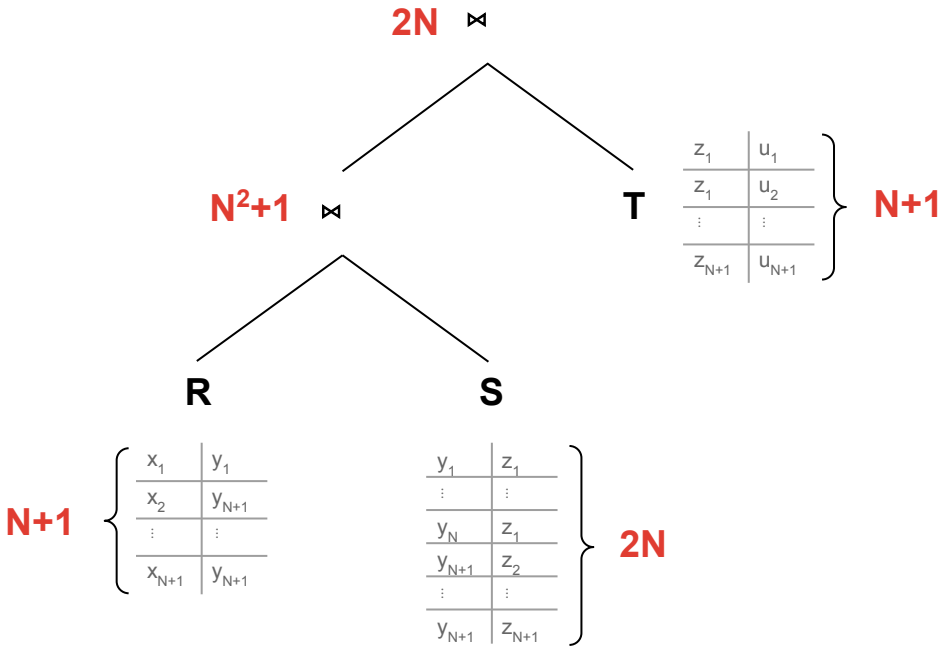
Stijn Vansummeren
UHasselt

Yisu Remy Wang
UCLA

Joins: Large Intermediate Results

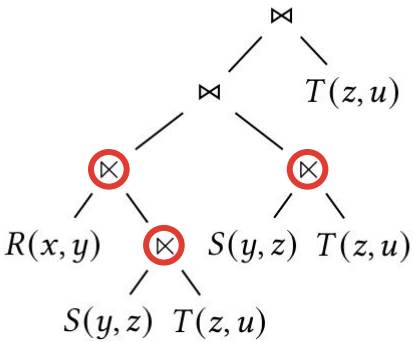
tuples

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



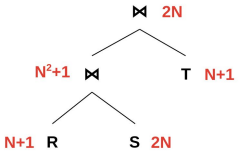
Yannakakis

- Acyclic queries
- Instance-optimal: O(|IN|+|OUT|)
- Faster when many dangling tuples, constant-factor slow down otherwise



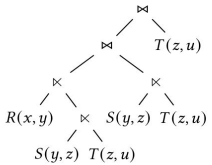
BINARY JOIN

✗ Slow when blowup of intermediate, dangling tuples
✓ Fast otherwise



YANNAKAKIS

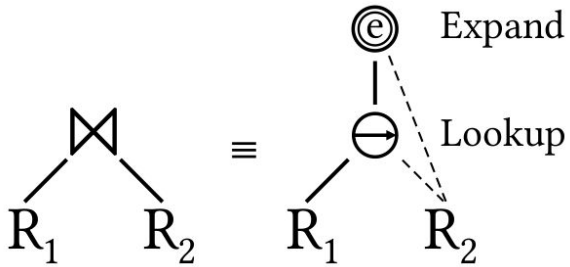
✓ Fast when many dangling tuples
✗ Slow otherwise



BEST OF BOTH WORLDS

Fast when:
✓ Many dangling tuples
✓ No dangling tuples

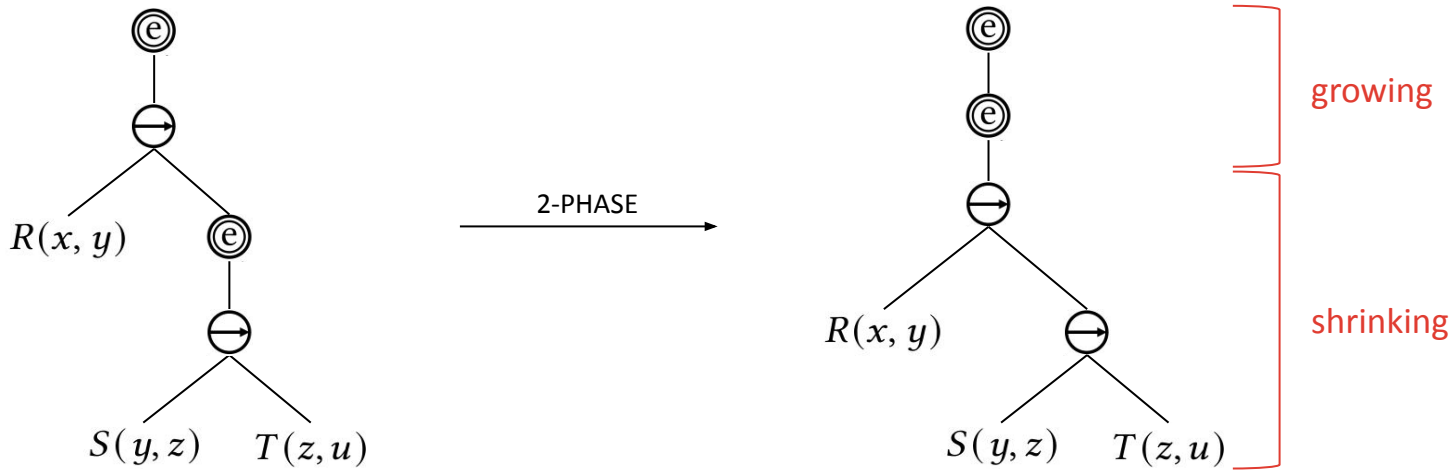
Split hash join into lookup & expand



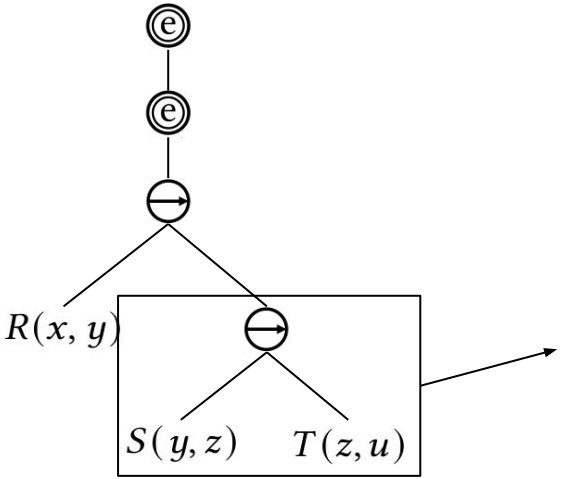
[1] A. Birler, A. Kemper, and T. Neumann. 2024. Robust Join Processing with Diamond Hardened Joins. *Proc. VLDB Endow.* 17, 11 (July 2024), 3215–3228.

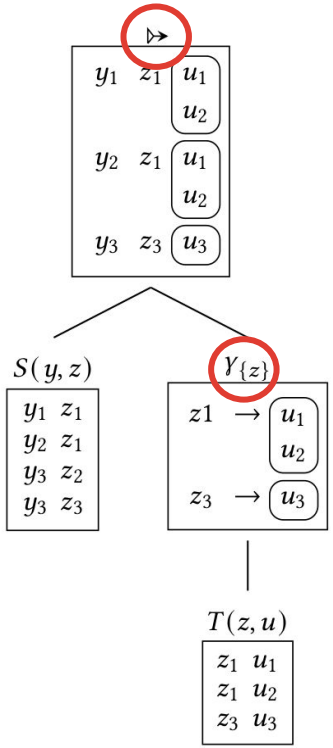
O(|IN|+|OUT|)

INSTANCE-OPTIMAL

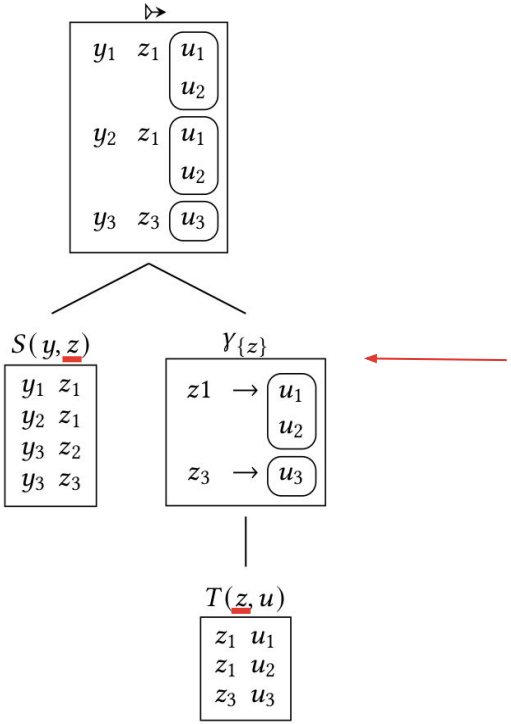


Nested Semijoin Algebra (NSA): logical model

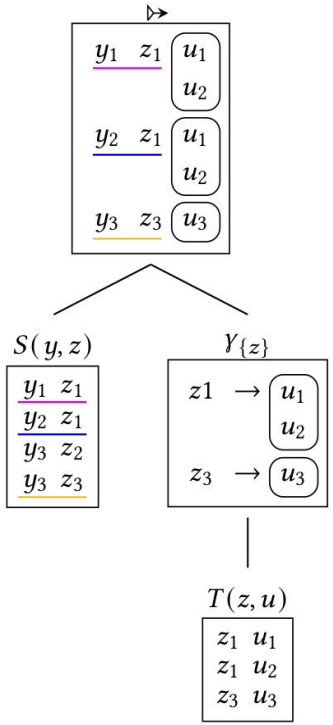




Nested Semijoin Algebra (NSA): logical model



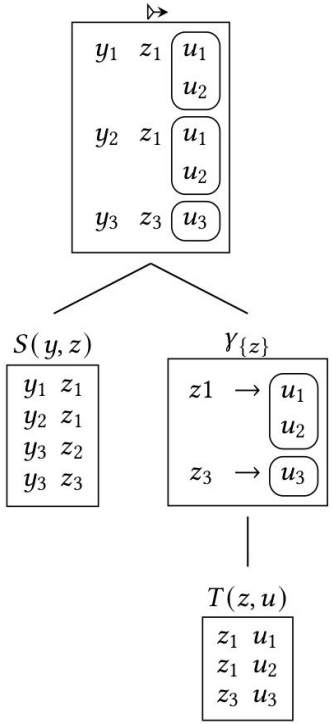
Nested Semijoin Algebra (NSA): logical model



NSA = relational algebra + L&E

Query Shredding: physical model

- Nested relation is represented by a **collection of flat relations**
- Physical implementation of NSA operators work on the **shredded representation**



Instance-optimal NSA

Theorem 5.3. *Every 2-phase NSA expression that maps flat input relations to flat output relations is evaluated in time* $O(\text{IN} + \text{OUT})$ *by shredded processing, where* IN *is the sum of the cardinalities of the expression's flat input relations, and* OUT *is the output cardinality.*

ASYMPTOTIC COMPLEXITY

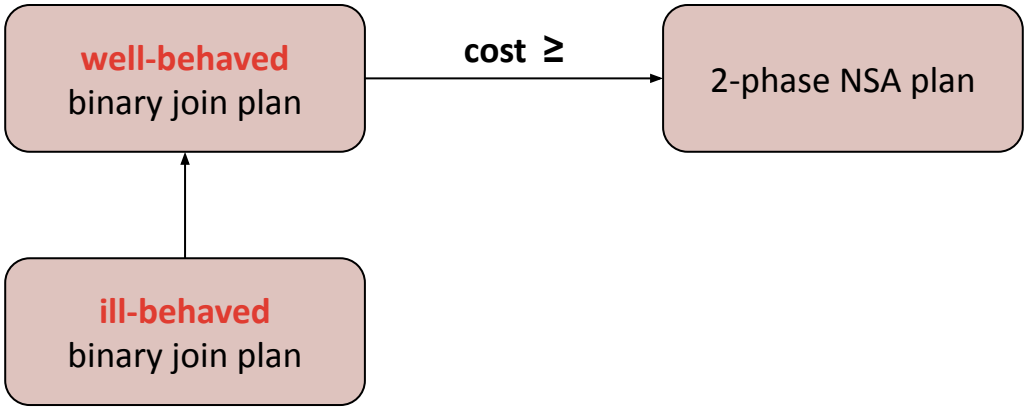
Theorem 5.5. *A join query Q can be evaluated by means of a 2-phase NSA join plan if and only if Q is acyclic.*

Detailed cost model

Includes probe cost, build cost, input data accesses

- Cost of **well-behaved** binary join plan ≥ cost of equivalent 2-phase NSA plan
- Rewrite **ill-behaved** into well-behaved while minimizing additional costs

Reoptimize existing query plans, while guaranteeing instance-optimality

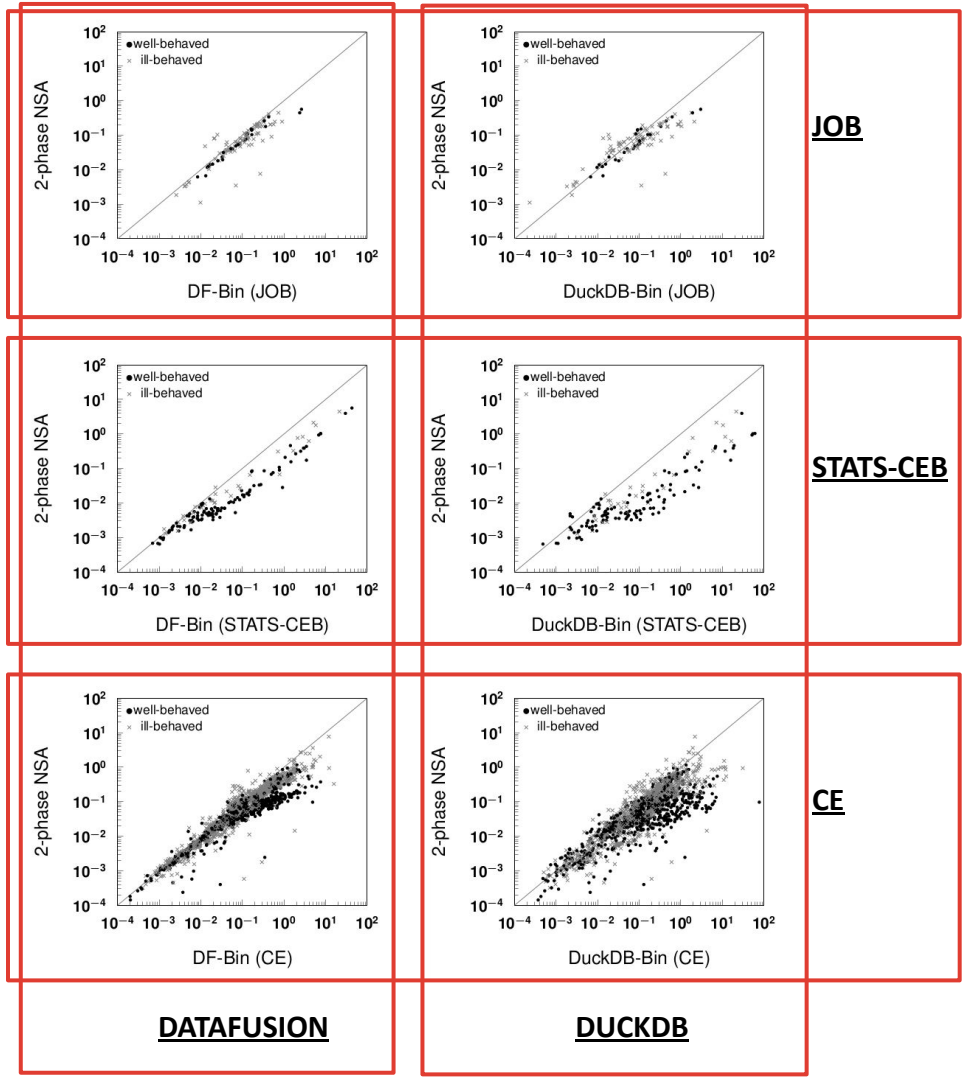


Experimental Evaluation

APACHE DATAFUSION™

- 3 benchmarks
- 1849 acyclic queries
- Baseline: binary plan in Datafusion/DuckDB
- **Faster** than Datafusion on **88.7%** of all queries
- **Faster** than DuckDB on **84.4%** of all queries

Baseline	Max. speedup	Max. slowdown
Datafusion (binary join)	188x (37s)	5.8x (0.65s)
DuckDB (binary join)	801x (196s)	6x (5.4s)



Instance-Optimal Acyclic Join Processing Without Regret: Engineering the Yannakakis Algorithm in Column Stores

Liese Bekkers
UHasselt, Data Science Institute

Frank Neven
UHasselt, Data Science Institute

Stijn Vansummeren
UHasselt, Data Science Institute

Yisu Remy Wang
University of California, Los Angeles

ABSTRACT

Acyclic join queries can be evaluated instance-optimally using Yannakakis' algorithm, which avoids needlessly large intermediate results through semi-join passes. Recent work proposes to address the significant hidden constant factors arising from a naive implementation of Yannakakis by decomposing the hash join operator into two suboperators, called Lookup and Expand. In this paper, we present a novel method for integrating Lookup and Expand plans in interpreted environments, like column stores, formalizing them using Nested Semijoin Algebra (NSA) and implementing them through a shredding approach. We characterize the class of NSA expressions that can be evaluated instance-optimally as those that are 2-phase: no 'shrinking' operator is applied after an unnest (i.e., expand). We introduce Shredded Yannakakis (SYA), an evaluation algorithm for acyclic joins that, starting from a binary join plan, transforms it into a 2-phase NSA plan, and then evaluates it through the shredding technique. We show that SYA is provably robust (i.e., never produces large intermediate results) and without regret (i.e., is never worse than the binary join plan under a suitable cost model) on the class of well-behaved binary join plans. Our experiments on a suite of 1,849 queries show that SYA improves performance for 88.7% of the queries with speedups up to 188x, while remaining competitive on the other queries. We hope this approach offers a fresh perspective on Yannakakis' algorithm, helping system engineers better understand its practical benefits and facilitating its adoption into a broader spectrum of query engines.

1 INTRODUCTION

Computing joins efficiently has been a fundamental challenge in query processing since the inception of the relational model. Thanks to decades of research and engineering, contemporary query engines excel on common benchmark such as TPC-H featuring foreign-key joins of a limited number of relations. However, queries with up to a thousand of relations featuring many-to-many joins are not uncommon anymore in modern data analysis scenarios [6, 22, 27]. Unfortunately, for such queries, consistently finding a good join order is very difficult. At the same time, a poorly chosen join order will bring even state-of-the-art systems to their knees [25]. In recent work [4], Birler, Kemper, and Neumann (henceforth BKN) have dubbed the problem underlying this phenomenon the *diamond problem*: a poor query plan will compute subresults that are orders of magnitude larger than the output, even if these subresults are unnecessary to produce this final output—thereby wasting significant processing time.

Avoiding the diamond problem is intrinsically linked to query engine *robustness*: by limiting the sizes of intermediate results, the engine's runtime becomes bounded and predictable. How to avoid the diamond problem has in fact been a major topic in database theory for decades. From the concept of acyclicity [3, 10] and Yannakakis' seminal algorithm (YA) for optimally processing acyclic queries [38], over various notions of query width and query decompositions [12], to the more recent worst-case-optimal (WCO) [28, 29, 36] and factorized [18, 30] processing algorithms: much research has been done to identify and exploit structural properties of join queries that can either completely eliminate or bound the size of intermediate results. Although many of these techniques have been known for decades, they have not yet found wide-spread adoption in practical query engines. Indeed, most contemporary systems [1, 11, 19, 23, 26, 31] continue to use non-robust binary join plans for most queries, possibly resorting to WCO joins in certain cases—in particular for cyclic queries. The reason for this lack of adoption is that the above-mentioned research focuses on *asymptotic* complexity and optimizes for the worst-case input instance in avoiding the diamond problem. In fact, when implemented in a concrete system, these techniques can be significantly slower than traditional techniques on common-case instances and queries [4, 25]. From an engineering viewpoint we are hence in search for provably robust query processing algorithms *without regret*: competitive with traditional join algorithms while avoiding the diamond problem.

Towards this goal, BKN suggest to move to a larger space of query plans [4]. Concretely, they propose to decompose the traditional hash join operator into two suboperators called Lookup and Expand (or L&E for short). Lookup (denoted ⋉) finds the first match of a given tuple in a hash table, while expand (◎) iterates over the rest of the matches. By considering query plans where these two suboperators can be freely combined and reordered, dangling tuples (i.e., tuples that do not contribute to the output) can be eliminated as early as possible, hence avoiding the diamond problem. It is shown that L&E plans can be used to optimally process acyclic joins as well as effectively process certain cyclic joins when an additional operator is added. However, their approach to create L&E plans does not formally guarantee to always avoid the diamond problem (see point (4) below for more detail).

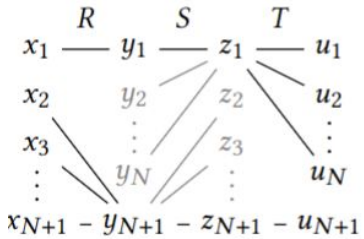
While BKN successfully implement L&E plans inside Umbra [26], a compiled query engine, it is unclear how to effectively implement L&E plans inside *interpreted* query engines. Indeed, Umbra generates code from L&E plans using the produce-consume interface [24] favored in compiled engines, and then rely on compiler optimizations to remove inefficiencies. Obtaining the same behavior in an interpreted engine poses two challenges. First, in the typical architecture of an interpreted engine, (physical) operators adopt a

ARXIV

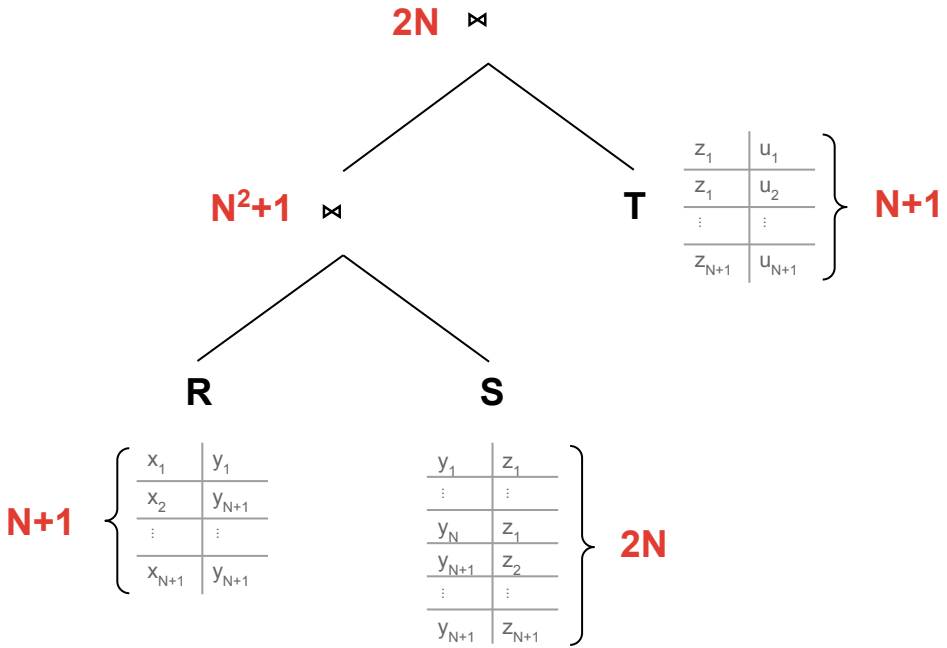
Joins: Large Intermediate Results

tuples

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



(c) Database db_2



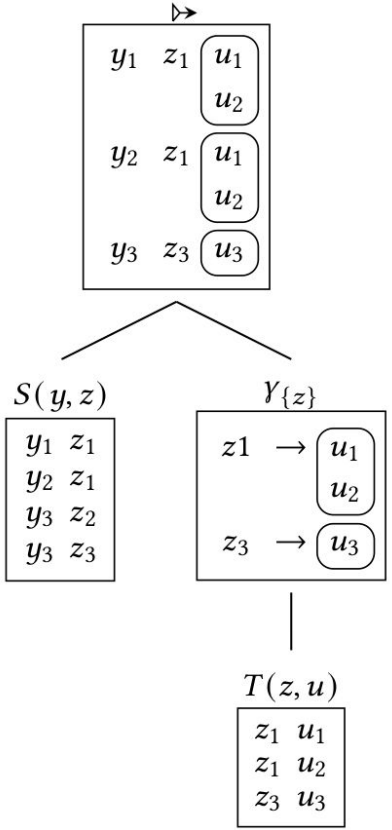
Nested Semijoin Algebra (NSA)

Standard relational operators:

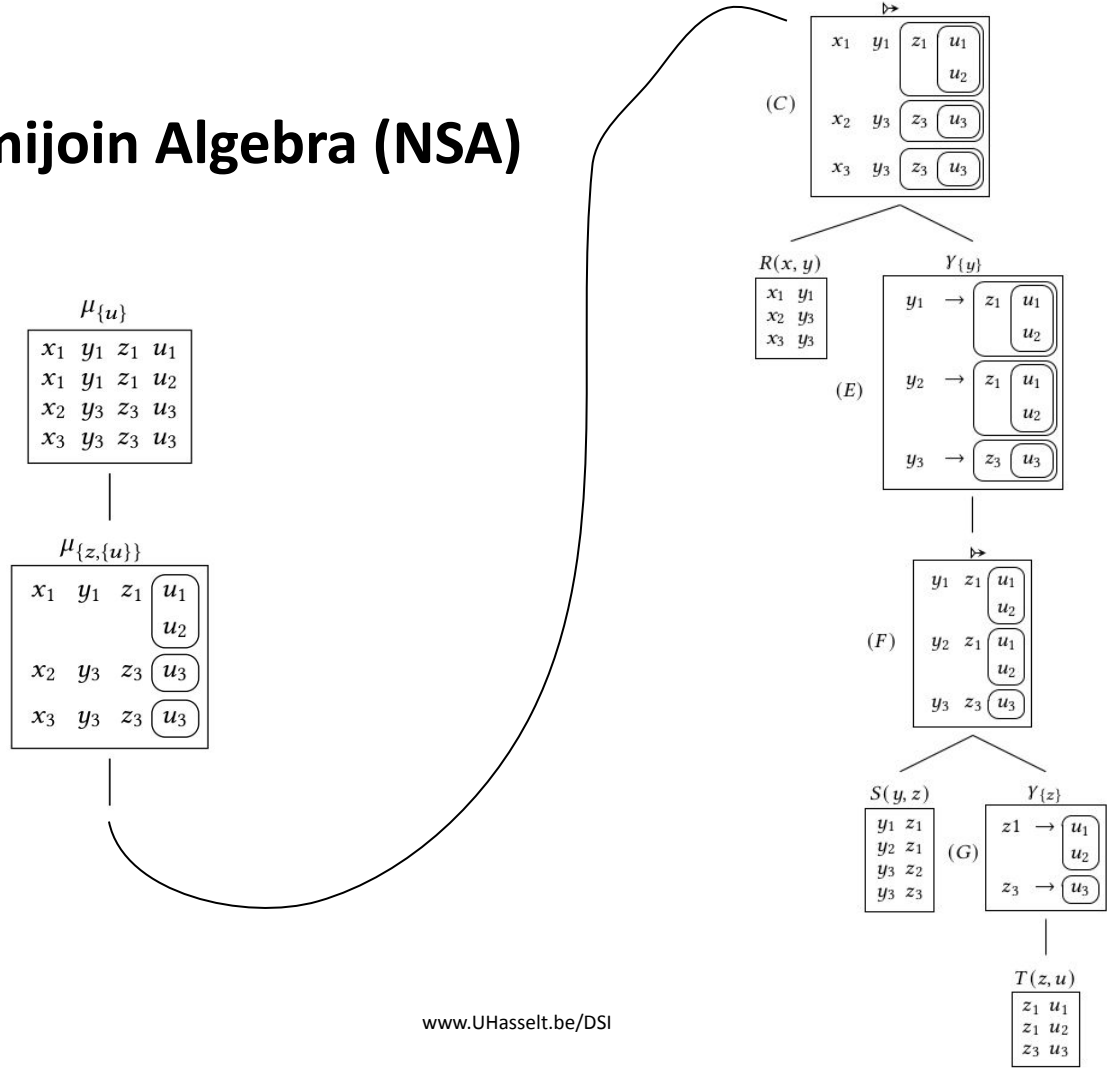
- filter (σ)
- projection (π)
- renaming (ρ)
- bag-union (U)
- bag-difference (—)

New operators:

- Group-by (ɣ)
- Nested semijoin (↣)
- Unnest (μ)

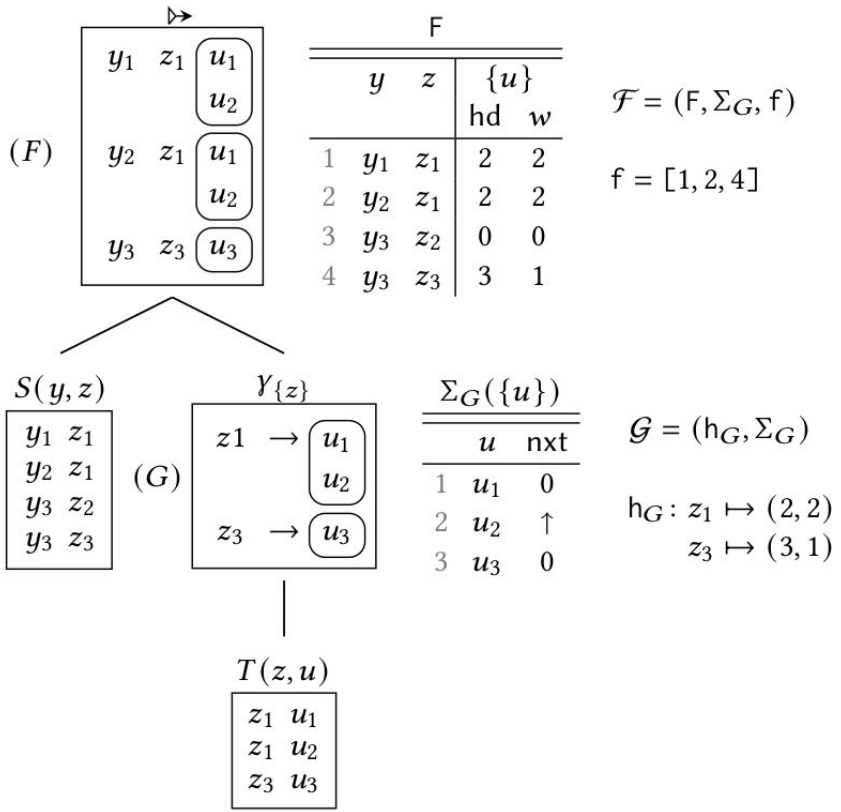


Nested Semijoin Algebra (NSA)

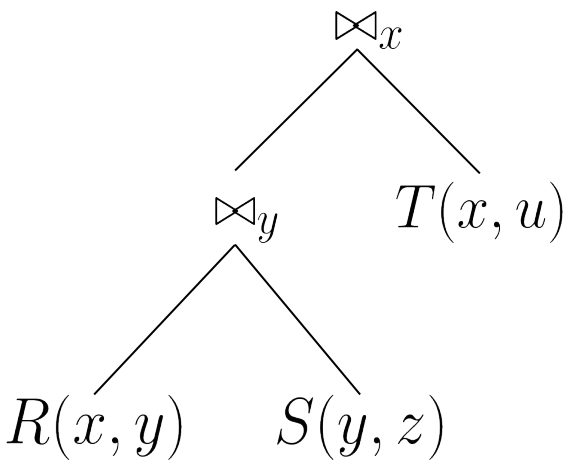


Query Shredding

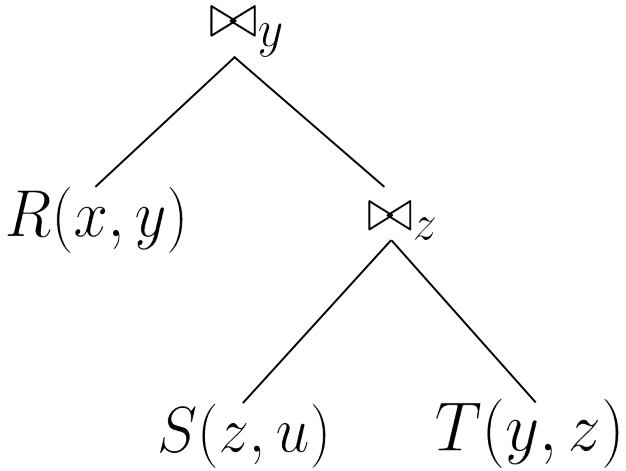
- Nested relation is represented by a **collection of flat relations**
- Physical implementation of NSA operators work on the **shredded representation**

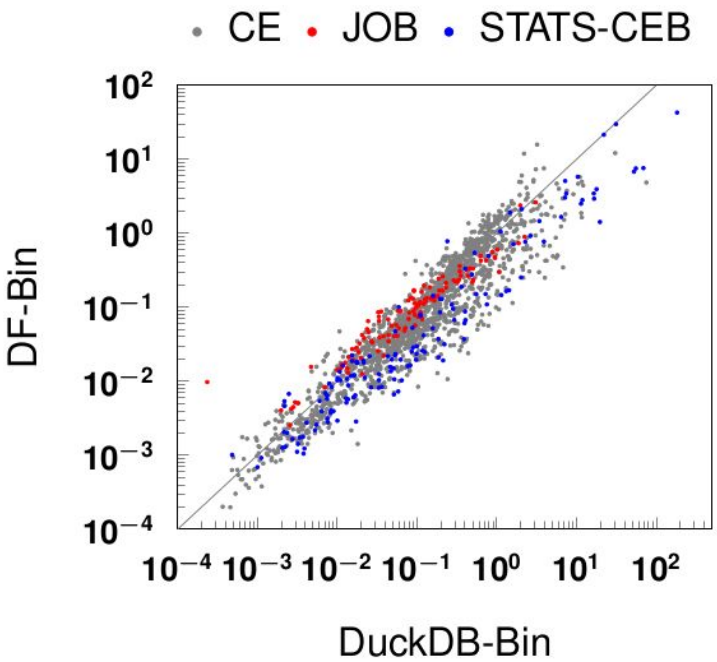


WELL-BEHAVED



ILL-BEHAVED




CE
JOB
STATS-CEB
DF-Bin
DuckDB-Bin
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-4} 10^{-3} 10^{-2} 10^{-1} 10^{0} 10^{1} 10^{2}