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

Liese Bekkers
UHasselt, Data Science Institute
liese.bekkers@uhasselt.be

Frank Neven
UHasselt, Data Science Institute
frank.neven@uhasselt.be

Stijn Vansummeren
UHasselt, Data Science Institute
stijn.vansummeren@uhasselt.be

Yisu Remy Wang
University of California, Los Angeles
remywang@cs.ucla.edu

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. 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 85.3% of the queries with speedups up to 62.5x, while remaining competitive on the other queries.

PVLDB Reference Format:
Liese Bekkers, Frank Neven, Stijn Vansummeren, and Yisu Remy Wang. .
PVLDB, 18(8): 2413 - 2426, 2025.
doi:10.14778/3742728.3742737

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/UHasselt-DSI-Data-Systems-Lab/code-reproducability-yannakakis-vldb2025.

1 INTRODUCTION

Computing joins efficiently has been a fundamental challenge in query processing since the inception of the relational model. Unfortunately, consistently finding a good join order remains very difficult. This is especially true for the increasingly common queries with up to a thousand relations featuring many-to-many joins [7, 23, 28]. At the same time, a poorly chosen join order will bring even state-of-the-art systems to their knees [26]. In recent work [5], 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 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, 11] and Yannakakis' seminal algorithm (YA) for optimally processing acyclic queries [39], over various notions of query width and query decompositions [13], to the more recent worst-case-optimal (WCO) [29, 30, 37] and factorized [19, 31] 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, 12, 20, 24, 27, 32] 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 [5, 26]. 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 [5]. 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

Proceedings of the VLDB Endowment, Vol. 18, No. 8 ISSN 2150-8097.
doi:10.14778/3742728.3742737

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 [27], 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 [25] 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 *uniform* (physical) data model. In column stores, this data model is simply a relation, implemented as set of column segments. While BKN state that an L&E plan is also meant to produce a relation, they also impose several constraints. For instance, after performing $R \ominus S$, one cannot access the non-join attributes of S without first applying an expand operation. This suggests that the output of $\ominus$ is not a standard relation, making it unclear what exactly the (physical) data model is one should implement for L&E plans.

The second challenge, particular to column stores, is that the common wisdom in column stores is to let operators process *a column-at-a-time*. This is empirically a (large) constant factor faster than element at-a-time processing since it allows using vectorization when applicable as well as amortize function-call overhead. Yet, the code generated for L&E plans by BKN proceeds tuple-at-a-time.

In this paper, we build further upon the ideas in BKN by investigating the implementation of L&E-based query processing inside the more common *interpreted* query engines, in particular column stores. We address the challenges above and obtain an evaluation algorithm for acyclic joins, named Shredded Yannakakis (SYA), that is *provably robust without regret* for a subclass of queries. We summarize our contributions next and highlight the differences with as well as improvements over BKN.

(1) In contrast to BKN who describe lookup and expand in terms of their effect on some intermediate state during execution, we provide a formal semantics to L&E plans based on the nested relational model [6, 36] which is an extension of the relational model where individual records may themselves contain entire relations. In particular, we design a set of nested relational operators that we call the *Nested Semijoin Algebra* (NSA). Here, lookup can be expressed as a form of nesting while expand is a form of unnesting. By formalizing these operations algebraically, we explicitly define the logical data model, allowing us to extend beyond lookup/expand and joins, and generalize to all standard relational operators.

(2) We use NSA to implement L&E plans inside conventional *interpreted* query engines, in particular column stores. Our implementation is based on *query shredding* techniques for simulating nested relational algebra with standard relational algebra [8, 9, 33, 38]. We take special care to provide an efficient column-oriented implementation for completely unnesting deeply nested relations.

(3) BKN observe that L&E plans consisting of two distinct phases—where the first phase exclusively performs Lookups and the second phase exclusively performs Expands—execute in time $O(\text{IN} + \text{OUT})$ for all inputs. In other words, such 2-phase plans are *instance-optimal*. We extend this result to include all NSA operators, not only L&E, by defining 2-phase NSA expressions as those in which no 'shrinking' operator is applied after an unnest (i.e., expand) operation has been performed. We show that a join query can be evaluated by means of a 2-phase NSA join plan if and only if it is acyclic. This result, therefore, generalizes the instance-optimality of YA to NSA plans and provides an additional characterization for the class of acyclic joins.

(4) The aforementioned formal guarantees focus on asymptotic complexity, which often overlooks crucial constant factors. To address this, we perform a finer-grained analysis of NSA plans in terms of a cost model that takes such constant factors into account (more specifically, the cost of building and probing hash maps as well as generating single column vectors). We identify a class of traditional binary join plans—referred to as *well-behaved*—that can be transformed into equivalent 2-phase NSA plans that are guaranteed to *always* have a cost that is no worse than the binary join plan. For binary join plans that are not well-behaved, we offer a heuristic to transform them into equivalent well-behaved plans, while minimizing additional cost.

Shredded Yannakakis (SYA) refers to the algorithm that takes a binary join plan as input, transforms it into a well-behaved plan if needed, and then evaluates the resulting 2-phase NSA plan using shredding. Importantly, SYA can be seamlessly integrated with an existing query optimizer that generates traditional binary join plans, providing a provably robust solution that *consistently* avoids the diamond problem. Additionally, SYA is guaranteed to be robust *without regret* on the class of well-behaved binary join plans.

In comparison, while BKN observe that 2-phase L&E plans can achieve instance-optimality, they adopt a cost-optimisation-based approach to generating L&E plans that does not require, nor guarantee these plans to be 2-phase. As a result, the generated plans are not guaranteed to be instance-optimal. Thus, as with binary join algorithms, the robustness of the system still depends on the quality of the cost estimation and the optimizer. In contrast, the rewriting we propose in this paper is always provably robust, and without regret on a clear subclass.

(5) We implement SYA inside Apache Datafusion [20], a high-performance main-memory-based columnar interpreted query engine written in Rust. Our experimental set-up comprises multiple established benchmarks and includes 1,849 queries evaluated over real-world data. We show that the performance of SYA is *always* competitive with that of binary join plans, and often much better—improving performance for 85.3% of the queries with speedups up to 62.5x—while at the same time guaranteeing robustness. Additionally, our implementation of SYA shows relative speedups that are comparable, and sometimes exceed, those observed by BKN in their compiled query engine. The idea of L&E decomposition can hence be successfully formulated and implemented in interpreted query engines, in particular column stores.

In summary, we show how to process acyclic joins instance-optimally and without regret. We hope that this perspective can help system engineers to better understand YA, and pave the way for its adoption into existing systems.

This paper is organized as follows. We introduce background in Section 2, NSA in Section 3, and shredding in Section 4. We discuss asymptotic complexity and instance-optimality of 2-phase NSA in Section 5, and cost-based complexity and SYA in Section 6.

We discuss experiments in Section 7, and conclude in Section 8. Related work is discussed throughout the paper. Full proofs of formal statements may be found in the extended paper version [4].

2 PRELIMINARIES AND BACKGROUND

For a natural number $n > 0$ we denote the set $\{1, \ldots, n\}$ by $[n]$. We are concerned with the evaluation of *natural join queries*, a.k.a. full conjunctive queries, which are queries of the form:

$$Q = R_1(\overline{x}_1) \bowtie \cdots \bowtie R_k(\overline{x}_k). \tag{1}$$

Here, $k \geq 1$; each R_i is a relation symbol; and each $\overline{x}_i$ is a tuple of pairwise distinct attributes that denotes the schema of R_i, for $i \in [k]$. Expressions of the form $R_i(\overline{x}_i)$ are called *atoms*.

Following the SQL-standard, we adopt bag semantics for join queries. Each input relation $R_i(\overline{x}_i)$ is assumed to be a bag (i.e., multiset) of input tuples over $\overline{x}_i$, and Q computes a bag of tuples over $\overline{x}_1 \cup \cdots \cup \overline{x}_k$. Tuple t occurs in the result of Q if for every $i \in [k]$ the tuple $t[\overline{x}_i]$ (i.e., t projected on $\overline{x}_i$), occurs with multiplicity $m_i > 0$ in input relation R_i. The result multiplicity of t is then $m_1 \times \cdots \times m_k$. In what follows, we use doubly curly braces $\{\!\{\ldots\}\!\}$ to denote bags as well as bag comprehension and denote by $\mathit{supp}(M)$ the *set*, without duplicates, of all elements present in a bag M.

Example 2.1. We use $Q_3 = R(x, y) \bowtie S(y, z) \bowtie T(z, u)$ as an example query throughout the paper. The query is over binary relations and can be seen to compute graph paths of length three.

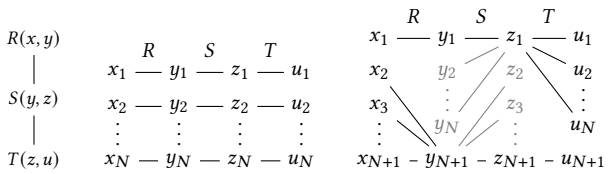
Binary Join Plans. The standard approach to processing a join query Q is to compute one binary join at a time. A *binary plan* (also known as a binary join order) is a rooted binary tree where each internal node is a join operator $\bowtie$ and each leaf node is one of the atoms $R_i(\overline{x}_i)$ of the query. To be correct under bag semantics, it is required that each atom occurs exactly as many times in the plan as it occurs in Q. We will only consider such valid plans in what follows. A binary plan is *left-deep* if the right child of every join node is a leaf; it is *right-deep* if the left child of every join node is a leaf; and it is *bushy* otherwise.

We interpret binary plans as physical query plans where all the joins are evaluated by means of hash-joins. We focus on hash-joins as they are the most common type of joins in database systems. Concretely, every join node in a binary plan indicates a hash-join where the left child is the probe side and the right child is the build side. Leaf nodes indicate input relations.

Example 2.2. Consider the binary plan $P = (R \bowtie S) \bowtie T$ for Q_3. Figure 1 illustrates two input databases. In database db_1, every relation has N tuples and every tuple joins with exactly one tuple of the other relations. On this database Q_3 hence returns N output tuples. Processing Q_3 on db_1 by means of left-deep plan P involves building a hash table on S and T; $|R|$ probes of R-tuples in the hash table on S; and $|R \bowtie S| = N$ probes into the hash table on T, hence doing $O(N)$ work in total, which is optimal.

The second database db_2 has $N + 1$ tuples in R and T, and $2N$ tuples in S. Specifically, the relations are defined as follows:

$$R = \{(x_1, y_1)\} \cup \{(x_{i+1}, y_{N+1}) \mid i \in [N]\}$$
$$S = \{(y_i, z_1) \mid i \in [N]\} \cup \{(y_{N+1}, z_{i+1}) \mid i \in [N]\}$$
$$T = \{(z_1, u_i) \mid i \in [N]\} \cup \{(z_{N+1}, u_{N+1})\}$$



(a) J_3 **(b) Database** db_1 **(c) Database** db_2

Figure 1: Join tree J_3 for the three-path query Q_3, and two input databases. Tuples in db_2 not contributing to the final output are in gray.

While there are only $2N$ output tuples to be produced, plan P is $\Omega(N^2)$ since it will do at least $|R \bowtie S| = N^2 + 1$ probes into T. It hence wastes time computing tuples in $R \bowtie S$ which in the end do not contribute to the output.

While we may be tempted to think that we were just unlucky in choosing an suboptimal binary plan to process db_2 in the previous example, this is not the case: it is straightforward to verify that *any* binary join plan for Q_3 will produce a quadratic subresult. As such, binary join plans are highly effective on certain inputs but cannot efficiently process joins on *all* inputs, even if the query is acyclic—a concept that we introduce next.

Acyclicity and Yannakakis' Algorithm. A join query Q is *acyclic* if it admits a join tree [3, 11]. A *join tree for* Q is a rooted undirected tree J in which each node is an atom of Q. To be correct under bag semantics, it is required that each atom in Q appears exactly as as many times in J as it does in Q. Join trees are required to satisfy the *connectedness property*: for every attribute x, all the nodes containing x form a connected subtree of J. To illustrate, Figure 1a shows a join tree for Q_3.

Checking whether a query is acyclic and constructing a join tree if it exists can be done in linear time w.r.t. the size of the query by means of the GYO algorithm [15, 35, 40]. A seminal result by Yannakakis [39] states that acyclic join queries can be processed *instance-optimally* under data complexity, i.e., in time that is asymptotically linear in the size of the input plus the output. Yannakakis' Algorithm (YA) does so by fixing a join tree and computing in three passes. Define the *semijoin* $R \ltimes S$ of bag R by S to be the bag containing all R-tuples for which a joining tuple in S exists. If a tuple t appears in $R \ltimes S$ it has the same multiplicity as in R.

1. The first pass operates bottom-up over the join tree. For the leaves there is nothing to do. When we reach an internal node R with children $S_1, \ldots, S_k$ YA will replace R by the semijoin of R and all of its children, i.e., we set $R := (\ldots((R \ltimes S_1) \ltimes S_2) \ldots S_k)$.
2. The second pass operates top-down over the join tree. There is nothing to do for the root. For all other nodes R with parent P, R is replaced by the semijoin of R and its parent, $R := R \ltimes P$.
3. The final pass uses standard binary joins to join the relations resulting from the second pass. While YA is typically described to again work bottom-up over the join tree, any binary join plan P for Q that avoids needless Cartesian products[1] can be used in this step.

[1] Meaning that if in a subplan $P' = P_1 \bowtie P_2$ of P no attributes are shared between P_1 and P_2, then the same must hold for all ancestors of P'.

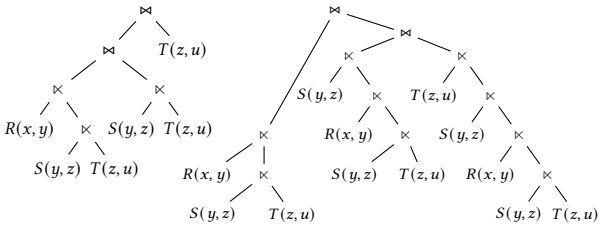


Figure 2: Semijoin plans induced by YA on join tree J_3(Fig.1a). Left: pass two and three combined. Right: all three passes.

The first two passes are known as a *full semijoin reduction* and remove so-called *dangling tuples* from the input: input tuples that cannot be joined to form a complete join result. Once dangling tuples are removed, standard binary joins can be used to compute the actual join result. At that point any intermediate result tuple produced is guaranteed to participate in at least one output tuple.

Example 2.3. Reconsider Q_3 and the input database db_2 from Example 2.2. Assume we execute YA using the join tree J_3 for Q_3 shown in Figure 1a. The bottom-up pass of the algorithm first replaces S with $S \ltimes T$, removing tuples over $z_2, \ldots, z_N$ from S. Note that this retains the full range of y-values in S. Next, R is semijoined with the reduced S. However this does not remove any tuples from R, since all y-values in R are also in S. The top-down pass then replaces S with $S \ltimes R$, removing tuples over $y_2, \ldots, y_N$. The final semijoin $T \ltimes S$ again removes nothing. At this point, all gray-colored tuples in Figure 1c are removed, leaving only the black-colored tuples. On this reduced database, any binary join plan without Cartesian product runs instance-optimally. Note that the removal of dangling tuples is essential, as we know from Example 2.2 and the subsequent discussion that on the original input db_2 every binary join plan will require $\Omega(N^2)$ time.

A straightforward way to implement YA in a database engine is to record the sequence of joins and semijoins that YA does in a physical query plan [14]. These kinds of query plans, which we will refer to as *semijoin plans*, are binary join plans where leaf nodes are replaced by trees that compute semijoins on input relations. For example, the right of Figure 2 shows a semijoin plan for Q_3, corresponding to executing YA using the join tree J_3 of Figure 1a and using the left-deep join order $(R \bowtie S) \bowtie T$ in the last phase.

Unfortunately, this straightforward implementation of YA creates significant overhead when the input database contains no, or only few dangling tuples. Indeed, for Q_3 observe that every relation now participates in at least one join and at least one semijoin, while some relations, like S, participate in five semijoins. Semijoins are also executed by means of hashing and therefore also incur build and probing costs even if they do not remove any tuples in the concrete input database that we execute on. This commonly happens: BKN note that on the Join Order Benchmark [22], this way of implementing YA by adding full semijoin reductions yields a 5-fold slowdown compared to binary join plans.

One way to overcome this limitation is to adopt a cost-based approach and selectively add semijoin operators only when they are deemed useful [34]. However, this no longer guarantees instance-optimality. Another possibility, which preserves instance-optimality, is to observe that instead of doing the full three passes of classical YA, the second and third pass can actually be combined [2, 17, 18]. It then suffices to do only the first pass of semijoin-reductions. This modification of YA leads to somewhat simpler plans as illustrated in the left of Figure 2 for our running example Q_3 and join tree J_3. Note, however, that while this reduces the overhead, it does not completely eliminate it since T and S continue to participate in multiple (semi)joins. Recent so-called *enumeration-based* join evaluation algorithms go one step further: they compute only the semijoin $R \ltimes (S \ltimes T)$ and reuse the hash tables created during the semijoin to *enumerate* the join result $R \bowtie S \bowtie T$ using a specialized algorithm [2, 17, 18]. While such enumeration algorithms have previously been difficult to cast as operators in a physical query plan algebra, and have to date been limited to specialized research prototypes, L&E/NSA plans will provide exactly this functionality.

In conclusion. Binary join plans suffer from the diamond problem. By contrast, semijoin plans induced by running YA (in full, or with the latter two phases combined) are instance-optimal and hence avoid the diamond problem, but on common inputs they may suffer from a constant-factor slowdown compared to binary join plans. Our objective in this paper, therefore, is to engineer the instance-optimality of YA in a database engine without regret.

3 NESTED SEMIJOIN ALGEBRA

In this section, we provide a formal syntax and semantics for L&E plans, including how they interact with other relational algebra (RA) operators, in terms of a set of nested relational operators that we call the *Nested Semijoin Algebra* (NSA). Having specified the data model and nested operators required to support L&E plans, we subsequently use this formalisation in Section 4 to derive an implementation strategy of L&E plans in interpreted query engines.

The nested relational model is an extension of the standard relational model. In a *nested relation*, a tuple may consist not only of scalar data values but also of entire relations in turn. The *nested relational algebra* (NRA) for querying nested relations is obtained by generalizing the operators of relational algebra (selection, projection, join, …) to work on nested relations, and by adding two extra operators: *nesting* and *unnesting* [36]. Many variants of the nested relational model have been proposed, including extensions that allow for mixed collection types such as sets, bags, lists, arrays [6] as well as dictionaries [10]. In this paper, we consider a variant where each (nested) relation is bag-based, and where we also have dictionaries. To make the connection with L&E plans, we depart from the standard set of operators of NRA, and instead introduce a set of operators that we call the Nested Semijoin Algebra (NSA).

Schemes and Nested Relations. Just like flat relations have flat schemes, nested relations have nested schemes. We refer to the attributes that appear in the scheme of classical flat relations as *flat* attributes. Let $\mathcal{A}$ denote the set of all flat attributes. A *(nested) scheme* is a finite set X, like $\{x, \{y\}, \{u, \{v\}\}\}$, that consists of flat attributes (x in this case) and other schemes (i.e., $\{y\}$ and $\{u, \{v\}\}$). No flat attribute is allowed to occur more than once, so $\{x, \{y\}, \{u, \{x\}\}\}$ is not a valid scheme. Schemes are also called *nested* attributes. We range over flat attributes by lowercase letters ($x, y, \ldots$); over schemes by uppercase letters ($X, Y, \ldots$), both from the end of the alphabet; and over finite sets of flat attributes by $\overline{x}$. We write $\mathcal{A}(X)$

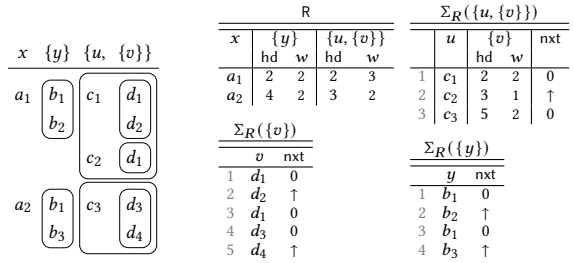


Figure 3: (left) A nested relation R. (right) Its shredded representation $\mathcal{R}$. The gray numbers indicate tuple offsets; nxt points to the next tuple (via ↑) or is 0 when there is none.

for the set of all flat attributes occurring somewhere in X (either directly or in some inner nested scheme); and $sub(X)$ for the set of all schemes occurring in X (again, either directly or in some inner nested scheme). So for $X = \{x, \{y\}, \{u, \{v\}\}\}$, $\mathcal{A}(X) = \{x, y, u, v\}$ and $sub(X) = \{\{y\}, \{u, \{v\}\}, \{v\}\}$.

Fix a scheme X. A *relation* over a X is a finite bag of tuples over X. Here, a *tuple over* X is a mapping t on X such that $t(x)$ is a scalar data value (of appropriate type) for each flat attribute $x \in X$, and $t(Y)$ is a non-empty relation over Y for each nested attribute $Y \in X$. Note that if X is *flat*, i.e., if $X \subseteq \mathcal{A}$, then this definition of a relation over X coincides with the usual one. We call R a *flat relation* in that case. We restrict inner nested relations to be non-empty as in this paper we always start from flat relations and the operators that we consider will never introduce empty inner nested relations. We write $R\colon X$ and $t\colon X$ to denote that R is a relation (resp. t is a tuple) over scheme X. We write $|R|$ denote the *cardinality* of R, i.e., the total number of tuples in R. Note that $|R|$ only refers to the number of tuples in the outer-most bag of R, and does not say anything about the cardinality of the inner-nested relations appearing in those tuples. Figure 3 shows a nested relation with cardinality 2 and scheme $\{x, \{y\}, \{u, \{v\}\}\}$.

We adopt the following notation on tuples. If $s\colon X$ and $t\colon Y$ are tuples over disjoint schemes then $s \circ t$ denotes their concatenation, which is a tuple over $X \cup Y$. Furthermore, if $Z \subseteq X$ then $t[Z]$ denotes the restriction (i.e., projection) of mapping t to the attributes in Z.

Dictionaries. A *dictionary scheme* is an expression of the form $\overline{y} \rightsquigarrow Z$ such that no flat attribute in $\overline{y}$ occurs anywhere in Z. A *dictionary over* $\overline{y} \rightsquigarrow Z$ is a finite mapping D that maps $\overline{y}$-tuples to non-empty relations over Z. The tuples in the domain $dom(D)$ of D are called the *keys* of D. The *cardinality* of D, denoted $|D|$ is the number of keys. We write $D\colon \overline{y} \rightsquigarrow Z$ to indicate that D is a dictionary over $\overline{y} \rightsquigarrow Z$. Conceptually, a dictionary is a special kind of nested relation with scheme $\overline{y} \cup \{Z\}$; in contrast to a nested relation it also allows to lookup keys.

NSA. Our Nested Semijoin Algebra (NSA) consists of the standard relational operators filter (σ), projection (π), renaming (ρ), bag-union ($\cup$), bag difference ($-$)—all straightforwardly extended to operate on nested relations—and four new operators: group-by (γ), nested semijoin ($\triangleright\!\!\!\triangleright$), unnest ($\mu$), and flatten ($\mu^*$). We can think of γ, $\triangleright\!\!\!\triangleright$, and μ as corresponding to the three separate phases of a traditional hash-based join: hash-table building, probing, and output construction, respectively. We define these additional operators next and provide examples in Figure 4.

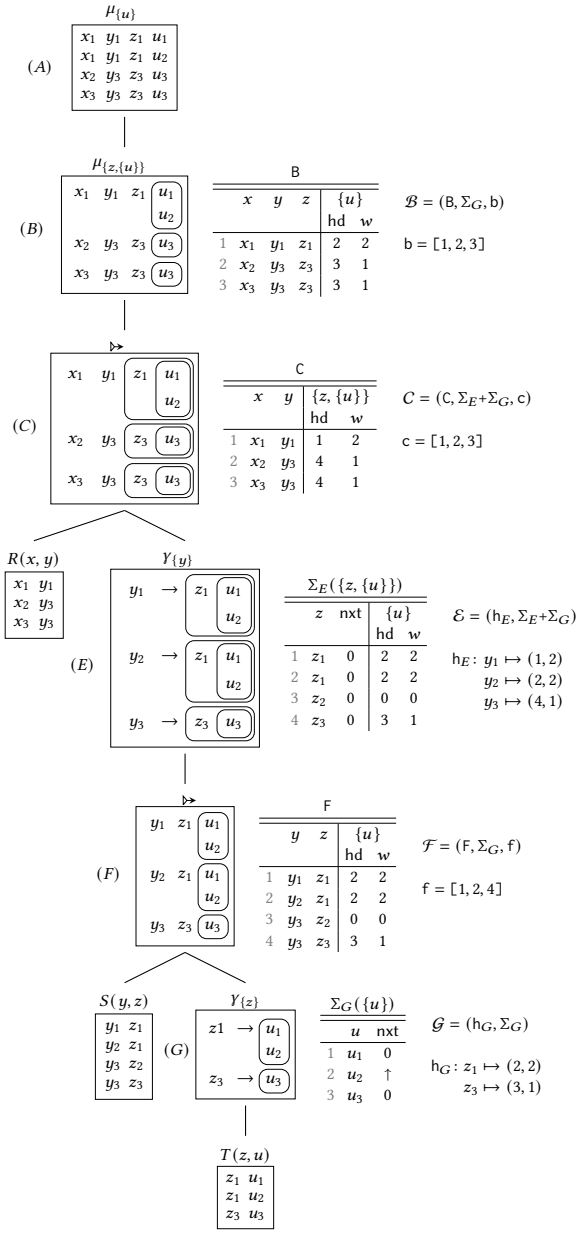


Figure 4: Example evaluation of an NSA expression. Intermediate nested relations and dictionaries are labeled $(A), (B), \ldots$ Shredded processing is illustrated on the right.

The *group-by* operator $\gamma_{\overline{y}}$ when applied to a relation $R\colon X$ creates a dictionary by grouping the tuples in R on the attributes in $\overline{y}$, and mapping each group-key to its group projected on $Z = X \setminus \overline{y}$. Formally, the result dictionary $D\colon \overline{y} \rightsquigarrow Z$ has $supp(\pi_{\overline{y}}(R))$ as keys, and maps each key $t \mapsto \pi_Z(\sigma_{\overline{y}=t}(R))$. As an example, in Figure 4, $\gamma_{\{z\}}(T)$ is shown as G, and $\gamma_{\{y\}}(F)$ as E.

The *nested semijoin* operator $\triangleright\!\!\!\triangleright$ takes two arguments, a relation $R\colon X$ and a dictionary $D\colon \overline{y} \rightsquigarrow Z$. It is required that X is *compatible* with $\overline{y} \rightsquigarrow Z$, meaning that (i) $\overline{y} \subseteq X$ and (ii) $\mathcal{A}(Z) \cap \mathcal{A}(X) = \emptyset$, implying that the union $X \cup \{Z\}$ is again a scheme. Compatibility is

$$\frac{}{R\colon \overline{x}_R} \qquad \frac{e\colon X \quad \overline{y} \subseteq X}{\sigma_{\theta(\overline{y})}(e)\colon X} \qquad \frac{e\colon X \quad Y \subseteq X}{\pi_Y(e)\colon Y} \qquad \frac{e\colon X}{\rho_\varphi(e)\colon \varphi(X)}$$

$$\frac{e_1\colon X \quad e_2\colon X}{e_1 \cup e_2\colon X} \qquad \frac{e_1\colon \overline{x} \quad e_2\colon \overline{x}}{e_1 - e_2\colon \overline{x}} \qquad \frac{e\colon X \quad \overline{y} \subseteq X \quad Z = X \setminus \overline{y}}{\gamma_{\overline{y}}(e)\colon \overline{y} \rightsquigarrow Z}$$

$$\frac{e_1\colon X \quad e_2\colon \overline{y} \rightsquigarrow Z \quad X \sim \overline{y} \rightsquigarrow Z}{e_1 \triangleright\!\!\rightarrow e_2\colon X \cup \{Z\}} \qquad \frac{e\colon X \quad Y \in X \setminus \mathcal{A}}{\mu_Y(e)\colon X \setminus \{Y\} \cup Y} \qquad \frac{e\colon X}{\mu^*(e)\colon \mathcal{A}(X)}$$

Figure 5: NSA type rules.

denoted $X \sim \overline{y} \rightsquigarrow Z$. The nested semijoin operator $R \triangleright\!\!\rightarrow D$ probes D for each tuple t in R; if D contains $t[\overline{y}]$, then it extends t by a single nested attribute, Z, which contains the entire relation associated to $t[\overline{y}]$ by D,

$$R \triangleright\!\!\rightarrow D \stackrel{\text{def}}{=} \{\!\{t \circ \{Z \mapsto D(t[\overline{y}])\} \mid t \in R, t[\overline{y}] \in dom(D)\}\!\}. \quad (2)$$

Figure 4 depicts the result of $S \triangleright\!\!\rightarrow G$ as F, and that of $R \triangleright\!\!\rightarrow E$ as C.

The unnest operator $\mu_Y(R)$ unnests a nested attribute $Y \in X$ from input relation $R\colon X$ and has semantics

$$\mu_Y(R) \stackrel{\text{def}}{=} \{\!\{s[X \setminus \{Y\}] \circ t \mid s \in R, t \in s(Y)\}\!\}. \quad (3)$$

It hence pairs each tuple $s \in R$ with all tuples in the relation $s(Y)$. Figure 4 shows the result of $\mu_{\{u\}}(B)$ as A, and that of $\mu_{\{z,\{u\}\}}(C)$ as B.

Finally, the flatten operator $\mu^*(R)$ completely flattens a nested relation $R\colon X$, returning a flat relation with scheme $\mathcal{A}(X)$. Specifically, if $Y_1, \ldots, Y_k$ is an enumeration of $sub(X)$ such that schemes occur before their subschemes (i.e., for all i, j, if $Y_i \in Y_j$ then $j < i$), then $\mu^*(R) \stackrel{\text{def}}{=} \mu_{Y_1} \ldots \mu_{Y_k}(R)$. For example, if $R\colon \{x, y, \{z\{u\}\}$ then $\mu^*(R) = \mu_{\{u\}}(\mu_{\{z,\{u\}\}}(R))$. While μ^* is hence already expressible in NSA through repeated unnests, we add μ^* as a primitive operator to NSA for reasons that will become clear in Section 4.

Like standard relational algebra, the expressions in NSA must be well-typed. Figure 5 shows the NSA typing rules, where we write $e\colon X$ to denote that NRA expression e is well-typed and has output scheme X. There, R ranges over *flat* input relation symbols, for which we assume to have an associated input scheme $\overline{x}_R$. For the selection operator, $\theta(\overline{y})$ ranges over selection predicates that concern the values in attributes in $\overline{y}$. For the renaming operator ρ, the subscript φ denotes a permutation of $\mathcal{A}$ and we denote by $\varphi(X)$ the result of applying such a permutation recursively to scheme X.

Complexity. For the complexity results that follow, it is important to emphasize that the NSA type rules (i) restrict to flat input relations, and (ii) restrict all operators that involve checking tuple-equality, like filter, difference, group-by, and nested semijoin, to check equality on *flat* tuples only. Indeed, recall that by convention $\overline{x}$ denotes a flat scheme. Then, the type rule for group-by, for example, indicates that only flat tuples can be group-by keys. The reason for this restriction is that tuples over a flat scheme have a size that is constant in data complexity, whereas nested tuples can have arbitrary size. Hence checking equality over flat tuples is constant time, whereas it may be linear for nested tuples. We adopt the same restriction to selection predicates θ in a selection $\sigma_{\theta(\overline{y})}(R)$: only predicates $\theta(\overline{y})$ for which we can check in constant time (in data complexity, in the RAM model of computation) that a tuple $t \in R$ satisfies θ are allowed.

Relating NSA to other operators. Standard relational algebra operators such as join and flat semijoin, as well as the lookup ($\ominus$) and expand ($\circledcirc$) operators of BKN and the nesting operator (ν) of standard nested relational algebra [36][2] are cleanly expressible in NSA as a composition of NSA operators. For example,

$$R(x,y) \bowtie S(y,z) \equiv \mu_{\{z\}}(R \triangleright\!\!\rightarrow \gamma_{\{y\}}(S)) \quad (4)$$

$$R(x,y) \ltimes S(y,z) \equiv \pi_{x,y}(R \triangleright\!\!\rightarrow \gamma_{\{y\}}(S)) \quad (5)$$

$$R(x,y) \ominus S(y,z) \equiv R \triangleright\!\!\rightarrow \gamma_{\{y\}}(S) \quad (6)$$

$$\circledcirc(R(x,y) \ominus S(y,z)) \equiv \mu_{\{z\}}(R \triangleright\!\!\rightarrow \gamma_{\{y\}}(S)) \quad (7)$$

$$\nu_{x,y}\, R(x,y,u,v) \equiv \pi_{x,y}(R) \triangleright\!\!\rightarrow \gamma_{\{x,y\}}(R) \quad (8)$$

Actually, we can take the right-hand sides in the above expressions as the *definition* in NSA of the operators on the left-hand side. As such, this provides a formalisation of L&E plans in terms of NSA.

The advantage of our algebraic approach is that it clearly defines the underlying data model and allows free operator composition.

4 SHREDDED PROCESSING

We next turn our attention to the efficient processing of NSA, focusing on its implementation in main memory column stores. NSA is a form of Nested Relational Algebra (NRA), and it is well-known that one can evaluate NRA using standard flat relational algebra operators by representing a nested relation as a collection of flat relations, and simulating nested relational operators by flat relational operators on this representation [8, 9, 33, 38]. We adapt this technique, known as *query shredding*, to implement NSA. We differ from traditional shredding in that some nested operators, in particular μ, are implemented by means of a *join* of flat relations. In our setting, however, we want to use NSA as a description of physical query plans where γ, $\triangleright\!\!\rightarrow$ and μ correspond to the three phases of a traditional hash join: build, probe, and construct. Specifically, μ must then be limited to constructing the output tuples *when the set of matching tuples have already previously been identified* by an earlier $\triangleright\!\!\rightarrow$ operator; its shredded implementation hence should not require further joins. To obtain this behavior we modify the traditional shredded representation of a nested relation: each nested attribute Y will be encoded by a flat attribute that holds an iterator over the elements of Y, instead of an abstract identifier as is traditionally done. Additionally, to support efficient flatten (μ^*), we also store the *weight* of every Y, which is the total number of tuples produced when flattening Y. A benefit of the shredding approach is that it only requires modest change to existing query engines to implement: for many NSA operators we can simply delegate to the implementation of existing relational algebra operators; only γ, $\triangleright\!\!\rightarrow$, μ, and μ^* require separate treatment.

To simplify notation in the discussion that follows, we restrict our attention in this section to the shredded processing of nested relations $R\colon X$ for which $\emptyset$ does not occur multiple times in X. So, $X = \{x, \{y, \emptyset\}\}$ is allowed but $X = \{x, \emptyset, \{y, \emptyset\}\}$ is not. We silently assume throughout this section that all considered NSA operators consume and produce nested relations satisfying this criterion. Our implementation does not have this restriction.

We begin by describing how to represent nested relations in Section 4.1; evaluation algorithms are discussed in Section 4.2.

[2] But restricted to using flat tuples as nesting keys.

4.1 The shredding representation

Columnar layout. We assume that we are working in main memory, and that a flat relation $R(x_1, \ldots, x_n)$ is physically represented as a tuple $\mathsf{R} = (\mathsf{R}.x_1, \ldots, \mathsf{R}.x_n)$ of vectors $\mathsf{R}.x_i$, all of length $|R|$. It is understood that values at the same offset in these vectors encode a complete tuple, i.e., $R = \{\!\{(\mathsf{R}.x_1[i], \ldots, \mathsf{R}.x_n[i]) \mid 1 \le i \le |R|\}\!\}$. In particular, it is possible to refer to tuples positionally, i.e., the 1st tuple in R, the second tuple in R, and so on.[3] We will refer to R as a physical relation, and denote the number of tuples in R by $|\mathsf{R}|$. If $1 \le i \le |\mathsf{R}|$ and $\overline{y} = y_1, \ldots, y_k$ is a subset of $\{x_1, \ldots, x_n\}$, then we write $\mathsf{R}[i](\overline{y})$ for the tuple $(\mathsf{R}.y_1[i], \ldots, \mathsf{R}.y_k[i])$. We denote the length of a vector v by $|v|$. A *position vector for* R is a vector p of natural numbers, all between 1 and $|\mathsf{R}|$. We assume an operation $\mathtt{take}(\mathsf{R}.x, p)$ that can be used to construct a new vector from an existing vector x in R and a position vector p on R: $\mathtt{take}(\mathsf{R}.x, p)$ returns a new column c of length $|p|$ such that $c[i] = \mathsf{R}.x[p[i]]$ for all i. The $\mathtt{take}$ operation hence re-arranges the entries of $\mathsf{R}.x$ according to p, possibly repeating some entries and filtering out others. If the entries in p are strictly increasing then p is a *selection vector for* R. Note that this implies $|p| \le |\mathsf{R}|$, and if $|p| = |\mathsf{R}|$ then $p = [1, \ldots, |\mathsf{R}|]$. We denote the latter vector also by $\mathtt{all}_{\mathsf{R}}$. Note that $\mathtt{take}(\mathsf{R}.x, p)$ can only filter entries when p is a selection vector.

Weights. The *weight* of a nested relation R is the total number of tuples produced when flattening R, i.e. $\mathit{weight}(R) = |\mu^*(R)|$. Similarly, the weight of a nested tuple t is $|\mu^*(\{\!\{t\}\!\})|$, the total number of tuples produced when flattening t.

Schemes shredding. For a scheme $X = \{y_1, \ldots, y_k, Z_1, \ldots, Z_\ell\}$ define the *flat* schemes $\mathit{shred}(X)$ and $\mathit{ishred}(X)$ as

$$\mathit{shred}(X) \stackrel{\text{def}}{=} \{y_1, \ldots, y_k, \mathsf{hd}\_Z_1, \ldots, \mathsf{hd}\_Z_\ell, \mathsf{w}\_Z_1, \ldots, \mathsf{w}\_Z_\ell\} \quad (9)$$

and $\mathit{ishred}(X) = \mathit{shred}(X) \cup \{\mathsf{nxt}\}$. Here, the attributes hd_Z_i, w_Z_i, and nxt are fresh flat attributes, all pairwise distinct as well as distinct from the y_j. Intuitively, hd_Z_i will store the head of a linked list that represents the contents of nested attribute Z_i, whereas nxt will be used to point to the next tuple in such lists. w_Z_i will store the weight of the nested Z_i relations. Observe that if X is flat to begin with, then $\mathit{shred}(X) = X$.

Relation shredding. The shredded representation of a nested relation $R\colon X$ is a triple $\mathcal{R} = (\mathsf{R}, \Sigma_R, \mathsf{r})$ where (i) R is a physical relation over $\mathit{shred}(X)$; (ii) Σ_R is a *store* over X: a collection of physical relations, one physical relation $\Sigma_R(Y)$ for every nested attribute $Y \in \mathit{sub}(X)$, such that $\Sigma_R(Y)$ has schema $\mathit{ishred}(Y)$; and (iii) r is a selection vector for R.

The nxt attribute of the tuples in $\Sigma_R(Y)$ is used to encode a linked list of tuples: for all positions $1 \le i \le |\Sigma_R(Y)|$, if $\Sigma_R(Y).\mathsf{nxt}[i] = 0$ then the tuple at position i in $\Sigma_R(Y)$ is the final tuple in the list; otherwise its successor in the list is the tuple at offset $\Sigma_R(Y).\mathsf{nxt}[i]$. Correspondingly, each hd_Y-value of a tuple in R points to the head of the linked list in $\Sigma_R(Y)$ storing the nested tuples.

Example 4.1. To clarify the discussion that follows, we illustrate shredding by means of Figure 3 which shows a nested relation $R\colon X$ with $X = \{x, \{y\}, \{z, \{u\}\}\}$ on the left, and its shredded representation $\mathcal{R}$ on the right. We omit the selection vector. $\mathcal{R}$ contains four flat relations, or "shreds": one for the full scheme $x, \{y\}, \{u, \{v\}\}$, and one each for the nested schemes $\{u, \{v\}\}$, $\{v\}$, and $\{y\}$. Let us focus on the first shred R. The first column stores values of the flat attribute x. The attribute $\{y\}$ has two columns: hd stores pointers (offsets) to the $\Sigma_R(\{y\})$ shred, each indicating the head of the linked list of tuples nested under the corresponding x-value. The w column stores the weight, which is the size of the nested relation if we were to flatten it. In each nested shred, the nxt column link together tuples that belong to the same nested segment. For example, the first tuple in R has $x = a_1$, $\{y\}.\mathsf{hd} = 2$, $\{y\}.w = 2$; this points to the head of the linked list in $\Sigma_R(\{y\})$ at offset 2, whose nxt column points to tuple above it. Similarly, the $\{u, \{v\}\}$ attribute of the first tuple has $\mathsf{hd} = 2$ and $w = 3$, pointing to the 2nd tuple in $\Sigma_R(\{u, \{v\}\})$. However, note that although $w = 3$, the linked list has length 2, becuase the shred over $\Sigma_R(\{u, \{v\}\})$ further nests the shred over $\{v\}$.

The shredded representation of R works as follows: every tuple $t \in R$ is represented by exactly one tuple in R. Let i be the index of the tuple in R representing t. Then $t(x) = \mathsf{R}.x[i]$ for every flat $x \in X$. For every nested attribute $Y \in X$ we have that $\mathsf{R}.\mathsf{w}\_Y[i] = \mathit{weight}(t(Y))$. Furthermore, $\mathsf{R}.\mathsf{hd}\_Y[i] = j$ for some $1 \le j \le |\Sigma_R(Y)|$, which is the head index of the linked list of tuples in $\Sigma_R(Y)$ that together represent the tuples occurring in $t(Y)$. Note that the tuples in $t(Y)$ may themselves contain further nested relations, and the shredding hence proceeds recursively.

Every tuple in R will be represented in the above sense in R. To allow efficient implementation of repeated semijoins of the form $(S \triangleright\!\!\!\succ e_1) \triangleright\!\!\!\succ e_2$, we do allow that in the shredding $\mathcal{R}$ for $R = S \triangleright\!\!\!\succ e_2$, the physical relation R contains tuples that have already been filtered out by the nested semijoin, i.e., we allow that $|\mathsf{R}| \ge |R|$. In that case, the selection vector r of R contains the offsets of the *valid* tuples in R, i.e., those that actually represent tuples in R (having passed previous $\triangleright\!\!\!\succ$). So, we always have $|\mathsf{r}|$ equal to the cardinality of R. Additionally, if X is a flat scheme, then R is not allowed to contain redundant tuples, i.e., $|\mathsf{R}| = |\mathsf{r}|$, and $\mathsf{r} = \mathtt{all}_{\mathsf{R}}$. It is important to observe that if X is a flat scheme, then Σ_R is empty; the shredding $\mathcal{R}$ of R is then simply $\mathcal{R} = (\mathsf{R}, \emptyset, \mathtt{all}_{\mathsf{R}})$.

Dictionary shredding. The shredded representation of a dictionary $D\colon \overline{y} \rightsquigarrow Z$ is similarly defined as the shredding of a nested relation, except that it has a hash-map as first component and does not have a selection vector. Concretely, the shredding of D is a pair $\mathcal{D} = (\mathsf{h}, \Sigma_D)$ where h is a hash-map, mapping $\overline{y}$-tuples to pairs (j, w), and Σ_D is a store over $\{Z\}$. For every $\overline{y}$-tuple t, if $\mathsf{h}(t) = (j, w)$ then j is the head index in $\Sigma_D(Z)$ of the linked list of tuples that together represent the nested relation $D(t)$, and $w = \mathit{weight}(D(t))$. See node (G) in Figure 4 for an example.

4.2 Processing

We implement NSA by defining a physical operator f for every NSA operator f. Physical operators consume and produce shredded representations: if $\mathcal{R}$ is the shredding of R then $\mathsf{f}(\mathcal{R})$ is the shredding of $f(R)$. For the NSA operators $f \in \{\sigma, \pi, \rho, \cup, -\}$ that also exist in flat RA, the physical operator f simply consists of applying the corresponding flat physical RA operator to one or more physical relations in $\mathcal{R}$, possibly with a slight variation. For example, consider $f = \pi_Y$ and $R\colon X$. To get a representation of $\pi_Y(R)$ from $\mathcal{R} = (\mathsf{R}, \Sigma_R, \mathsf{r})$, it suffices to simply return $(\pi_{\mathit{shred}(Y)}(\mathsf{R}), \Sigma'_R, \mathsf{r})$ where Σ'_R is obtained

[3] Note that we start our offsets at 1, so the first tuple has offset 1.

```
1: def groupby(R,Σ_R,r,X,ȳ,Z):
2:   w = multiply_weights(R, X)
3:   nxt = [0] * |R|
4:   h = {} # maps keys -> (pos, weight)
5:   for i in r :
6:     key = R[i](ȳ)
7:     if h.contains(key) :
8:       (j, prev_w) = h[key]
9:       nxt[i] = j
10:      h[key] = (i, prev_w + w[i])
11:    else
12:      h[key] = (i, w[i])
13:  Σ_R(Z) = new_physical_relation()
14:  Σ_R(Z).nxt = nxt
15:  for u in shred(Z) { Σ_R(Z).u = R.u }
16:  return (h, Σ_R)
```

```
1: def semijoin(R,Σ_R,r,h,Σ_D,X,ȳ,Z):
2:   sel = []
3:   R.hd_Z = [0] * |R|
4:   R.w_Z = [0] * |R|
5:   for i in r :
6:     key = R[i](ȳ)
7:     if h.contains(key) :
8:       sel.append(i)
9:       (R.hd_Z[i], R.w_Z[i]) = h[key]
10:  return (R,Σ_R + Σ_D, sel)
11:
12: # iterator over linked list at row
13: def itr(R,Σ_R,Y,row):
14:   curr = R.hd_Y[row]
15:   while curr != 0 :
16:     yield curr
17:     curr = Σ_R(Y).nxt[curr]
```

```
1: def unnest(R,Σ_R,r,X,Y):
2:   pos_R = []; pos_Y = []
3:   for i in r :
4:     for j in itr(R, Σ_R, Y, i) :
5:       pos_R.append(i)
6:       pos_Y.append(j)
7:   O = new_physical_relation()
8:   for u in shred(X) \ {hd_Y} :
9:     O.u = take(R.u, pos_R)
10:  for u in shred(Y) \ {nxt} :
11:    O.u = take(Σ_R(Y).u, pos_Y)
12:  del(Σ_R, Y)
13:  return (O, Σ_R, all_0)
```

Figure 6: Physical implementation of group-by, nested semijoin, and unnest.

```
1: def flatten(R,r,Σ_R,X):
2:   O = new_physical_relation()
3:   rep = [1] * |r|
4:   rflatten(R,r,rep,Σ_R,X,O)
5:   return (O, ∅, all_0)
6:
7: def rflatten(R,pos,rep,Σ_R,X,O):
8:   w = multiply_weights(R, X)
9:   # Generate output columns for all flat attrs in R
10:  generate (R,pos,rep,X,O)
11:  # Recursively generate columns for nested attrs in R
12:  for Y in X \ A :
13:    npos = []; nrep = []
14:    # w = total weight of all remaining nested attrs
15:    w = div(w, R.w_Y)
16:    for (row, i) in enumerate(pos) :
17:      for k = 1 to rep[row] :
18:        for j in itr(R,Σ_R,Y,i) :
19:          npos.append(j); nrep.append(w[i])
20:        # already update rep for the next Y
21:        rep[row] *= R.weightY[i]
22:    rflatten (Σ_R(Y),npos,nrep,Σ_R,Y, O)
23:
24: def generate(R,pos,rep,w,X,O):
25:   rwpos = []
26:   for (i,r) in zip(pos, rep), j in 1..r*w[i]:
27:     rwpos.append(i)
28:   for u in X ∩ A :
29:     O.u = take(R.u, rwpos)
```

Figure 7: Physical implementation of flatten.

from Σ_R by removing all entries in $sub(X \setminus Y)$. This works because $\pi_{shred(Y)}(\mathsf{R})$ retains only those columns in R required to represent $\pi_Y(R)$. Because the nested attributes in $sub(X \setminus Y)$ are removed from R we can also remove them from Σ_R. The other standard operators in $\{\sigma, \pi, \rho, \cup, -\}$ are similarly implemented by calling flat RA physical operators. Because of space constraints, we omit their definition. The interested reviewer may find them in [4].

The implementation of $\gamma, \mapsto$, and μ is defined in Figure 6 and illustrated on an example in Figure 4. Group-by $\gamma_{\overline{y}}(R)$ with $R\colon X$ follows conventional hash-build. Given shredding $\mathcal{R} = (\mathsf{R}, \Sigma_R, \mathsf{r})$ of nested relation R, as well as schemes $\overline{y}$ and $Z = X \setminus \overline{y}$, we compute the weight of each tuple in R, using a function `multiply_weights` (definition not shown) that returns a vector w with $w[i]$ equal to the product over all nested attributes $Y \in X$ of $\mathsf{R}.\mathsf{w}\_Y[i]$. Then vector nxt of length |R| is created, initialized to 0 so that initially all tuples terminate the linked lists encoded in nxt. We further initialize h to the empty hash-map. We then iterate over the group-by keys mentioned in r, adding them to h, and storing the position of the most recent R-tuple with the current key as well as the total weight of the key. If we have previously already encountered the same key, the current tuple's nxt value is set to point to the position in R of the previous tuple with the same key, and the weight is updated. Finally, we create the store entry for Z by selecting the columns in Z from R, and adding nxt.

The implementation of nested semijoin $R \mapsto D$ takes as argument the shredding $\mathcal{R} = (\mathsf{R}, \Sigma_R, \mathsf{r})$ of nested relation $R\colon X$ and the shredding $\mathcal{D} = (\mathsf{h}, \Sigma_D)$ of dictionary $D\colon \overline{y} \rightsquigarrow Z$, as well as the schemes X, $\overline{y}$, and Z. It simply executes as a conventional hash join probe. We collect in selection vector sel the positions of the valid tuples in R whose keys can be successfully probed in D. We add new vectors hd_Z and w_Z to R, in which we store the matching positions in $\Sigma_D(Z)$ according to h, as well as their weights. In line 10, Σ_R+Σ_D denotes the disjoint union of the two stores Σ_R and Σ_D.[4]

Unnesting $\mu_Y(R)$ takes as argument the shredding $(\mathsf{R}, \Sigma_R, \mathsf{r})$ of R as well as Y. It first creates two position vectors, pos_R and pos_Y that are populated with valid positions in R and $\Sigma_R(Y)$, respectively. Specifically, for every tuple $t \in R$ that is represented at position i in R we append the positions of all the tuples in $\Sigma_R(Y)$ that encode the elements of $t(Y)$ to pos_Y; and we add i to pos_R as many times as $|t(Y)|$. In line 4, `itr(R,Σ_R,Y,i)` returns an iterator over the positions in $\Sigma_R(Y)$ that encode the elements of $t(Y)$. We use the position vectors to index into R resp. $\Sigma_R(Y)$ to create the physical representation O of the output in lines 7–12. Finally, we remove the entry for Y from Σ_R as this is no longer required.

[4]This is disjoint because the type rules for $\mapsto$ require X compatible with $\overline{y} \to Z$. In particular, $X \cup \{Z\}$ is a scheme; therefore the domains of Σ_R and Σ_D must be disjoint.

Flatten. When multiple unnest operations are applied in sequence, there is an overhead in the number of `take` operations applied. To illustrate, consider $\mu_{\{u\}}\,\mu_{\{z,\{u\}\}}(C)$ from Figure 4, where $C\colon \{x, y, \{z, \{u\}\}\}$. The first unnest, $\mu_{\{z,\{u\}\}}(C)$, will already perform a `take` on x, y, and z (among others) to produce the result with scheme $\{x, y, z, \{u\}\}$. The second unnest performs a `take` again on x, y, z to produce the final result. While this overhead is modest in Figure 4, it grows linearly in the number of μ applied sequentially. For $\mu_{Y_1}\,\mu_{Y_2}\ldots\mu_{Y_k}(R)$, the outer-most flat attributes would be copied and re-arranged k times by means of `take` before producing the final, flat relation. It is for this reason that we have included flatten (μ^*) as a primitive operator in NSA, and provide the dedicated physical implementation `flatten` shown in Figure 7. It performs only a single `take` operation per flat attribute.

Flatten is implemented by calling the auxiliary function `rflatten`, which takes a physical relation `R` as argument, a position vector `pos` for `R`, and a numerical vector `rep` of the same length as `pos` containing only strictly positive numbers, called the *repetition vector*.[5] Initially, `pos` is the selection vector of $\mathcal{R}$ and `rep` contains all 1s, but this changes when we call `rflatten` recursively. Intuitively, for each tuple i specified in `pos` and matching repetition number r specified in `rep`, `rflatten` will completely flatten the i-th tuple of `R`, but additionally *repeat* each produced flattened tuple r times. To be precise, let s_i denote the nested tuple represented at offset i in `R`. Let us write $\mu^*(s_i)$ for $\mu^*(\{\!\{s_i\}\!\})$. When we implement $\mu^*(s_i)$, it will produce tuples in a certain order; say it produces the flattened tuples $t_1, \ldots, t_n$. Then, let $\mu^*(s_i, r)$ be this sequence with every t_j repeated r times as follows

$$\mu^*(s_i, r) \stackrel{\text{def}}{=} \underbrace{t_1, \ldots, t_1}_{r \text{ times}}, \ldots, \underbrace{t_n, \ldots, t_n}_{r \text{ times}}. \tag{10}$$

Assuming `pos` $= [i_1, \ldots, i_p]$ and `rep` $= [r_1, \ldots, r_p]$, the call to `rflatten(R, pos, rep)` will produce a physical relation that represents the sequence of tuples $\mu^*(s_{i_1}, r_1), \ldots, \mu^*(s_{i_p}, r_p)$, in this order.

To understand how `rflatten` works consider the flattening of a single tuple s_i having flat attributes $\overline{x}$ and nested attributes $Y_1, \ldots, Y_k$. By definition,

$$\mu^*(s_i) = \{\!\{s_i[\overline{x}]\}\!\} \times \mu^*(s_i(Y_1)) \times \cdots \times \mu^*(s_i(Y_k)).$$

All tuples in $\mu^*(s_i)$ hence have the same $\overline{x}$-values, which is combined with the cartesian product of flattening $Y_1, \ldots, Y_k$. As such, for each $u \in \overline{x}$ we can easily produce the entire u-column of $\mu^*(s_i)$ by taking $s_i(u)$ and repeating this value $r_i * \mathit{weight}(s_i(Y_1)) \times \cdots \times \mathit{weight}(s_i(Y_k))$ times. This is exactly what `rflatten` does in lines 8 and 9 by first calculating the total weight for each tuple, and subsequently calling `generate` to produce each column.

Next, lines 12–22 produce $\mu^*(s_i(Y_1)) \times \cdots \times \mu^*(s_i(Y_k))$ in a column-wise fashion, so that (i) the recursive μ^* calls can independently produce the columns for their respective flat attributes and (ii) these independent calls produce the flattened tuples in an order so that all columns together give a physical representation for the entire cartesian product. For the recursive call on Y_1 this is trivial: flatten each tuple in $s_i(Y_1)$ and repeat it $\mathit{weight}(s_i(Y_2)) \times \cdots \times \mathit{weight}(s_i(Y_k))$ times to account for the cartesian products that follow. For the recursive call on Y_ℓ with $2 \le \ell \le k$ this becomes more involved. Assume that $s_i(Y_\ell)$ contains the tuples represented at offsets $[j_1, \ldots, j_m]$ in $\Sigma_R(Y_\ell)$. Then, letting $r' = r_i \times \mathit{weight}(s_1(Y_1)) \times \ldots \mathit{weight}(s_1(Y_{\ell-1}))$, the recursive call to `rflatten` in line 22 will flatten Y_ℓ with the position vector containing

$$\underbrace{[j_1, \ldots, j_m] + \ldots + [j_1, \ldots, j_m]}_{r' \text{ times}}$$

This ensures that every tuple already produced in the recursive calls for $s(Y_1), \ldots, s(Y_{\ell-1})$ get paired with every tuple of $s(Y_\ell)$. To ensure that the they also get paired with the recursive calls that follow, the repetition vector for Y_ℓ specifies that the flattened result of each j_q is to be repeated $\mathit{weight}(s_i(Y_{\ell+1})) \times \cdots \times \mathit{weight}(s_i(Y_k))$ times. Lines 13–21 construct the correct position and repetition vector in this respect.

5 INSTANCE-OPTIMAL NSA EXPRESSIONS.

In this section we study the asymptotic complexity of shredded processing and identify a class of instance-optimal NSA expressions. We focus on the RAM model of computation with unit cost model and assume that hashing is $O(1)$ per tuple, both for hash map building as well as probing. We focus on data complexity, i.e., the NSA operators to be executed as well as the input and scheme of each operator is fixed.

Define the *size of shredding* $\mathcal{R} = (\mathsf{R}, \Sigma_R, \mathsf{r})$ of relation $R\colon X$, to be the sum of cardinalities of all physical relations in $\mathcal{R}$, i.e. $|\mathsf{R}| + \sum_{Y \in sub(X)} |\Sigma_R(Y)|$. Note that $|R|$ equals the size of $\mathcal{R}$ for flat relations. Similarly, the size of shredding $\mathcal{D} = (\mathsf{h}, \Sigma_D)$ of dictionary $D\colon \overline{y} \rightsquigarrow Z$ is $|\mathsf{h}|$ plus $\sum_{Y \in sub(Z)} |\Sigma_D(Y)|$ where $|\mathsf{h}|$ is the number of keys in h. By analysis of the physical operators proposed in Section 4.2 we readily obtain:

Proposition 5.1. *For every NSA operator except μ and μ^*, shredded processing runs in time $O(\mathit{IN})$ while μ and μ^* run in $O(\mathit{IN} + \mathit{OUT})$ where IN and OUT are the sizes of the operator's shredded input, and output, respectively.*

General NSA expressions may suffer from the diamond problem. Indeed, every binary join plan is a valid NSA expression by means of the equivalence (4). Hence, the NSA expression in Figure 8b, which is the equivalent of binary join plan $R(x, y) \bowtie (S(y, z) \bowtie T(z, u))$ exhibits the diamond problem when run on instances like db_2 from Figure 1c (see also Example 2.2). The utility of NSA for avoiding the diamond problem is as follows: all operators except μ, μ^* are *linear* and hence produce shredded outputs whose size is at most linear in that of the input. In contrast to the standard join, this is true in particular for the nested semijoin $R \triangleright\!\!\!\rightarrow D$. Indeed, every tuple in R can produce at most one tuple in $R \triangleright\!\!\!\rightarrow D$. As such, like the classic flat semijoin, the output of $R \triangleright\!\!\!\rightarrow D$ cannot increase in size. This is also the reason why we call $\triangleright\!\!\!\rightarrow$ a *nested semijoin*. By contrast, μ and μ^*, because they pair each tuple t in input R with the tuples in an inner relation of t, can produce outputs whose cardinality is not linear in the input size. Observe that μ, μ^* are hence the *only* operators that can cause "dangling tuples" to be created. This happens when they generate a more-than-linear subresult while another operator applied later removes tuples from this subresult. If no such later

[5] Additionally, `rflatten` takes the associated store Σ_R as input, as well as the physical relation 0 in which the output is to be constructed. We ignore these in our explanation.

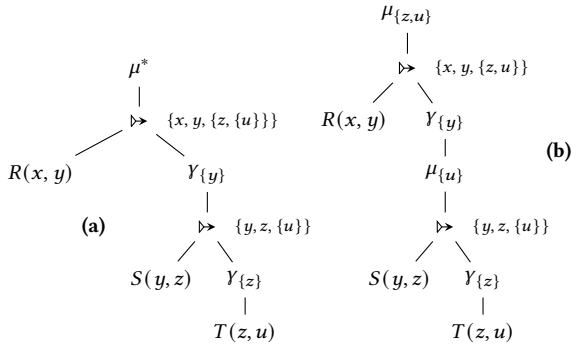


Figure 8: (a) A 2-phase NSA plan. (b) A non 2-phase NSA plan.

operator is applied, all tuples produced by μ, μ^* will appear in the output, and none will be dangling. This motivates the following definition.

Definition 5.2. An NSA expression is *non-shrinking* if it always produces an output (relation or dictionary) whose cardinality is at least as large as the cardinality of its largest input. An NSA expression is *2-phase* if, when viewed as a syntax tree, every μ and μ^* operator has only non-shrinking operators as ancestors.

In other words, the output of a 2-phase expression e is computed in two phases: a first phase where subexpressions generate linear-sized subresults (possibly filtering tuples from their input), and a second phase (delimited by the first μ or μ^*) where subexpressions may create subresults of larger-than-linear cardinality but where the tuples in these subresults, once created, can afterwards never be eliminated from the final output. It is important to stress that non-shrinking is a requirement on the output *cardinality* produced by an operator, not its size. In particular, the store in the operator's output may be smaller than that of either input.

The following theorem shows that all 2-phase NSA expressions avoid the diamond problem. We refer to [4] for the proof.

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.*

A similar result was observed in [5] for expressions with only $\ominus$ and $\copyright$. Here, we generalize it to include all other NSA operators.

It is straightforward to verify that π, ρ, $\cup$, μ and μ^* are the only non-shrinking operators in NSA. For π this holds because projection is bag-based; hence it has exactly the same output cardinality as the input. For ρ and $\cup$ this is trivial. Unnest and flatten themselves are non-shrinking because inner nested relations cannot be empty.

Example 5.4. Figure 8a is a two-phase NSA expression. Figure 8b is not two-phase since $\mu_{\{u\}}$ has $\triangleright\!\!\!\rightarrow$ as ancestor. Note that this plan is equivalent to $R(x,y) \bowtie (S(y,z) \bowtie T(z,u))$, which exhibits the diamond problem.

We should hence prefer 2-phase NSA expressions as physical query plans since these are the only expressions guaranteed to avoid the diamond problem. This begs the question of when a 2-phase NSA expression exists for a given query. The following theorem answers this question for join queries. Call an NSA expression a *join plan* if it uses only the operators γ, $\triangleright\!\!\!\rightarrow$, μ, and μ^*.

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.*

BKN [5] have already illustrated the "if" direction of Theorem 5.5; here we generalize it to a characterisation of the ayclic joins.

6 COMPARING BINARY JOIN PLANS TO 2-PHASE NSA PLANS

For parsimony, let us refer to binary join plans simply as "binary plans" and to 2-phase NSA plans as "2NSA plans" in what follows.[6] In Section 5 we have shown that 2NSA plans are *robust* physical plans for acyclic join queries, as we can evaluate such plans instance-optimally in $O(\text{IN}+\text{OUT})$ time. Instance-optimality, however, is only concerned with asymptotic complexity and may hide an important constant factor. In this section we therefore move from asymptotic complexity to analyzing plans based on a more detailed cost model that accounts for the sizes of the hashmaps built, the number of probes done in them, and the number of input data accesses.

Cost model. We adopt three abstract cost functions,

$$\mathit{build}\colon \mathbb{N} \to \mathbb{R}_{\geq 0} \qquad \mathit{probe}\colon \mathbb{N}^2 \to \mathbb{R}_{\geq 0} \qquad \mathit{take}\colon \mathbb{N} \to \mathbb{R}_{\geq 0},$$

so that $\mathit{build}(N)$ represents the runtime cost of a building a hash map on a relation with N tuples; $\mathit{probe}(N, M)$ represents the cost of probing N keys in a hash map of M entries; and $\mathit{take}(N)$ represents the cost of generating a single column (vector) of length N, whose content is populated by doing N accesses in an already existing column. In other words, $\mathit{take}(N)$ is the cost of `take(u,pos)` when $|pos| = N$. We assume monotonicity: if $N \leq N'$ and $M \leq M'$ then $\mathit{probe}(N, M) \leq \mathit{probe}(N', M')$ and similarly for *build* and *take*.

Let us analyze binary join and the NSA plan operators in this cost model. Let $\#R$ denote the number of attributes (flat or nested) in the scheme of R, i.e., if $R\colon X$ then $\#R = |X|$. Consider a traditional binary join $R \bowtie S$ of flat relations R and S on join keys $\overline{y}$. It will build on S, yielding a hash map with $|\gamma_{\overline{y}}(S)|$ keys. It probes into this hashmap from R, and needs to construct all columns in $R \bowtie S$. Its total cost hence is

$$C[\![R \bowtie S]\!] = \mathit{build}(|S|) + \mathit{probe}(|R|, |\gamma_{\overline{y}}(S)|) + \#(R \bowtie S) \times \mathit{take}(|R \bowtie S|).$$

Furthermore, by inspecting the physical operators given in Figures 6 and 7 we obtain, for nested relations R and S and dictionary D

$$\begin{aligned} C[\![\gamma_{\overline{y}}(S)]\!] &= \mathit{build}(|S|) \\ C[\![R \triangleright\!\!\!\rightarrow D]\!] &= \mathit{probe}(|R|, |D|) \\ C[\![\mu_Y(R)]\!] &= \#(\mu_Y(R)) \times \mathit{take}(|\,\mu_Y(R)|) \\ C[\![\mu^*(R)]\!] &= \#(\mu^*(R)) \times \mathit{take}(|\mu^*_Y(R)|) \end{aligned}$$

The cost of an entire plan (binary or NSA) is then the sum of costs of each individual operator, given the true cardinalities of the relations produced by the operator's subexpressions.

Example 6.1. It is instructive to compare the cost of right-deep binary plan $P = R(x,y) \bowtie (S(y,z) \bowtie T(z,u))$ with that of the 2NSA plan e shown Figure 8a. Let $k = \#(S \bowtie T)$ and $\ell = \#(P)$. Then

[6] Recall that 2NSA plans are 2-phase NSA expressions using only $\gamma, \triangleright\!\!\!\rightarrow, \mu$, and μ^*.

$$C[\![P]\!] = build(|T|) + probe(|S|, |\gamma_{\{z\}}\, T|) + k\ take(|S \bowtie T|) + build(|S \bowtie T|) + probe(|R|, |\gamma_{\{y\}}(S \bowtie T)|) + \ell\ take(|R \bowtie S \bowtie T|).$$

We note that this is exactly the cost of the (non-two-phase) NSA plan in Figure 8(b), which is obtained by applying equivalence (4) to P. The embedding of binary plans in NSA hence preserves cost.

To compute the cost of the 2NSA plan e of Figure 8a, we first note that the subexpression $S(y,z) \triangleright\!\!\rightarrow \gamma_{\{z\}}\, T(z,u)$ produces a nested relation whose cardinality is exactly $|S \ltimes T|$ (the flat semijoin between S and T). Its parent operator $\gamma_{\{y\}}$ therefore builds a hashmap on $|S \ltimes T|$ tuples. Continuing this reasoning yields

$$C[\![e]\!] = build(|T|) + probe(|S|, |\gamma_{\{z\}}\, T|) + build(|S \ltimes T|) + probe(|R|, |\gamma_{\{y\}}(S \ltimes T)|) + \ell\ take(|R \bowtie S \bowtie T|).$$

Since $|S \ltimes T| \leq |S \bowtie T|$, this is at most $C[\![P]\!]$ due to monotonicity.

The crucial reason why in Example 6.1 2NSA plan e has at most the cost of binary plan P is that e can be obtained from P by first turning P into an NSA plan using equivalence (4) (which yields the plan of Fig. 8b) and then rewriting the latter into a 2NSA plan by pulling to the top all μ operations, and combining them into a single μ^*. We next show that we can generalize this rewriting to arbitrary binary plans as long as they are *well-behaved*.

Well-behaved plans. Denote by $\mathrm{LL}(P)$ the *left-most leaf atom* of binary plan P, when viewing P as a tree. For example, if $P = (R \bowtie (S \bowtie T)) \bowtie U$ then $\mathrm{LL}(P) = R$. For plans that consist of a single atom, $\mathrm{LL}(P)$ is the atom itself. Denote by $\mathrm{LA}(P)$ the set of attributes of $\mathrm{LL}(P)$ and by $\mathrm{JA}(P)$ the set of join attributes of P's root join node. For example, $\mathrm{JA}\,(R(x,y) \bowtie (S(y,z) \bowtie T(z,u))) = \{y\}$. If P is a leaf relation then $\mathrm{JA}(P) = \emptyset$. A binary join plan P is *well-behaved* if for every subplan $P' = P_1 \bowtie P_2$ in P (including P itself) we have $\mathrm{JA}(P') \subseteq \mathrm{LA}(P_1)$ and $\mathrm{JA}(P') \subseteq \mathrm{LA}(P_2)$.

To illustrate, both the right-deep $R(x,y) \bowtie \big(S(y,z) \bowtie T(z,u)\big)$ and the bushy $[R(x,y) \bowtie \big(S(y,z) \bowtie T(z,u)\big)] \bowtie U(x,v))$ are well-behaved while $R(x,y) \bowtie \big(T(z,u) \bowtie S(y,z)\big)$ is not: there $\mathrm{JA}(P) = \{y\} \not\subseteq \{z\} = \mathrm{LA}(T(z,u) \bowtie S(y,z))$.

If P is well-behaved, then let P^ν be the 2NSA expression obtained by recursively replacing every $\bowtie$ by means of $\triangleright\!\!\rightarrow \gamma_{\mathrm{JA}(P)}$:

$$R^\nu = R \qquad (P_1 \bowtie P_2)^\nu = P_1^\nu \triangleright\!\!\rightarrow \gamma_{\mathrm{JA}(P_1 \bowtie P_2)}\, P_2^\nu.$$

Theorem 6.2. *P^ν is a well-typed 2NSA expression for every well-behaved binary plan P. Moreover, $\mu^*(P^\nu) \equiv P$ and the cost of $\mu^*(P^\nu)$ is at most that of P, on every database.*

Theorem 6.2 identifies a large class of binary plans for which we can find equivalent 2NSA plans without regret. Ill-behaved plans may incur additional cost when converted to 2NSA plans, as we illustrate in the full paper [4] .

Making binary plans well-behaved. We propose the following strategy for generating 2NSA plans that may performance-wise compete with the binary plans generated by existing query optimizers, while additionally being provably instance-optimal. If the optimizer already outputs a well-behaved plan, we simply execute $\mu^*(P^\nu)$, which is guaranteed to match the cost. Otherwise, we apply a dynamic programming algorithm to "repair" the ill-behaved plan, making it well-behaved while minimizing the additional cost, and execute the obtained well-behaved plan. This repair algorithm is described in detail in the full paper [4].

7 EXPERIMENTAL EVALUATION

Implementation. Leveraging the shredding approach introduced in Section 4, we implemented 2NSA plans inside Apache Datafusion [20] (v.34), a high-performance columnar query engine written in Rust that uses Apache Arrow as its data representation. Since Datafusion lacks a join order optimizer, we use the columnar engine DuckDB [32] (v1.0.0) to generate optimized plans for all considered queries. DuckDB's optimizer may introduce projections and filters in-between hash joins. To ensure that the resulting plans are strictly binary, we remove these intermediate filters and projections in the Datafusion binary plans, but keep filters and projections on input relations. Datafusion binary plans are subsequently transformed into 2NSA plans as discussed in Section 6.

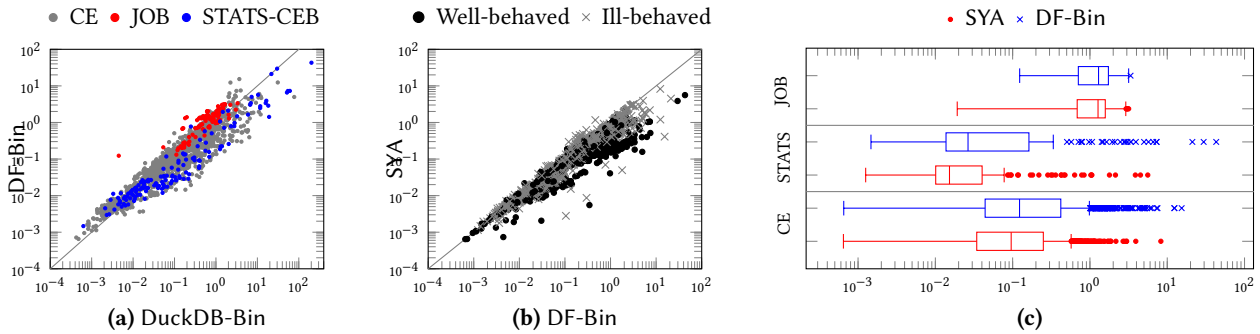
Setup. We first focus on interpreted columnar engines. There, we consider three ways of executing queries: DuckDB, using its original binary-join plans (DuckDB-Bin); Datafusion executing the stripped binary-join plans (DF-Bin); and our 2NSA implementation in Datafusion running the 2NSA plans (SYA). Subsequently, we extend this comparison to the compiled engine Umbra using the reproducability package of [5], both with and without L&E plans. To ensure fair comparison, we focus on plan execution time reporting the median of 10 runs. This excludes query optimization time but includes hash join time as well as reading input from disk, filters, projections and aggregations if applicable. We have also compared systems when considering only the hash join times. Due to space reasons, we defer these results, which show even more pronounced speedups, to the full paper version [4]. All experiments are conducted on a Ubuntu 22.04.4 LTS machine configured to use a single thread with an Intel Core i7-11800 CPU and 32GB of RAM.

Benchmarks. We employ three established benchmarks: the Join Order Benchmark (JOB) [21], STATS-CEB [16], and the cardinality estimation (CE) graph benchmark [7]. JOB and STATS-CEB contain only acyclic queries with base table filters and equijoins, followed by a single aggregation. We excluded query 7c from JOB due to an offset overflow error in Datafusion, and three STATS-CEB queries with output cardinalities exceeding 10^{10}, yielding 112 JOB queries and 143 STATS-CEB queries. The CE benchmark contains both cyclic and acyclic queries. After discarding the cyclic queries and the acyclic queries that ran out of memory, 1,594 queries remained. In summary, we employ a suite of 1,849 queries for our experiments.

DuckDB & Datafusion. We use log-log scatter plots, with each point representing a query's runtime under two approaches. The diagonal line indicates equal runtimes; points above (below) indicate slower (faster) performance for the approach on the Y-axis.

Figure 9a compares DuckDB-Bin with DF-Bin on the complete set of queries. DF-Bin is faster than DuckDB-Bin in 43% of the cases when considering total plan runtime, and in 60.4% of the cases when considering solely join time (not shown). Moreover, the average query runtime in DuckDB is 0.828 seconds, while Datafusion executes the same queries in 0.518 seconds on average. We conclude that DF-Bin is a robust baseline to use for further comparison against SYA, and focus on this comparison next.

Figure 9b compares DF-Bin with SYA. To be precise, SYA matches or outperforms DF-Bin on 94.6% of the queries (JOB), 94.4% (STATS-CEB), and 83.8% (CE). The improved robustness of SYA over DF-Bin is illustrated in Figure 9c, which shows a box-plot comparison



(a) DuckDB-Bin **(b)** DF-Bin **(c)**

Figure 9: Comparison of runtime performance in seconds: (a) DF-Bin vs. DuckDB-Bin; (b-c) SYA vs. DF-Bin.

Table 1: Speedups of SYA over DF-Bin and Umbra-L&E over Umbra-Bin.

		SYA absolute speedup (s)	SYA relative speedup	Umbra-L&E relative speedup
JOB	min	-0.09	0.76	**0.77**
	average	0.11	**1.21**	1.08
	median	0.01	1.01	**1.03**
	max	2.21	**6.42**	1.85
STATS-CEB	min	-0.01	0.87	**1.22**
	average	0.97	**3.13**	2.87
	median	0.01	2.07	**2.62**
	max	37.43	**23.97**	19.09
CE	min	-0.70	**0.40**	0.33
	average	0.20	**1.81**	1.65
	median	0.01	1.21	**1.44**
	max	14.96	**62.54**	16.82

of the absolute runtimes (log-scale) per dataset. The maximum speedups and slowdowns, both relative and absolute, are shown in Table 1. Slowdowns are small, as the maximum relative/absolute slowdowns are limited to 1.3x/89ms (JOB), 1.15x/5ms (STATS-CEB), and 2.5x/0.70s (CE). The maximum relative/absolute speedups are 6.4x/2.2s (JOB), 24x/37.4s (STATS-CEB), and 62.5x/15s (CE). We conclude that SYA is always competitive with binary join plans, and often much better, while at the same time guaranteeing robustness.

Recall from Section 6, that well-behaved plans can theoretically be translated into 2NSA plans without increasing their execution cost. We observe that 849 (46%) of the binary plans are well-behaved. 777 of these (92%) are indeed evaluated at least as fast by SYA as by DF-Bin in our experiments. For the remaining 72 queries where this is not the case, the absolute slowdown is at most 77ms. We conclude that our cost model, while an abstraction of reality, accurately predicts performance in the vast majority of cases. 54% of the binary plans are not well-behaved. For such plans the rewriting into a 2NSA plan is not guaranteed to be cost-preserving. However, in our experiments, for 80% of them, SYA is at least as fast as DF-Bin. This demonstrates that the benefit obtained by avoiding the diamond problem often outweighs the additional build and/or probe cost introduced by converting binary into 2NSA plans.

We next discuss some queries qualitatively. The query with the highest relative speedup (62.5x), is `yago_acyclic_star_6_39` from the CE benchmark. This well-behaved binary plan clearly suffers from the diamond problem, explaining the observed speedup: the query produces an intermediate result of 1.5×10^7 tuples, while the inputs are not larger than 2.1×10^5 and the output cardinality is 7.1×10^5. The query with the highest slowdown (2.5x) is query `yago_acyclic_chain_12_39` from the same benchmark. We observe that the binary plan is already well-optimized and does not suffer from the diamond problem. Moreover, the binary plan is not well-behaved, leading to a higher build cost in the 2NSA plan. This increased build cost, combined with the absence of the diamond problem in the binary plan, accounts for the observed slowdown.

Umbra. Absolutely speaking, binary join plans in Umbra (Umbra-Bin) execute on average one order of magnitude faster than DuckDB-Bin and DF-Bin. The same holds for L&E plans in Umbra (Umbra-L&E) compared to SYA (which is roughly Datafusion + L&E). We attribute this significant difference in absolute runtime to the fundamental differences between compiled query engines and interpreted column stores as well as other differences between the systems including the query planner and various low-level optimizations. When we compare the relative speedups obtained by SYA over DF-Bin to the speedups obtained by Umbra-L&E over Umbra-Bin, however, as shown in Table 1 we see that these speedups are comparable for all benchmarks. We can conclude that the benefits of L&E, originally implemented in a compiled query engine, can be successfully translated to column stores.

8 CONCLUSION

We have shown how to implement L&E decomposition inside column stores using nested relations and NSA as the logical model, and query shredding as the physical model. We have thereby illustrated the feasibility of implementing Yannakakis-style instance-optimal join processing inside a conventional main-memory columnar query engine *without regret*: fast on every acyclic join, and not only asymptotically. We hope that this perspective can help system engineers to better understand YA, and pave the way for its adoption into existing systems.

ACKNOWLEDGMENTS

This work was initiated during the research program on Logic and Algorithms in Database Theory and AI at the Simons Institute for the Theory of Computing. This research was supported by Hasselt University Bijzonder Onderzoeksfonds (BOF) Grants No. BOF22DOC07 and BOF20ZAP02, and the Research Foundation Flanders (FWO) under Grant No. G0B9623N.

REFERENCES

[1] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4 (2017), 20:1–20:44. https://doi.org/10.1145/3129246

[2] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4646)*, Jacques Duparc and Thomas A. Henzinger (Eds.). Springer, 208–222. https://doi.org/10.1007/978-3-540-74915-8_18

[3] Catriel Beeri, Ronald Fagin, David Maier, Alberto O. Mendelzon, Jeffrey D. Ullman, and Mihalis Yannakakis. 1981. Properties of Acyclic Database Schemes. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing, May 11-13, 1981, Milwaukee, Wisconsin, USA*. ACM, 355–362. https://doi.org/10.1145/800076.802489

[4] Liese Bekkers, Frank Neven, Stijn Vansummeren, and Yisu Remy Wang. 2025. *Instance-Optimal Acyclic Join Processing Without Regret: Engineering the Yannakakis Algorithm in Column Stores.* Technical Report. Full paper version, available at https://arxiv.org/abs/2411.04042.

[5] Altan Birler, Alfons Kemper, and Thomas Neumann. 2024. Robust Join Processing with Diamond Hardened Joins. *Proc. VLDB Endow.* 17, 11 (aug 2024), 3215–3228. https://doi.org/10.14778/3681954.3681995

[6] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. 1995. Principles of Programming with Complex Objects and Collection Types. *Theor. Comput. Sci.* 149, 1 (1995), 3–48. https://doi.org/10.1016/0304-3975(95)00024-Q

[7] Jeremy Chen, Yuqing Huang, Mushi Wang, Semih Salihoglu, and Kenneth Salem. 2022. Accurate Summary-based Cardinality Estimation Through the Lens of Cardinality Estimation Graphs. *Proc. VLDB Endow.* 15, 8 (2022), 1533–1545. https://doi.org/10.14778/3529337.3529339

[8] James Cheney, Sam Lindley, and Philip Wadler. 2014. Query shredding: efficient relational evaluation of queries over nested multisets. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 1027–1038. https://doi.org/10.1145/2588555.2612186

[9] Jan Van den Bussche. 2001. Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *Theor. Comput. Sci.* 254, 1-2 (2001), 363–377. https://doi.org/10.1016/S0304-3975(99)00301-1

[10] Alin Deutsch, Lucian Popa, and Val Tannen. 1999. Physical Data Independence, Constraints, and Optimization with Universal Plans. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie (Eds.). Morgan Kaufmann, 459–470. http://www.vldb.org/conf/1999/P44.pdf

[11] Ronald Fagin. 1983. Degrees of Acyclicity for Hypergraphs and Relational Database Schemes. *J. ACM* 30, 3 (1983), 514–550. https://doi.org/10.1145/2402.322390

[12] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *Proc. VLDB Endow.* 13, 11 (2020), 1891–1904. http://www.vldb.org/pvldb/vol13/p1891-freitag.pdf

[13] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. 2016. Hypertree Decompositions: Questions and Answers. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Tova Milo and Wang-Chiew Tan (Eds.). ACM, 57–74. https://doi.org/10.1145/2902251.2902309

[14] Georg Gottlob, Matthias Lanzinger, Davide Mario Longo, Cem Okulmus, Reinhard Pichler, and Alexander Selzer. 2023. Structure-Guided Query Evaluation: Towards Bridging the Gap from Theory to Practice. *CoRR* abs/2303.02723 (2023). https://doi.org/10.48550/ARXIV.2303.02723 arXiv:2303.02723

[15] M. H. Graham. 1979. *On the universal relation.* Technical Report. University of Toronto, Toronto, Ontario, Canada.

[16] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *Proc. VLDB Endow.* 15, 4 (2021), 752–765. https://doi.org/10.14778/3503585.3503586

[17] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. 2017. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1259–1274. https://doi.org/10.1145/3035918.3064027

[18] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2020. General dynamic Yannakakis: conjunctive queries with theta joins under updates. *VLDB J.* 29, 2-3 (2020), 619–653. https://doi.org/10.1007/S00778-019-00590-9

[19] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions Asked Frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Tova Milo and Wang-Chiew Tan (Eds.). ACM, 13–28. https://doi.org/10.1145/2902251.2902280

[20] Andrew Lamb, Yijie Shen, Daniël Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Liang-Chi Hsieh, and Chao Sun. 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 5–17. https://doi.org/10.1145/3626246.3653368

[21] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[22] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27, 5 (2018), 643–668. https://doi.org/10.1007/S00778-017-0480-7

[23] Riccardo Mancini, Srinivas Karthik, Bikash Chandra, Vasilis Mageirakos, and Anastasia Ailamaki. 2022. Efficient Massively Parallel Join Optimization for Large Queries. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 122–135. https://doi.org/10.1145/3514221.3517871

[24] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proc. VLDB Endow.* 12, 11 (2019), 1692–1704. https://doi.org/10.14778/3342263.3342643

[25] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.* 4, 9 (June 2011), 539–550. https://doi.org/10.14778/2002938.2002940

[26] Thomas Neumann. 2024. Closing the Gap between Theory and Practice in Query Optimization. In *Companion of the 43rd Symposium on Principles of Database Systems, PODS 2024, Santiago, Chile, June 9-15, 2024*. ACM, 4. https://doi.org/10.1145/3635138.3654765

[27] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[28] Thomas Neumann and Bernhard Radke. 2018. Adaptive Optimization of Very Large Join Queries. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 677–692. https://doi.org/10.1145/3183713.3183733

[29] Hung Q. Ngo. 2018. Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, Jan Van den Bussche and Marcelo Arenas (Eds.). ACM, 111–124. https://doi.org/10.1145/3196959.3196990

[30] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case Optimal Join Algorithms. *J. ACM* 65, 3 (2018), 16:1–16:40. https://doi.org/10.1145/3180143

[31] Dan Olteanu and Jakub Závodný. 2015. Size Bounds for Factorised Representations of Query Results. *ACM Trans. Database Syst.* 40, 1 (2015), 2:1–2:44. https://doi.org/10.1145/2656335

[32] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1981–1984. https://doi.org/10.1145/3299869.3320212

[33] Jaclyn Smith, Michael Benedikt, Milos Nikolic, and Amir Shaikhha. 2020. Scalable Querying of Nested Data. *Proc. VLDB Endow.* 14, 3 (2020), 445–457. https://doi.org/10.5555/3430915.3442441

[34] Konrad Stocker, Donald Kossmann, Reinhard Braumandl, and Alfons Kemper. 2001. Integrating Semi-Join-Reducers into State of the Art Query Processors. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, Dimitrios Georgakopoulos and Alexander Buchmann (Eds.). IEEE Computer Society, 575–584. https://doi.org/10.1109/ICDE.2001.914872

[35] Robert Endre Tarjan and Mihalis Yannakakis. 1984. Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs. *SIAM J. Comput.* 13, 3 (1984), 566–579. https://doi.org/10.1137/0213035

[36] Stan J. Thomas and Patrick C. Fischer. 1986. Nested Relational Structures. *Adv. Comput. Res.* 3 (1986), 269–307.

[37] Todd L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy (Eds.). OpenProceedings.org, 96–106. https://doi.org/10.5441/002/ICDT.2014.13

[38] Limsoon Wong. 1993. Normal Forms and Conservative Properties for Query Languages over Collection Types. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 25-28, 1993,*

Washington, DC, USA, Catriel Beeri (Ed.). ACM Press, 26–36. https://doi.org/10.1145/153850.153853

[39] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. IEEE Computer Society, 82–94.

[40] C. T. Yu and M. Z. Ozsoyoglu. 1979. An algorithm for tree-query membership of a distributed query. In *The IEEE Computer Society's Third International Computer Software and Applications Conference, COMPSAC 1979, 6-8 November, 1979, Chicago, Illinois, USA*. IEEE, 306–312. https://doi.org/10.1109/CMPSAC.1979.762509