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

Liese Bekkers
UHasselt, Data Science Institute

Frank Neven
UHasselt, Data Science Institute

Stijn Vansummeren
UHasselt, Data Science Institute

Yisu Remy Wang
University of California, Los Angeles

ABSTRACT

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

1 INTRODUCTION

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

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

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

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

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 [5, 35] 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 the so-called *query shredding* techniques for simulating nested relational algebra with standard relational algebra [7, 8, 32, 37]. 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 [19], a high-performance main-memory-based columnar 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 88.7% of the queries with speedups up to 188x—while at the same time guaranteeing robustness.

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 are given in the Appendix.

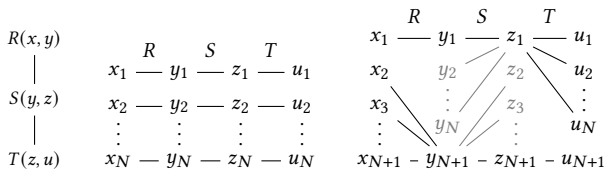
2 PRELIMINARIES AND BACKGROUND

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 $i \in [k]$



(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.

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 $supp(M)$ 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. For example, valid plans for Q_3 are $(R \bowtie S) \bowtie T$, which is left-deep, and $R \bowtie (S \bowtie T)$ which is right-deep. An example of a bushy plan is $(R \bowtie S) \bowtie ((T \bowtie U) \bowtie V)$.

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. 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.

Ayclicity and Yannakakis' Algorithm. A join query Q is *acyclic* if it admits a join tree [3, 10]. 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 [14, 34, 39]. A seminal result by Yannakakis [38] 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.

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. Then, during the first two passes, 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 [13]. These kinds of query plans, which we will refer to as *semijoin plans*, are binary join plans where leaf nodes

[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'.

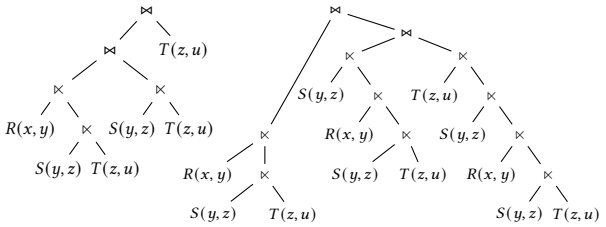


Figure 2: Semijoin plans induced by YA on join tree J_3(Fig.1a). Left: pass two and three combined. Right: all three passes.

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 [21], 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 [33]. 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, 16, 17]. 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, 16, 17]. 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* [35]. Many variants of the nested relational model have been proposed, including extensions that allow for mixed collection types such as sets, bags, lists, arrays [5] as well as dictionaries [9]. 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. We refer to the attributes that appear in the schema of classical flat relations as *flat* attributes. Let $\mathcal{A}$ denote the set of all flat attributes. The set $HF(\mathcal{A})$ of *hereditarily finite sets over* $\mathcal{A}$ is the smallest set containing $\mathcal{A}$, such that if $X_1, \ldots, X_n \in HF(\mathcal{A})$ then also $\{X_1, \ldots, X_n\} \in HF(\mathcal{A})$. A *scheme* is an element $X \in HF(\mathcal{A}) \setminus \mathcal{A}$ in which no flat attribute occurs more than once. Here, an element x is said to occur in X if $x \in X$ or x occurs recursively in some set $Y \in X$. We write $\mathcal{A}(X)$ for the set of all flat attributes occurring in X, and $sub(X)$ for the set of all schemes occurring in X. Schemes are also called *nested* attributes. Note that a flat attribute is not a scheme. We range over flat attributes by lowercase letters (x, y, ...) and over nested attributes by uppercase letters (X, Y, ...), both from the end of the alphabet. A finite set of flat attributes is denoted by $\overline{x}$.

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 $x \in X \cap \mathcal{A}$, and $t(Y)$ is a non-empty relation over Y for each nested attribute $Y \in X \setminus \mathcal{A}$. 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.

Example 3.1. Figure 3 shows a nested relation with cardinality 2 and scheme $\{x, \{y\}, \{u, \{v\}\}\}$. This scheme has two nested attributes, namely $\{y\}$ and $\{u, \{v\}\}$, and one flat attribute x.

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.

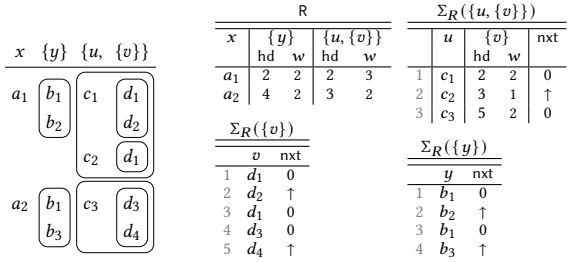


Figure 3: (left) A nested relation R. (right) Its shredded representation $\mathcal{R}$. The gray numbers indicate tuple offsets; nxt points to the next tuple (via ↑) or is 0 when there is none.

Dictionaries. A *dictionary scheme* is an expression of the form $\overline{y} \rightsquigarrow Z$ with $\overline{y} \cap \mathcal{A}(Z) = \emptyset$. 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 of D are called the *keys* of D. The *cardinality* of D, denoted $|D|$ is the number $|dom(D)|$ of keys. We write $D : \overline{y} \rightsquigarrow Z$ 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.

The *group-by* operator $\gamma_{\overline{y}}$ when applied to a relation $R : X$ creates a dictionary $D : \overline{y} \rightsquigarrow Z$ with $Z = X \setminus \overline{y}$. It does so by grouping the tuples in R on the attributes in $\overline{y}$, and mapping each group-key to its group projected on Z. Formally, the result dictionary D 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 $R : X$ and a dictionary $D : \overline{y} \rightsquigarrow Z$. 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 denoted $X \sim \overline{y} \rightsquigarrow Z$. The nested semijoin operator $R \triangleright\!\!\triangleright D$ 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,

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

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

The unnest operator $\mu_Y(R)$ unnests a nested attribute $Y \in X$ from input relation $R : X$ 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.

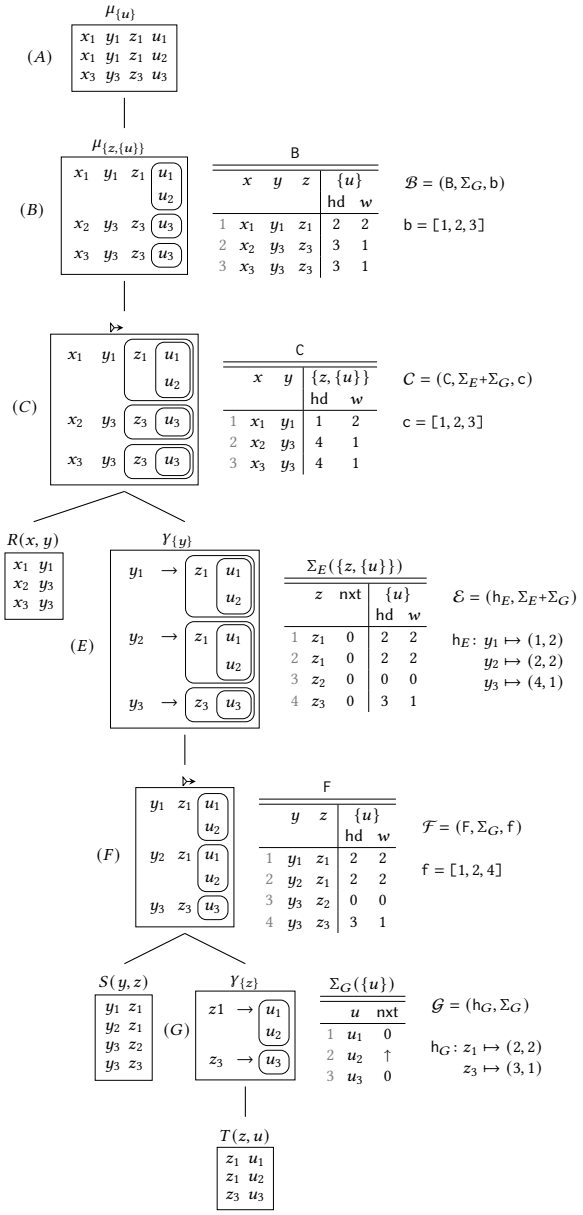


Figure 4: Example evaluation of an NSA expression. Intermediate nested relations and dictionaries are labeled $(A), (B), \ldots$ Shredded processing is illustrated on the right.

Finally, the flatten operator $\mu^*(R)$ completely flattens a nested relation $R : X$, returning a flat relation with scheme $\mathcal{A}(X)$. Specifically, if $Y_1, \ldots, Y_k$ is an enumeration of $sub(X)$, the nested attributes occurring in 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 $R : \{x, y, \{z\{u\}\}$ 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.

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

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

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

Figure 5: NSA type rules.

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 [35][2] are cleanly expressible in NSA as a composition of NSA operators. For example,

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

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

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

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

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

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 [7, 8, 32, 37]. We adapt this technique, known as *query shredding*, to implement NSA. We differ from traditional shredding in that there 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 and describe evaluation algorithms for each operator using this representation in Section 4.2.

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*

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

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

vector for R. Note that this implies $|p| \leq |\mathsf{R}|$, 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 $1 \leq i \leq |\Sigma_R(Y)|$, 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.

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 $1 \leq j \leq |\Sigma_R(Y)|$, 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 $(S \rtimes e_1) \rtimes e_2$, we do allow that in the shredding $\mathcal{R}$ for $R = S \rtimes e_2$, the physical relation R contains tuples that have already been filtered out by the nested semijoin, i.e., we allow that $|\mathsf{R}| \geq |R|$. 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 $\rtimes$). 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 from Σ_R by removing all entries in $\mathit{sub}(X \setminus Y)$. This works because $\pi_{\mathit{shred}(Y)}(\mathsf{R})$ retains only those columns in R required to represent $\pi_Y(R)$. Because the nested attributes in $\mathit{sub}(X \setminus Y)$ 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. We refer to the Appendix for their description.

The implementation of $\gamma, \rtimes$, 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 $|\mathsf{R}|$ 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 $R \rtimes D$ 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, $\Sigma_R + \Sigma_D$ 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`

[4]This is disjoint because the type rules for $\rtimes$ 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.

```
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.

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.

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: $\{x, y, \{z, \{u\}\}\}$. 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

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

$Y_1, \ldots, Y_k$. 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 $r_i * weight(s_i(Y_1)) \times \cdots \times weight(s_i(Y_k))$ 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 $weight(s_i(Y_2)) \times \cdots \times weight(s_i(Y_k))$ times to account for the cartesian products that follow. For the recursive call on Y_ℓ with $2 \leq \ell \leq k$ 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 $r' = r_i \times weight(s_1(Y_1)) \times \ldots weight(s_1(Y_{\ell-1}))$, 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 $weight(s_i(Y_{\ell+1})) \times \cdots \times weight(s_i(Y_k))$ 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 $O(\text{IN})$ while μ and μ^* run in $O(\text{IN} + \text{OUT})$ 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 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. The proof is in the Appendix.

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 [4] 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

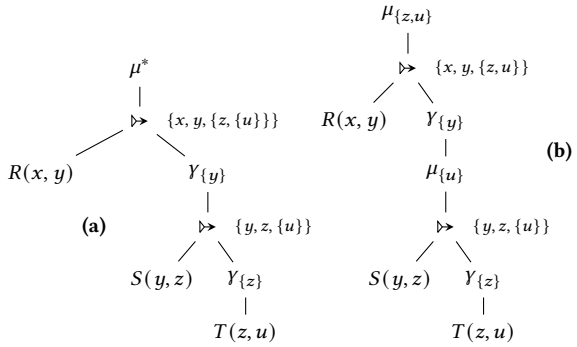


Figure 8: (a) A 2-phase NSA plan. (b) A non 2-phase NSA plan.

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.*

The proof is in the Appendix. BKN [4] 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. Indeed, recall from Section 2 that YA is also instance-optimal but often slower in practice because the required semijoin reduction implies building extra hash tables and doing extra probes compared to binary plans. 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

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

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

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

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 $\text{LL}(P)$ 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 $\text{LL}(P) = R$. For plans that consist of a single atom, $\text{LL}(P)$ is the atom itself. Denote by $\text{LA}(P)$ the set of attributes of $\text{LL}(P)$ and by $\text{JA}(P)$ the set of join attributes of P's root join node. For example, $\text{JA}(R(x, y) \bowtie (S(y, z) \bowtie T(z, u))) = \{y\}$. If P is a leaf relation then $\text{JA}(P) = \emptyset$. 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 $\text{JA}(P') \subseteq \text{LA}(P_1)$ and $\text{JA}(P') \subseteq \text{LA}(P_2)$.

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

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

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. We illustrate in Example C.5 in the Appendix that this is not possible for ill-behaved plans: equivalent 2NSA plans may incur additional cost.

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 the following dynamic programming algorithm to "repair" ill-behaved plans P while minimizing the additional cost.

Our "repair" algorithm actually constructs a join tree given an ill-behaved binary plan. This suffices, since given a join tree J, we can generate a well-behaved plan by induction: if A is the root of J, having child trees $J_1, \ldots, J_n$ (in this order) then construct the plan $((A \bowtie P_1) \bowtie \ldots) \bowtie P_n$ where P_i is the plan recursively constructed for J_i. In the Appendix we also show the converse direction: given a well-behaved plan one can construct a join tree. Well-behaved plans hence correspond one-to-one to join trees.

We define the repair algorithm by means of the set of mutually recursive functions shown in Figure 9. For every subplan P and atom $A \in P$ such that $\mathrm{JA}(P) \subseteq A$, $\tau_A(P)$ outputs the "optimal" join tree that is rooted by A and that covers all atoms of P. Here, "optimal" means that it incurs minimal cost penalty compared to P. This penalty is computed using δ. In the base case where $P = A$, $\tau_A(A)$ simply returns A. In the recursive case where $P = P_1 \bowtie P_2$, we first compute, for every atom A, the optimal A-rooted tree for the subplan that contains A. Then, for the other subplan P' we first use the function $\beta_{P'}(P)$ to compute the atom B of P' that contains $\mathrm{JA}(P)$ and for which $\tau_B(P')$ has minimal penalty among all candidates B. Subsequently we concatenate the trees with $t_1 +\!\!+ t_2$, attaching t_2 as the right-most child of the top-most atom (node) in t_1 containing $\mathrm{JA}(P_1 \bowtie P_2)$. The penalty $\delta_A(P)$ of the optimal tree rooted at A has a similar recursive structure. In the base case of a single atom the penalty is zero. In the recursive case, picking a root on the left incurs no additional cost, while picking a root B on the right incurs the extra cost of building on a relation of size $|B|$. Finally, the optimal tree for the entire plan P is the optimal tree for each choice of roots.

We note that the algorithm above assumes that the binary plan P satisfies the following property: for every node $P_1 \bowtie P_2 \in P$, there exist atoms $A \in P_1$ and $B \in P_2$ such that $\mathrm{JA}(P_1 \bowtie P_2) \subseteq \mathit{attr}(A) \cap \mathit{attr}(B)$. This ensures in particular that P is acyclic. All binary plans encountered in our experiments satisfy this property.

$$\tau_A(A) = A$$

$$\tau_A(P_1 \bowtie P_2) = \begin{cases} \tau_A(P_1) +\!\!+ \tau_{\beta_{P_1}(P_2)}(P_2) & \text{if } A \in P_1 \\ \tau_A(P_2) +\!\!+ \tau_{\beta_{P_2}(P_1)}(P_1) & \text{if } A \in P_2 \end{cases}$$

$$\delta_A(A) = 0$$

$$\delta_A(P_1 \bowtie P_2) = \begin{cases} \delta_A(P_1) + \delta_{\beta_{P_1}(P_2)}(P_2) & \text{if } A \in P_1 \\ \delta_A(P_2) + \delta_{\beta_{P_2}(P_1)}(P_1) + |\beta_{P_2}(P_1)| & \text{if } A \in P_2 \end{cases}$$

$$\beta_{P'}(P) = \operatorname*{arg\,min}_{B \in P \mid \mathrm{JA}(P' \bowtie P) \subseteq \mathit{attr}(B)} \delta_B(P)$$

$$\tau(P) = \tau_\alpha(P) \text{ where } \alpha = \operatorname*{arg\,min}_{A \in P \mid \mathrm{JA}(P) \subseteq \mathit{attr}(A)} \delta_A(P)$$

Figure 9: Dynamic programming algorithm to convert a binary plan into a well-behaved plan.

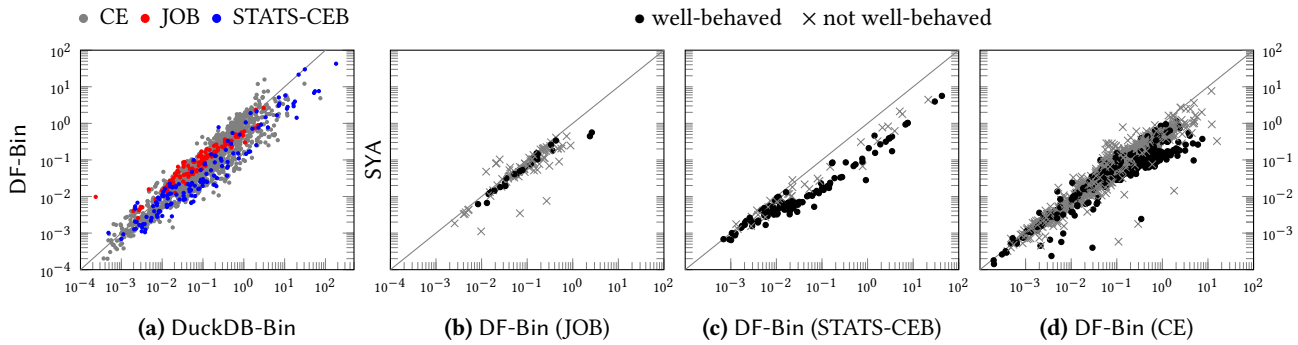
7 EXPERIMENTAL EVALUATION

We conduct an empirical evaluation of Shredded Yannakakis by comparing it to binary hash join on a comprehensive set of queries from well-established benchmarks.

Implementation. Leveraging the shredding approach introduced in Section 4, we implemented 2NSA plans inside Apache Datafusion [19] (v.34), a high-performance columnar query engine written in Rust that uses Apache Arrow as its in-memory data representation. Since Datafusion's query planner lacks a join order optimizer, we use DuckDB [31] (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. Corresponding 2NSA plans are obtained from the Datafusion binary plans through the algorithm of Figure 9.

Setup. 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). To ensure fair comparison, the reported runtimes represent the median of 10 runs accounting solely for the join execution time, excluding the time required for query planning, base table filtering, aggregation, and reading input relations from disk. 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) [20], STATS-CEB [15], and the cardinality estimation (CE) graph benchmark [6]. Both the JOB and STATS-CEB benchmarks consist of acyclic queries with only base table filters and equijoins, followed by a single aggregation. We excluded query 7c from JOB due to an offset overflow error that prevented its execution in Datafusion. Additionally, we removed three queries from STATS-CEB with an output cardinality exceeding 10^{10}, resulting in a final set of 112 queries for JOB and 143 for STATS-CEB. 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.



(a) DuckDB-Bin **(b)** DF-Bin (JOB) **(c)** DF-Bin (STATS-CEB) **(d)** DF-Bin (CE)

Figure 10: Comparison of runtime performance in seconds: (a) DF-Bin vs. DuckDB-Bin; (b)–(d) SYA vs. DF-Bin.

Plots. We use log-log scatter plots where each point corresponds to the runtime of a specific query, allowing us to compare the performance of two approaches. The diagonal line represents equal runtimes for both approaches. Points that lie above (below) this diagonal indicate cases where the runtime of the approach on the Y-axis is slower (faster).

DuckDB-Bin vs DF-Bin. Figure 10a compares DuckDB-Bin with DF-Bin on the complete set of queries, revealing that Datafusion achieves faster execution times for the majority of queries (68% of the queries to be precise) and maintains this advantage consistently across different runtime scales. We conclude that DF-Bin is therefore a robust baseline to use for further comparison against SYA, and focus on this comparison next.

JOB. Figure 10b shows that SYA either matches or outperforms DF-Bin on the majority of queries (92%), achieving speedups of up to 35x. We stress that the 8 queries where SYA is slower, remain fast to excute (below 0.12s). In fact, the query with the highest slowdown (4.1x) has an absolute runtime difference of only 78ms.

STATS-CEB. Figure 10c extends the runtime comparison to the STATS-CEB benchmark. Also here, SYA is faster than DF-Bin for almost all queries (96.50%), with speedups of up to 33x. The maximum slowdown is only 1.3x, while the highest absolute slowdown difference is restricted to 5.8ms.

CE. Figure 10d considers the CE benchmark and shows an overall improvement of SYA over DF-Bin. Notably, SYA outperforms in 87.8% of the queries, with speedups up to 188x. The maximum slowdown is 5.8x, and the highest absolute slowdown is 0.65s.

Well-behaved plans. We found that 46% of the binary plans are well-behaved. We showed in Section 6, that such plans are provably robust without regret. That is, can be translated into 2NSA plans without increasing their execution cost. We obtain that for the 848 binary plans that are well-behaved, 812 of them (96%) are indeed evaluated faster by SYA than DF-Bin. The 36 queries for which this is not the case all belong to the CE benchmark, and the highest absolute slowdown for a well-behaved plan is 55ms. We conclude that our cost model, while an abstraction of reality, accurately predicts performance in the vast majority of cases.

Ill-behaved plans. 54% of binary plans are not well-behaved. For such plans the rewriting into a 2NSA plan is not guaranteed to be cost-preserving. However, we do observe that for 83.8% of them, SYA is faster than 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.

Qualitative analysis. We focus on the query with the highest speedup (188x), specifically `yago_acyclic_tree_6_48` from the CE benchmark. Here, the binary plan clearly suffers from the diamond problem; the query produces an intermediate join result of 4.86×10^6 tuples, while the input relations are not larger than 1.87×10^4 and the output cardinality is only 18. Since the binary plan is not well-behaved, the conversion to a 2NSA plan is not cost preserving. Indeed, the 2NSA plan has to build on an additional input relation of 12,463 tuples compared to the binary plan. Nevertheless, the benefit of avoiding the diamond problem here significantly outweighs the extra build cost. We next discuss the query with the highest slowdown (5.8x), which is query `yago_acyclic_chain_12_73` from the same benchmark. We observe that the binary plan is already well-optimized: the input relations reach cardinalities up to 8.33×10^6, the output cardinality is 11, 605 and largest intermediate join result is only 4.7×10^4. The diamond problem does not show up here. Furthermore, 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 slowdown observed with SYA.

Conclusion. SYA demonstrates improvements over DF-Bin for the vast majority of queries (88.7%). Our experimental analysis further validates that the cost model introduced in Section 6 accurately predicts performance in 96% of cases. Even in the absence of formal guarantees, rewriting non-well-behaved plans into well-behaved ones consistently results in faster execution (83.8% of cases). Slowdowns are without regret: with a maximum absolute slowdown of 0.65s SYA remains competitive with DF-Bin.

8 CONCLUSION

We have shown how to implement the idea of L&E decomposition inside column stores using nested relations and NSA as the logical model, and query shredding as physical model. We have used this approach to illustrate 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.

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] 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

[5] 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

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

[13] 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

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

[15] 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

[16] 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

[17] 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

[18] 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

[19] 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

[20] 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

[21] 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

[22] 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

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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

[28] 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

[29] 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

[30] 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

[31] 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

[32] 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

[33] 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

[34] 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

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

[36] 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

[37] 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
[38] 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.
[39] 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

A EXTRA MATERIAL FOR SECTION 4 (SHREDDED PROCESSING)

In this section we give the full definition of the physical operators for $f \in \{\sigma, \pi, \rho, \cup, -\}$. Their implementation is shown in Figure **??**.

Let $R\colon X$. The implementation of selection $\sigma_{\theta(\overline{x})}(R)$ takes as argument the shredding $(\mathsf{R}, \Sigma_R, \mathsf{r})$ of R as well as the scheme X, predicate θ and its scheme $\overline{x}$. It simply builds a new selection vector s by iterating over all tuples in the selection vector r, and checking if the tuple satisfies θ.

The implementation of projection $\pi_Y(R)$ in Figure 11b was already discussed.

For the implementation of renaming $\rho_\varphi(R)$ we may assume w.l.o.g. that the φ only renames the flat attributes occurring in X, and leaves the flat $\mathsf{hd\_}Z$, $\mathsf{w\_}$, and nxt attributes that we have invented for shredding untouched. The implementation first extends φ to also map $\mathsf{hd\_}Z \mapsto \mathsf{hd\_}\varphi(Z)$ and then applies flat RA renaming with the extended φ to all physical relations in $(\mathsf{R}, \Sigma_)$.

For the implementation of difference $R - S$ we note that due to the typing rules, R and S must have the same flat scheme $\overline{x}$. The implementation takes the shreddings $(\mathsf{R}, \Sigma_R, \Sigma_r)$ and $\mathsf{S}, \Sigma_S, \Sigma_s)$ as input. Because the scheme of R and S is flat, Σ_R and Σ_S are empty, $\mathsf{r} = \mathsf{all}_\mathsf{R}$, and $\mathsf{s} = \mathsf{all}_\mathsf{S}$. It hence suffices to simply take flat RA difference of the top-level physical represenations R and S.

For the implementation of union $R \cup S$ we note that due to the typing rules, R and S must have the same scheme X. The implementation takes the shreddings $(\mathsf{R}, \Sigma_R, \mathsf{r})$ and $\mathsf{S}, \Sigma_S, \mathsf{s})$ as input, as well as their scheme X. For every nested attribute occurring (directly or recursively in X), we will take the point-wise union of $\Sigma_R(Y)$ and $\Sigma_S(Y)$ to compute the store of the output shredding. We have to take care to maintain the linked-lists encoded in the nxt column, however as well as the head-of-list positions encoded in the various $\mathsf{hd\_}Z$ for $Z \in sub(Y)$: when we take the pointwise union the positions in $\Sigma_S(Y)$ need to be shifted by the number of entries in $\Sigma_R(Y)$. We do so by means of the auxiliary recusive function $\mathtt{fix}$: given a (mutable reference to) physical relation T, it modifies T by modifying its $\mathsf{hd\_}Z$ columns as well as its nxt column (if it has it). The modification is done through the function $\mathtt{offset}$ (definition not shown) which takes a (mutable reference to) a vector and a natural number n, and increments each entry with a non-zero value by n.

Having computed the output store, it then remains to also take the flat RA union of the top-level physical represenations R and S, also offsetting the $\mathsf{hd\_}Y$ entries in S.

B EXTRA MATERIAL FOR SECTION 5 (INSTANCE-OPTIMAL NSA EXPRESSIONS.)

This appendix contains the proof of Theorem 5.3. We require the following auxiliary definitions and results.

Recall that a store on scheme X is a collection Σ of physical relations, one physical relation $\Sigma(Y)$ for every $Y \in sub(X)$. If Σ is a store for a scheme X, then we define the size of Σ, denoted $size(\Sigma)$, as $\sum_{Y \in sub(X)} |\Sigma(Y)|$, the sum of cardinalities of all physical relations in Σ.

Define the *detailed size of shredding* $\mathcal{R} = (\mathsf{R}, \Sigma_R, \mathsf{r})$ of nested relation $R\colon X$, denoted $dsize(\mathcal{R})$, to be the pair $(|\mathsf{R}|, size(\Sigma_R))$. Note that, since r is a selection vector on R, we necessarily have $|\mathsf{r}| \leq |\mathsf{R}|$, which is why we do not include it in the detailed size as we are working under data complexity.

Similarly, define the *detailed size of* shredding $\mathcal{D} = (\mathsf{h}, \Sigma_D)$ of dictionary D to be the pair $(|\mathsf{h}|, size(\Sigma_D))$. Recall that $|\mathsf{h}|$ is the number of keys in hash-map h.

Definition B.1. Let f be an NSA operator and let f be its physical implementation. Call f *strongly linear* if the following holds.

- If f is unary, then for every legal input I to f (which may be a relation, or a dictionary), and for every shredded representation $\mathcal{I}$ of I with detailed size (N, M) it holds that the output representation $\mathsf{f}(\mathcal{I})$ is computed in time $O(N + M)$ under data complexity[7] and, moreover, the detailed size of the output shredded representation is $(O(N), O(M))$.[8]
- If f is binary, then for all legal pairs of inputs I_1 and I_2 to f (where I_1 will be a relation and I_2 may be a relation or a dictionary), and for all shredded representations $\mathcal{I}_1$ and $\mathcal{I}_2$ of I_1 and I_2, respectively, with detailed sizes (N_1, M_1) and (N_2, M_2), it holds that the output representation $\mathsf{f}(\mathcal{I}_1, \mathcal{I}_2)$ is computed in time $O(N_1 + M_1 + N_2 + M_2)$ and, moreover, the detailed size of this representation is $(O(N_1 + N_2), O(M_1 + M_2))$.

From the definition of the physical operators given in Figures 6 and Figure 11 it is straightforward to obtain the following.

Proposition B.2. *All NSA operators except γ, μ and μ^* have shredded implementations that are strongly linear.*

We note that the $\mathtt{groupby}$ implementation of γ is almost strongly linear, but does not satisfy the detailed output size requirement. Instead it is straightforward to see:

Proposition B.3. *Let $R\colon X$ be an input to $\gamma_{\overline{y}}$ and let $\mathcal{R}$ be a shredded representation of R of detailed size (N, M). On input $\mathcal{R}$, the shredded implementation* $\mathtt{groupby}$ *computes in time $O(N + M)$ a shredded representation of $\gamma_{\overline{y}}(R)$ that has detailed size $(O(N), O(N + M))$.*

We next analyze the detailed complexity of μ and μ^*, using the following notion.

Definition B.4. Let f be a unary NSA operator and let f be its physical implementation. Call f *strongly input-output linear* (strongly IO linear) if for every legal input I to f and every shredded representation $\mathcal{I}$ of I with detailed size (N_I, M_I), f computes a shredded representation $\mathcal{J}$ of $J = f(I)$ of detailed size (N_J, M_J) such that (i) $M_J = O(M_I)$ and (ii) this shredded representation is computed in time $O(N_I + M_I + N_J + M_J)$.

[7] Recall that under data complexity we consider all schema information to be of constant size.
[8] I.e., if the detailed size of output $\mathsf{f}(\mathsf{I})$ is (N', M') then $N' = O(N)$ and $M' = O(M)$.

```
1: def select(R, Σ_R, rθ(x̄)):
2:   s = []
3:   for i in r :
4:     key = R[i](x̄)
5:     if key ⊨ θ :
6:       s.append(i)
7:   if X is flat and |s| < |R| :
8:     for u in shred(X) :
9:       R.u = take(R.u, s)
10:    s = all_R
11:  return (R, Σ_R, s)
```

(a)

```
1: def project(R, Σ_R, r, X, Y):
2:   for Z ∈ (X \ Y) \ 𝒜 :
3:     drop(Σ_R, Z)
4:   return (π_shred(Y)(R), Σ_R)
```

(b)

```
1: def difference(R, Σ_R, r, S, Σ_S, s):
2:   # Well-typedness ensures that R and S
     have equal flat scheme x̄
3:   # Σ_R and Σ_S are hence empty
4:   # Selection vectors are hence all_R and
     all_S, respectively
5:   return (R − S, Σ_R, all_{R−S})
```

(c)

```
1: def rename(R, Σ_R, r, φ):
2:   # assume R: X
3:   # assume φ renames only attrs in X,
4:   # not the new hd_Z, w_Z attributes
5:   for Y in sub(X) :
6:     φ(hd_Z) = hd_φ(Z)
7:     φ(w_Z) = w̄φ(Z)
8:   Σ = empty store
9:   for Y in sub(X) :
10:    Σ(φ(Y)) = ρ_φ(Σ_R(Y))
11:  return (ρ_φ(R), Σ, r)
```

(d)

```
1: def union(R, Σ_R, r, S, Σ_S, s, X):
2:   # scheme of R = scheme of S = X
3:   if |r| < |R| : # r != all_R
4:     for u in shred(X) :
5:       R.u = take(R.u, r)
6:   if |s| < |S| : # s != all_S
7:     for u in shred(X) :
8:       S.u = take(S.u, s)
9:   Σ = empty store
10:  for Y in X \ 𝒜 :
11:    fix(Σ_S(Y), Σ_R, Σ_S, Y)
12:  for Y in sub(X) :
13:    Σ(Y) = Σ_R(Y) ∪ Σ_S(Y)
14:  fix(S, Σ_R, Σ_S, X)
15:  return (R ∪ S, Σ, all_{R∪S})
```

(e)

```
1: def fix(T, Σ_R, Σ_S, Y):
2:   # scheme of T = Y
3:   for Z ∈ Y \ 𝒜 :
4:     n = |Σ_R(Z)|
5:     offset(T.hd_Y, n)
6:     fix(Σ_S(Z), Σ_R, Σ_S, Z)
7:   if nxt is an attribute of T :
8:     n = |Σ_R(Y)|
9:     offset(T.nxt, n)
```

(f)

Figure 11: Physical implementation of the "standard" NSA operators, using traditional flat RA operators.

Proposition B.5. *The shredded implementations of both μ and μ^* are strongly input-output-linear.*

Proof. For μ and its physical operator `unnest` given in Figure 6, observe that the store of the output representation is the store of the input representation where some entries have been dropped. The size of the output store is hence at most that of the input store. This hence establishes the condition on the detailed output representation size. The time complexity follows straightforwardly from the definition of `unnest`: the vectors `pos_R` and `pos_Y` will be populated to be of the same length as N_J, the size of physical relation in the output representation `J` computed by `unnest`; hence the time spent constructing these vectors in lines 2–6 is $O(N_J)$. The `take` operations in lines 8–11 are linear in `pos_R` and `pos_Y`, respectively, hence $O(N_J)$.

For μ^* and its physical operator `flatten` given in Figure 7, observe that the store of the output representation is empty. It hence trivially satisfies the condition on the detailed output representation size. The time complexity follows straightforwardly from the definition of `flatten`: in every recursive call of `rflatten`, the position vector `pos` is bounded in length by the length of N_J, the size of physical relation in the output representation `J` computed by `flatten`. To be more precise, it is possible to show using an inductive argument that in every such call also $\sum_{1\leq \text{row}\leq|\text{pos}|} \texttt{rep}[\text{row}] \times |\text{pos}|$ and $\sum_{1\leq \text{row}\leq|\text{pos}|} \texttt{rep}[\text{row}]\times w[\text{row}]$ are bounded by N_J, where w is the weight vector computed in Line 8. Hence, computing the new vectors `npos` and `nrep` in lines 13–21, which provide the arguments to the recursive call and therefore have the same bound, runs in time $O(N_J)$. The computation of `w` in Line 8 is linear in the size of argument physical relation `R`, which is either the physical relation of `flatten`'s shredded input $\mathcal{I}$, or a physical relation of the store in $\mathcal{I}$. Hence, this is certainly $O(N_I + M_I)$. The call to generate in line 10 is also $O(N_J)$ since $N_J \geq \sum_{1\leq \text{row}\leq|\text{pos}|} \texttt{rep}[\text{row}]\times w[\text{row}]$. □

Proposition B.6. *The NSA operators $\pi, \rho, \cup, \mu$ and μ^* are non-shrinking, whereas $\sigma, -, \triangleright\!\!\!\rightarrow$ and γ are shrinking.*

Proof. Projection π is non-shrinking because projection is bag-based; hence it has exactly the same output cardinality as the input. For ρ and $\cup$, non-shrinking is trivial by definition. Unnest and flatten themselves are non-shrinking because inner nested relations cannot be empty.

All of σ, $-$ and $\triangleright\!\!\!\rightarrow$ may return a nested relation whose output cardinality is smaller than the cardinality of the largest input nested relation. Indeed, $\triangleright\!\!\!\rightarrow$ takes as input a nested relation R and a dictionary D; the produced output relation may have cardinality that is smaller than $|R|$.

Also γ is shrinking: it returns a dictionary whose cardinality can be smaller than the cardinality of the input. This happens in particular when the input has multiple copies of the same key, which will only occur once as a key in the dictionary. Therefore the dictionary's cardinality may be smaller than the cardinality of the input relation. □

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.*

Proof. Let $e\colon X$ be a 2-phase NSA expression such that X is flat. Let db be an input database and let $\text{IN} = \sum_{R(\overline{x})\text{ atom in }e} |db(R)|$ be the sum of the cardinalities of the input relations in db.

First, note that because X is flat, no ancestor of an μ or μ^* operator in e can be γ. This is because γ produces a dictionary which is not a relation, and which itself hence cannot be the output of e. Therefore, if an ancestor of μ or μ^* is γ then the dictionary produced by γ has to be "consumed" by a later operator before we are at e's root. The only operation that we can do with dictionaries, is using it in a $\triangleright\!\!\rightarrow$ operator. But since $\triangleright\!\!\rightarrow$ is a shrinking operator by Proposition B.6, it cannot occur as an ancestor of μ or μ^*.

This implies that our expression e is a well-typed expression generated by the following grammar

$$\begin{aligned} e &::= f \mid \mu_Y(e) \mid \mu^*(e) \mid \pi_Y(e) \mid \rho_\varphi(e) \mid e_1 \cup e_2 \\ f &::= R(\overline{x}) \mid \sigma_{\theta(\overline{y})}(f) \mid \pi_Y(f) \mid \rho_\varphi(f) \mid f_1 \cup f_2 \\ &\quad \mid f_1 - f_2 \mid \gamma_{\overline{y}}(f) \mid f_1 \triangleright\!\!\rightarrow f_2 \end{aligned}$$

We now note the following. Recall that for flat input relations R, the shredded representation is of the form $\mathcal{R} = (\mathsf{R}, \emptyset, \mathtt{all}_{\mathsf{R}})$ and has detailed size $(|R|, 0)$. Therefore, if f is any NSA expression in which neither μ nor μ^* occurs (cf the grammar above), we know from Propositions B.2 and B.3 that given any input database db, shredded processing will compute a shredded representation of $f(db)$ in time $O(\text{IN})$ and this shredded representation has detailed size $\big(O(\text{IN}), O(\text{IN})\big)$.

We next show by induction on e' that for any expression e' generated by the above grammar, computing the representation of $e'(db)$ runs in time $O(\text{IN} + |e'(db)|)$ and has detailed size $\big(O(\text{IN} + |e'(db)|), O(\text{IN})\big)$, from which the Theorem follows by taking $e' = e$ and observing that $\text{OUT} = |e'(db)|$.

- Case e' is of the form f. We have already established that computing a representation of $f(db)$ by means of shredded processing is done in time $O(\text{IN})$ and has detailed size $(O(\text{IN}), O(\text{IN}))$, from which the claim clearly follows.
- Case $e' = \mu_Y(e'')$. By induction hypothesis, a representation of $e''(db)$ can be computed in time $O(\text{IN}+|e''(db)|)$ and has detailed size $\big(O(\text{IN} + |e''(db)|), O(\text{IN})\big)$. Let (N'', M'') be the concrete detailed size of this representation. Because μ is strongly IO-linear by Proposition B.4, shredded processing computes a representation of $\mu_Y(e''(db))$ with concrete detailed size (N', M') in time $O(N' + M' + N'' + M'')$ such that $M' = O(M'')$. By definition of $\mathtt{unnest}$, the output representation $\mathcal{R} = (\mathsf{R}, \Sigma_R, \mathsf{r})$ that it produces satisfies $\mathsf{r} = \mathtt{all}_{\mathsf{R}}$. Hence $N' = |\mathsf{R}| = |\mathtt{all}_{\mathsf{R}}| = |\mathsf{r}|$. Because always $|\mathsf{r}|$ is the cardinality of the represented nested relation, we hence have $N' = |e'(db)|$. Therefore, $(N', M') = (O(|e'(db)|), M'') = \big(O(\text{IN} + |e'(db)|), O(\text{IN})\big)$, as desired. It also follows that the computation time is
$$\begin{aligned} &O(N' + M' + N'' + M'') \\ &\quad = O(N' + M'' + N'' + M'') \\ &\quad = O(N' + N'' + M'') \\ &\quad = O(|e'(db)|) + O(\text{IN} + |e''|(db)) + O(\text{IN}) \\ &\quad = O(\text{IN} + |e'(db)|) \end{aligned}$$
where in the last step we use the fact that $|e'(db)| \geq |e''(db)|$ as μ is non-shrinking.
- Case $e' = \mu^*(e'')$. Completely analogous to the previous case.
- Case $e' = \pi_Y(e'')$. By induction hypothesis, a representation of $e''(db)$ can be computed in time $O(\text{IN} + |e''(db)|)$ and has detailed size $\big(O(\text{IN} + |e''(db)|), O(\text{IN})\big)$. Because π is strongly linear, shredded processing on this representation computes a representation for $e'(db)$ in time $O\big(\text{IN} + |e''(db)|\big) + O(\text{IN})$ that has detailed size $\big(O(\text{IN} + |e'(db)|), O(\text{IN})\big)$. Since π is non-shrinking, $|e'(db)| \geq |e''(db)|$. Therefore, the computation time is
$$O\big(\text{IN} + |e''(db)| + \text{IN}\big) = O(\text{IN} + |e'(db)|$$
and the detailed representation size is $\big(O(\text{IN}+|e'(db)|), O(\text{IN})\big)$, as desired.
- Case $e = \rho_\varphi(e')$ and $e = e_1 \cup e_2$ are completely analogous to the previous case. □

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.*

Proof. BKN show that every acyclic join query can be evaluated by means of a 2-phase L&E plan. Every 2-phase L&E plan is also a 2-phase NSA join plan by (6) and (7). This hence proves the "if" direction.

For the converse direction, let us write $atoms(Q)$ for the bag (i.e., multiset) of all atoms in Q. Let $Q = R_1(\overline{x}_1) \bowtie \ldots \bowtie R_k(\overline{x}_k)$ be a join query and let e be a 2-phase NSA join plan that expresses Q. Then every atom in Q must be occur as an input relation in e, and it must occur as many times as it appears in Q or the cardinality of the flat output relation that it computes will not be correct. For a subexpression e' of e, let $atoms(e')$ be the sub-bag of $atoms(Q)$ that appear in e.

Let f be the largest subexpression of e that does not include μ or μ^*. Because e is two-phase, because ν and μ can only have non-shrinking operators as ancestors, and because $\triangleright\!\!\rightarrow$ is the only binary operator in $\{\mu, \mu^*, \gamma, \triangleright\!\!\rightarrow\}$ but is shrinking (Proposition B.6), it necessarily follows that $atoms(f) = atoms(e)$. We next show how to construct from f a join tree for $atoms(f) = atoms(e) = atoms(Q)$, which hence shows that Q is acyclic.

For our construction it will actually be simpler to construct a *width-1 generalized hypertree decomposition (GHD)* for $atoms(f)$ instead of a regular join tree. It is well-known that a width-1 GHD tree exists for a multiset of atoms if and only if a join tree exists.

A *width-1 GHD* is a rooted tree $\mathcal{T}$ such that (1) all of $\mathcal{T}$'s nodes are either atoms, or flat schemes; and (2) for every flat attribute appearing somewhere in $\mathcal{T}$, all nodes mentioning x are connected in $\mathcal{T}$.

We claim that for every subexpression $g\colon X$ of f that produces a nested relation, we can create a width-1 GHD $\mathcal{T}_g$ for $atoms(g)$ such that the root of $\mathcal{T}_g$ is exactly the scheme $X \cap \mathcal{A}$. And, for every dictionary subexpression $g\colon \overline{y} \to Z$ similarly we can create $\mathcal{T}_g$ for $atoms(g)$ with root the scheme $\overline{y}$. The result follows from this

claim since in particular $\mathcal{T}_f$ will be a width-1 GHD for $atoms(f) = atoms(Q)$.

The claim is proved by induction on g.

- If $g = R_i(\overline{x}_i)$ for some i then $\mathcal{T}_g$ consists of two nodes: the leaf node g itself, and the scheme $\overline{x}_i$ which is the parent of g and forms the root of $\mathcal{T}_g$.
- If $g = e_1 \triangleright\!\!\!\rightarrow e_2$ then $e_1 \colon X$ and $e_2 \colon \overline{y} \to Z$ with $\overline{y} \subseteq X$, for some $X, \overline{y}$, and Z. By induction hypothesis we have width-1 GHDs J_{e_1} and J_{e_2} for $atoms(e_1)$ resp. $atoms(e_2)$ with as roots the schemes $X \cap \mathcal{A}$ and $\overline{y}$, respectively. Then create the tree J_g by taking the union of the two trees $\mathcal{T}_{e_1}$ and $\mathcal{T}_{e_2}$ where we make the root of $\mathcal{T}_{e_2}$ a child of the root of $\mathcal{T}_{e_1}$. In particular, the root of $\mathcal{T}_{e_1}$ is the root of $\mathcal{T}_g$. It is readily verified that the result is a join tree (i.e., has the connectedness property) and its root has scheme $X \cap \mathcal{A}$.
- If $g = \gamma_{\overline{y}}(g')$ then $g' : X$ for some X with $\overline{x} \subseteq Z$ and $Z = X \setminus \overline{y}$. By induction hypothesis, there exists $\mathcal{T}_{g'}$ for $atoms(g') = atoms(g)$ whose root has scheme $X \cap \mathcal{A}$. Then let $\mathcal{T}_g$ be the width-1 GHD obtained by adding $\overline{y}$ as new root on top of $\mathcal{T}_{g'}$. □

C EXTRA MATERIAL FOR SECTION 6 (COMPARING BINARY JOIN PLANS TO 2-PHASE NSA PLANS)

To prove Theorem 6.2, we first observe the following equivalences. To avoid confusion in what follows, we explicitly add the database on which an NSA expression is executed in our cost formulas. Hence, $C[\![e]\!]_{db}$ denotes the cost of executing expression e on database db. We denote by $e(db)$ the result of executing NSA expression e on database db.

Lemma C.1. *Let $e_1 \colon X$ and $e_2 \colon \overline{y} \rightsquigarrow Z$ be NSA expressions with $X \sim \overline{y} \rightsquigarrow Z$. Then*

$$\mu^*\left(\mu^*(e_1) \triangleright\!\!\!\rightarrow e_2\right) \equiv \mu^*(e_1 \triangleright\!\!\!\rightarrow e_2) \text{ and}$$
$$C[\![\mu^*\left(\mu^*(e_1) \triangleright\!\!\!\rightarrow e_2\right)]\!]_{db} \geq C[\![\mu^*(e_1 \triangleright\!\!\!\rightarrow e_2)]\!]_{db}$$

for every database db.

Proof. Abbreviate

$$f \stackrel{\text{def}}{=} \mu^*\left(\mu^*(e_1) \triangleright\!\!\!\rightarrow e_2\right) \qquad \text{and} \qquad g \stackrel{\text{def}}{=} \mu^*(e_1 \triangleright\!\!\!\rightarrow e_2).$$

The scheme of $\mu^*(e_1)$ is $\mathcal{A}(X)$. Since $X \sim \overline{y} \rightsquigarrow Z$ we know in particular that $\overline{y} \subseteq X \subseteq \mathcal{A}(X)$. Hence, when computing subexpression $\mu^*(e_1) \triangleright\!\!\!\rightarrow e_2$ of f, all flat attributes in the scheme $\mathcal{A}(X)$ of $\mu^*(e_1)$ that are required to do a nested semijoin with e_2 are actually already in $X \cap \mathcal{A}$. Therefore, $\mu^*(\mu^*(e_1) \triangleright\!\!\!\rightarrow e_2)$ is equivalent to instead computing $e_1 \triangleright\!\!\!\rightarrow e_2$, and flattening the result. Hence, $f \equiv g$.

To see the claim concerning the cost, fix an arbitrary database db. Let $k = \#\mu^*(e_1)$ and $\ell = \#f = \#g$. Then

$$\begin{aligned} C[\![f]\!]_{db} &= C[\![\mu^*(e_1)]\!]_{db} + C[\![e_2]\!]_{db} \\ &\quad + probe(|\mu^*(e_1)(db)|, |e_2(db)|) \\ &\quad + \ell\ take(|f(db)|) \\ &= C[\![e_1]\!]_{db} + k\ take(|\mu^*(e_1)(db)|) + C[\![e_2]\!]_{db} \\ &\quad + probe(|\mu^*(e_1)(db)|, |e_2(db)|) \\ &\quad + \ell\ take(|f(db)|) \\ &\geq C[\![e_1]\!]_{db} + C[\![e_2]\!]_{db} \\ &\quad + probe(|e_1(db)|, |e_2(db)|) \\ &\quad + \ell\ take(|g(db)|) \\ &= C[\![g]\!]_{db} \end{aligned}$$

Here, the inequality uses the fact that $|\mu^*(e_1)(db)| \geq |e_1(db)|$ and the fact that $f(db) = g(db)$. □

Lemma C.2. *Let $e_1 \colon X$ and $e_2 \colon Y$ be NSA expressions and let $\overline{y} = \mathcal{A}(X) \cap \mathcal{A}(Y)$. If $\overline{y} \subseteq X$ and $\overline{y} \subseteq Y$, then*

$$\mu^*\left(e_1 \triangleright\!\!\!\rightarrow \gamma_{\overline{y}}\mu^*(e_2)\right) \equiv \mu^*(e_1 \triangleright\!\!\!\rightarrow \gamma_{\overline{y}}\, e_2) \text{ and}$$
$$C[\![\mu^*\left(e_1 \triangleright\!\!\!\rightarrow \gamma_{\overline{y}}\mu^*(e_2)\right)]\!]_{db} \geq C[\![\mu^*(e_1 \triangleright\!\!\!\rightarrow \gamma_{\overline{y}}\, e_2)]\!]_{db}$$

for every database db.

Proof. Abbreviate

$$f \stackrel{\text{def}}{=} \mu^*\left(e_1 \triangleright\!\!\!\rightarrow \gamma_{\overline{y}}\mu^*(e_2)\right) \qquad \text{and} \qquad g \stackrel{\text{def}}{=} \mu^*(e_1 \triangleright\!\!\!\rightarrow \gamma_{\overline{y}}\, e_2).$$

The scheme of $\mu^*(e_2)$ is $\mathcal{A}(Y)$, that of $\gamma_{\overline{y}}\mu^*(e_2)$ is $\overline{y} \rightsquigarrow \mathcal{A}(Y) \setminus \overline{y}$. Because $\overline{y} = \mathcal{A}(X) \cap \mathcal{A}(Y)$, $\overline{y} \subseteq X$ and $\overline{y} \subseteq Y$ we know that $X \sim \overline{y} \rightsquigarrow \mathcal{A}(Y) \setminus \overline{y}$. Hence, f is well-typed.

The scheme of e_2 is Y, that of $\gamma_{\overline{y}}\, e_2$ is $\overline{y} \rightsquigarrow Y \setminus \overline{y}$. Since $\overline{y} \subseteq X$ and $\mathcal{A}(X) \cap \mathcal{A}(Y \setminus \overline{y}) = \emptyset$, $X \sim \overline{y} \rightsquigarrow Y \setminus \overline{y}$. Hence, also g is well-typed.

Since $\overline{y} \subseteq Y$ we know that when computing subexpression $\gamma_{\overline{y}}\mu^*(e_2)$ of f, all flat attributes $\overline{y}$ in the scheme $\mathcal{A}(Y)$ of $\mu^*(e_2)$ that are required for the grouping and that serve as keys for the later nested semijoin with e_1, are actually already in Y, the scheme of e_2. Therefore, $\mu^*\left(e_1 \triangleright\!\!\!\rightarrow \gamma_{\overline{y}}\mu^*(e_2)\right)$ is equivalent to instead computing $e_1 \triangleright\!\!\!\rightarrow \gamma_{\overline{y}}\, e_2$, and flattening the result. Hence, $f \equiv g$.

To see the claim concerning the cost, fix an arbitrary database db. Let $k = \#\mu^*(e_2)$ and $\ell = \#f = \#g$. Then

$$\begin{aligned} C[\![f]\!]_{db} &= C[\![e_1]\!]_{db} + C[\![\mu^*(e_2)]\!]_{db} \\ &\quad + build(|\mu^*(e_2)(db)|) \\ &\quad + probe(|e_1(db)|, |\gamma_{\overline{y}}\mu^*(e_2)(db)|) \\ &\quad + \ell\ take(|f(db)|) \\ &= C[\![e_1]\!]_{db} + C[\![e_2]\!]_{db} + k\ take(|\mu^*(e_2)(db)|) \\ &\quad + build(|\mu^*(e_2)(db)|) \\ &\quad + probe(|e_1(db)|, |\gamma_{\overline{y}}\mu^*(e_2)(db)|) \\ &\quad + \ell\ take(|f(db)|) \\ &\geq C[\![e_1]\!]_{db} + C[\![e_2]\!]_{db} \\ &\quad + build(|e_2(db)|) \\ &\quad + probe(|e_1(db)|, |\gamma_{\overline{y}}\, e_2(db)|) \\ &\quad + \ell\ take(|g(db)|) \\ &= C[\![g]\!]_{db} \end{aligned}$$

Here, the inequality uses the fact that $|\mu^*(e_2)(db)| \geq |e_2(db)|$, the fact that $|\gamma_{\overline{y}}\mu^*(e_2)(db)| \geq |\gamma_{\overline{y}}\, e_2(db)|$ because every $\overline{y}$-key of $\mu^*(e_2)(db)$ is also in $e_2(db)$, and the fact that $f(db) = g(db)$. □

Lemma C.3. *If binary plan P is well-behaved, then P^ν is well-typed with scheme X. The flat attributes in X are exactly* $\mathrm{LA}(P)$*, i.e., $X \cap \mathcal{A} = \mathrm{LA}(P)$, and the set of all attributes contained in X (directly or recursively) is exactly $attr(P)$, the set of all flat attributes mentioned in $attr(P)$, i.e., $\mathcal{A}(X) = attr(P)$.*

Proof. The proof is by induction on P.

- If P is an atom $R(\overline{x})$ then $P^\nu = R(\overline{x})$, and the result trivially holds.
- If $P = P_1 \bowtie P_2$, then $P^\nu = P_1^\nu \mathbin{⋉\!\!\!\gamma}_{\mathrm{JA}(P)} P_2^\nu$. By induction hypothesis, the lemma holds for P_1^ν and P_2^ν. Let X_1 be the scheme of P_1^ν and X_2 be the scheme of P_2^ν. Then, because P is well-behaved, $\mathrm{JA}(P) \subseteq \mathrm{LA}(P_1) \subseteq X_1$ and $\mathrm{JA}(P) \subseteq \mathrm{LA}(P_2) \subseteq X_2$. Hence, the expression $\gamma_{\mathrm{JA}(P)}\, P_2^\nu$ is well-typed, and has dictionary scheme $\mathrm{JA}(P) \rightsquigarrow Z_2$ where $Z_2 = X_2 \setminus \mathrm{JA}(P)$. This dictionary scheme is compatible with X_1 since $\mathrm{JA}(P) \subseteq X_1$ and
$$\begin{aligned}&\mathcal{A}(X_1) \cap \mathcal{A}(Z_2)\\ &= \mathcal{A}(X_1) \cap \mathcal{A}(X_2 \setminus \mathrm{JA}(P))\\ &= \mathcal{A}(X_1) \cap (\mathcal{A}(X_2) \setminus \mathrm{JA}(P))\\ &= attr(P_1) \cap (attr(P_2) \setminus \mathrm{JA}(P))\\ &= attr(P_1) \cap (attr(P_2) \setminus (attr(P_1) \cap attr(P_2)))\\ &= \emptyset.\end{aligned}$$
Hence, P^ν is well-typed and has scheme $X = X_1 \cup \{Z_2\}$. The flat attributes of X are exactly the flat attributes of X_1, which by induction hypothesis equals $\mathrm{LA}(P_1) = \mathrm{LA}(P)$. Moreover,
$$\begin{aligned}\mathcal{A}(X) &= \mathcal{A}(X_1) \cup \mathcal{A}(Z_2)\\ &= \mathcal{A}(X_1) \cup \mathcal{A}(X_2 \setminus \mathrm{JA}(P))\\ &= attr(P_1) \cup (attr(P_2) \setminus \mathrm{JA}(P))\\ &= attr(P_1) \cup (attr(P_2) \setminus (attr(P_1) \cap attr(P_2)))\\ &= attr(P_1) \cup attr(P_2)\end{aligned}$$
□

Corollary C.4. *If binary plan P is well-behaved, then $\mu^*(P^\nu)$ is well-typed and has scheme $attr(P)$.*

Proof. By Lemma C.3, P^ν is well-typed with scheme X such that $\mathcal{A}(X) = attr(P)$. Therefore, $\mu^*(P^\nu)$ is also well-typed, with scheme $\mathcal{A}(X) = attr(P)$. □

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.*

Proof. Assume P is well-behaved. Then $\mu^*(P^\nu)$ is well-typed by Corollary C.4. Since P^ν does not contain any μ or μ^*, expression $\mu^*(P^\nu)$ is trivially two-phase. It remains to show equivalence and the cost bound, which we do by induction on P.

If $P = R(\overline{x})$ the statement holds trivially. So, assume $P = P_1 \bowtie P_2$ and assume that the induction hypothesis holds for P_1 and P_2. We distinguish four cases.

(1) Both P_1 and P_2 are atoms, say $P_1 = R(\overline{x})$ and $P_2 = S(\overline{y})$. In that case $P = R(\overline{x}) \bowtie S(\overline{y})$ and $\mu^*(P^\nu) = \mu^*\big(R \mathbin{⋉\!\!\!\gamma}_{\overline{z}} S\big)$ where $\overline{z} = \mathrm{JA}(P) = \overline{x} \cap \overline{y}$. These two are clearly equivalent (use equivalence (4), replacing μ by μ^*). Moreover, the binary join plan has cost
$$C[\![P]\!] = build(|S|) + probe(|R|, |\,\gamma_{\overline{z}}\, S|) + \ell\; take(|R \bowtie S|),$$
where $\ell = |\overline{x} \cup \overline{y}|$ denotes the total number of attributes occurring in the join result. The 2NSA plan has cost
$$C[\![\mu^*(P^\nu)]\!] = build(|S|) + probe(|R|, |\,\gamma_{\overline{z}}\, S|) + \#\mu^*(P^\nu)\; take(|\mu^*(P^\nu)|)$$
Due to equivalence of P and $\mu^*(P^\nu)$, we necessarily have $\#\mu^*(P^\nu) = \ell$ and $|\mu^*(P^\nu)| = |R \bowtie S|$. Hence, P and $\mu^*(P^\nu)$ have equal cost.

(2) Neither P_1 nor P_2 are atoms. By induction hypothesis, P_1 is equivalent to $\mu^*(P_1^\nu)$ and P_2 is equivalent to $\mu^*(P_2^\nu)$; both 2NSA expressions have a cost that is no worse than that of P_1 resp. P_2. Let $P(db)$ denote the result of evaluating P on db, and similarly for P_1, P_2. Let $\ell = |\,attr(P)|$. Then
$$\begin{aligned}C[\![P]\!]_{db} = \;& C[\![P_1]\!]_{db} + C[\![P_2]\!]_{db}\\ &+ build(|P_2(db)|)\\ &+ probe(|P_1(db)|, |\,\gamma_{\mathrm{JA}(P)}\, P_2(db)|)\\ &+ \ell\; take(|P(db)|).\end{aligned}$$
Let e be the NSA expression that simulates P by executing $\mu^*(P_i^\nu)$ instead of P_i for $i = 1, 2$, and joining the results,
$$e = \mu^*\left(\mu^*(P_1^\nu) \mathbin{⋉\!\!\!\gamma}_{\mathrm{JA}(P)} \mu^*(P_2^\nu)\right).$$
Note that while e itself is not two-phase, it is clearly equivalent to P. Moreover,
$$\begin{aligned}C[\![e]\!]_{db} = \;& C[\![\mu^*(P_1^\nu)]\!]_{db} + C[\![\mu^*(P_2^\nu)]\!]_{db}\\ &+ build(|\mu^*(P_2^\nu)(db)|)\\ &+ probe(|\mu^*(P_1^\nu)(db)|, |\,\gamma_{\mathrm{JA}(P)}\, \mu^*(P_2^\nu)(db)|)\\ &+ \#e(db)\; take(|e(db)|).\end{aligned}$$
Because e is equivalent to P, we have $\#e(db) = \ell$ and $|e(db)| = |P(db)|$. Moreover, because $\mu^*(P_i^\nu)$ is equivalent to P_i we have $|\mu^*(P_i^\nu)(db)| = |P_i(db)|$. Finally, by induction hypothesis we have $C[\![\mu^*(P_i^\nu)]\!]_{(db)} \le C[\![P]\!]_{db}$. Combining all of these, we conclude
$$C[\![e]\!]_{db} \le\; C[\![P]\!]_{db}.$$
We next prove the theorem by showing that $\mu^*(P^\nu)$ is equivalent to e, and has a cost that is no worse than e. We will do this by using Lemmas C.1 and C.2.

To that end, first observe that subexpression $\gamma_{\mathrm{JA}(P)}\, \mu^*(P_2^\nu)$ of e has dictionary scheme $\mathrm{JA}(P) \rightsquigarrow attr(P_2) \setminus \mathrm{JA}(P)$ since $\mu^*(P_2^\nu)$ has scheme $attr(P_2)$ by Corollary C.4. By Lemma C.3, subexpression P_1^ν of e has a scheme X such that $\mathcal{A}(X) = attr(P_1)$ and $\mathrm{JA}(P) \subseteq \mathrm{LA}(P_1) \subseteq X$. Because $attr(P_1)$ and $attr(P_2)$ have exactly $\mathrm{JA}(P)$ in common, it follows that the scheme X of P_1^ν is compatible with the dictionary scheme $\mathrm{JA}(P) \rightsquigarrow attr(P_2) \setminus \mathrm{JA}(P)$ of $\mu^*(P_2^\nu)$. Hence, the preconditions of Lemma C.1 apply when taking $e_1 = P_1^\nu$ and $e_2 = \gamma_{\mathrm{JA}(P)}\, \mu^*(P_2^\nu)$. Let e' be the following expression:
$$e' = \mu^*\left(P_1^\nu \mathbin{⋉\!\!\!\gamma}_{\mathrm{JA}(P)} \mu^*(P_2^\nu)\right).$$
By Lemma C.1, $e' \equiv e$ and has a cost that is no worse then e.

We next apply Lemma C.2 to e'. To that end, observe that by Lemma C.4, subexpression P_2^ν has a scheme Y such that $\mathcal{A}(Y) = attr(P_2)$ and $\mathrm{JA}(P) \subseteq \mathrm{LA}(P_2) \subseteq Y$. Since, by definition $\mathrm{JA}(P) = attr(P_1) \cap attr(P_2)$ it follows that $\mathcal{A}(X) \cap \mathcal{A}(Y) = \mathrm{JA}(P)$ and $\mathrm{JA}(P) \subseteq X$ and $\mathrm{JA}(P) \subseteq Y$. Hence, the preconditions of Lemma C.1

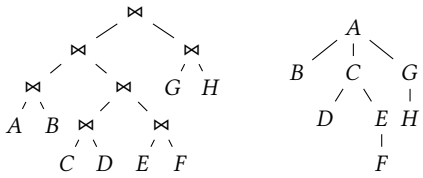


Figure 12: Correspondence between a well-behaved plan P (left) and a join tree (right). Letters A,B,... denote atoms.

apply when taking $e_1 = P_1^v$ and $e_2 = P_2^v$. Then, let e'' be the following expression:

$$e'' = \mu^* \left(P_1^v \triangleright\!\!\!\triangleright \gamma_{\mathrm{JA}(P)}\, \mu^*(P_2^v) \right)$$

This expression is equivalent to e' by Lemma C.2 and has a cost that is no worse than e'. Note that $e'' = \mu^*(P^v)$. Hence

$$\mu^*(P^v) = e'' \equiv e' \equiv e \equiv P$$

and

$$C[\![\mu^*(P^v)]\!]_{db} = C[\![e'']\!]_{db} \leq C[\![e']\!]_{db} \leq C[\![e]\!]_{db} \leq C[\![P]\!],$$

i.e., its cost is at most that of P.

(3) and (4) The cases where one of P_1, P_2 are atoms is entirely similar to the case (2). □

Discussion regarding ill-behaved plans. Theorem 6.2 identifies a large class of binary plans for which we can find equivalent 2NSA plans without regret. The following Example illustrates that this is not possible for ill-behaved plans: equivalent 2NSA plans may incur additional cost.

Example C.5. Consider $P_1 = R(x,y) \bowtie (T(z,u) \bowtie (S(y,z))$ which, as previously discussed, is not well-behaved. It builds on S and probes from T. Subsequently, it builds on $T \bowtie S$ and probes from R. We cannot guarantee the same costs in 2NSA: if we convert subplan $O = T(z,u) \bowtie (S(y,z)$ to O^v then we also build on S and probe from T, but we cannot construct a legal nested semijoin of R with O^v since O does not have the join attributes between R and O as flat attributes. The only legal equivalent 2NSA expressions will hence either have to build on S instead of T, or build on R and probe from O. As another example, consider the left-deep plan $P_2 = (R(x,y) \bowtie S(y,z)) \bowtie T(z,u)$, which is also not well-behaved. It builds on S and T, and probes from R into S, and from $R \bowtie S$ into T. If we convert subplan $N = R(x,y) \bowtie S(y,z)$ to N^v then again we cannot legally perform nested-semijoin with (the group-by on) T since the join attributes of P are not flat attributes of N^v. We can of course construct the plan from Figure **??** instead, which builds on $S \ltimes T$ and T, and hence is no worse than P in terms of build cost. However, note that in this NSA plan the number of probes into T is $|S|$, while in P this is $|R \bowtie S|$, which may be lower.

Correspondence between well-behaved plans and join trees. To transform a well-behaved P into a join tree J, iteratively replace every join node in P with its left child. Figure 12 illustrates the construction. Well-behavedness of P guarantees that the result satisfies the connectedness property, and is hence a valid join tree. Remark that the conversion of a join tree into a well-behaved plan described in Section 6 is exactly the inverse operation. Well-behaved plans and join trees are hence in one-to-one correspondence.