Indexed Streams: A Formal Intermediate Representation for Fused Contraction Programs

SCOTT KOVACH, Stanford University, USA
PRANEETH KOLICHALA, Stanford University, USA
TIANCHENG GU, Stanford University, USA
FREDRIK KJOLSTAD, Stanford University, USA

We introduce indexed streams, a formal operational model and intermediate representation that describes the fused execution of a contraction language that encompasses both sparse tensor algebra and relational algebra. We prove that the indexed stream model is correct with respect to a functional semantics. We also develop a compiler for contraction expressions that uses indexed streams as an intermediate representation. The compiler is only 540 lines of code, but we show that its performance can match both the TACO compiler for sparse tensor algebra and the SQLite and DuckDB query processing libraries for relational algebra.

CCS Concepts: • **Software and its engineering → Domain specific languages**; **Source code generation**.

Additional Key Words and Phrases: contractions, streams, operational semantics, functional programming

ACM Reference Format:
Scott Kovach, Praneeth Kolichala, Tiancheng Gu, and Fredrik Kjolstad. 2023. Indexed Streams: A Formal Intermediate Representation for Fused Contraction Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 154 (June 2023), 25 pages. https://doi.org/10.1145/3591268

1 INTRODUCTION

Languages for computing on irregular, high-dimensional data, such as sparse tensor algebra and relational algebra, are commonly used in many fields. For example, sparse linear and tensor algebra are used in scientific simulations and neural networks, relational algebra is used in data retrieval and processing, and both are used together in data analytics.

Performance is critical in these domains, as users often perform heavy computation on large data sets. Fast execution of tensor and relational operations requires both efficient iteration over irregular data structures and operator fusion. Operator fusion enables higher performance by reducing memory usage, avoiding unnecessary work, and moving computation closer to the data to make better use of the memory hierarchy.

Taking advantage of opportunities for fused execution across operations requires extensive manual effort or else a specialized compiler. In recent years, researchers and engineers have developed a number of compilers supporting fusion for sparse tensor algebra [Bik et al. 2022; Kjolstad et al. 2017], relational algebra [Kemper and Neumann 2011; Menon et al. 2017], and both together [Aberger et al. 2018; Schleich et al. 2019]. Moreover, these works have shown that operator fusion enables asymptotic speedups in both tensor algebra and relational algebra [Abo Khamis et al. 2016; Ngo et al. 2018].

Authors' addresses: Scott Kovach, Stanford University, USA, dskovach@stanford.edu; Praneeth Kolichala, Stanford University, USA, pkolich@stanford.edu; Tiancheng Gu, Stanford University, USA, timothygu@stanford.edu; Fredrik Kjolstad, Stanford University, USA, kjolstad@stanford.edu.

2475-1421/2023/6-ART154
https://doi.org/10.1145/3591268

However, prior compilers that generate fused code are complex and difficult to reason about. Their complexity makes it hard to ensure their correctness, posing problems for engineers and decision makers who rely on the results of scientific simulations and data analysis. If software can be designed using simple foundations and a small trusted computing base, then it carries a reduced surface area for software defects. Green et al. [2007] give an elegant mathematical model that can express problems involving tensors, relations, and more in a unified way. However, as a mathematical model, it does not directly provide the means for efficient computation.

Our primary contribution is a formal intermediate representation, called *indexed streams*, for the fused execution of sparse computations, including tensor and relational algebra. Indexed streams are an abstract data type that represents a stream of values labeled by ordered indices. They can be composed using the same algebraic operators as Green et al., but they have a precise computational interpretation. Given this interpretation, indexed streams can express three key optimizations that yield code with optimal asymptotic complexity [Ahrens et al. 2022; Ngo et al. 2018]:

(1) **Fusion:** Index stream operations are fused by default. By fusing computation, index streams can avoid unnecessary computation and memory allocation.
(2) **Hierarchical Iteration Order:** Indexed streams support hierarchical iteration and arbitrary iteration orders. By choosing the right iteration order, an indexed stream skips more unnecessary work in outer loops.
(3) **Sparse Data Structures:** Indexed streams can directly represent compressed data structures, which enable algorithms that iterate over only nonzero values.

We also define a language of *contraction expressions* that can express the core operations of relational algebra and tensor algebra. We show that indexed streams define a model for this language, and we prove that this new model is correct with respect to the semantics of Green et al. Our proof has been mechanized in the Lean theorem prover [Moura et al. 2015; Moura and Ullrich 2021].

Additionally, we show that indexed stream semantics captures known compilation strategies by using it to derive an extensible compiler for the contraction language in only 540 lines of code. Evaluated against the sparse tensor algebra compiler TACO, it supports the same data structures and its performance is competitive. It can also match the query evaluation performance of the SQLite [Hipp 2020] and DuckDB [Raasveldt and Mühleisen 2019] database systems. Our compiler allows full data structure customization and produces fused code with no additional burden on the user. Our technical contributions are

- the formally specified indexed stream intermediate representation (Section 5),
- a mechanized correctness proof that indexed stream composition operators are correct with respect to the denotational semantics of contraction expressions (Section 6), and
- a compiler for contraction expressions whose performance matches specialized systems for sparse tensor and relational algebra (Section 7).

2 MOTIVATING EXAMPLES

We highlight three performance concerns that significantly affect the design of contraction expression algorithms: operator fusion, hierarchical iteration, and data structure choice. Indexed streams are an abstract data type designed to address these issues in a compositional way.

2.1 Operator Fusion

When calculating an arithmetic expression involving several arrays or relations, fusing operators can have a large effect on performance. For instance, a typical language may evaluate an element-wise product of three vectors, $x \cdot y \cdot z$, by evaluating $v \coloneqq x \cdot y$ before evaluating $v \cdot z$. This approach requires additional memory and may take up to twice as many steps.

A *fused* execution of $x \cdot y \cdot z$ iterates across the coordinate set and compute the scalar expression $x_i \cdot y_i \cdot z_i$ for each triple of non-zero values. This avoids allocation and unnecessary traversals. Fusion can also achieve asymptotic speedup if z is sparse, as prematurely calculating $x \cdot y$ is wasteful [Ahrens et al. 2022].

2.2 Hierarchical Iteration Ordering

Data sets are often multi-dimensional, and we refer to a dimension over which a data set varies as an *attribute*. Many algorithms rely on *hierarchical* storage formats that allow more efficient access to specific attribute values.

For instance, when multiplying together several sparse factors, since $0 \cdot x = 0$, it is permissible to skip over any attribute value that is associated with a zero value in *any one* of the input factors. Skipping over an index at a higher level saves the work of operating on an entire slice of the computation in the inner loops. Storing data hierarchically requires an ordering of the attributes, and this choice can have significant consequences for performance.

Example 2.1. Suppose we wish to filter the relation $T : X \times Y \to \{0, 1\}$ by two predicates $p_X : X \to \{0, 1\}$ and $p_Y : Y \to \{0, 1\}$. A naïve algorithm might iterate across the entirety of T, testing each predicate on each tuple in T. Instead, a hierarchical representation of T would store the set of X attribute values that occur in T, and, for each one, the set of Y values associated with it. For a given $x \in X$, this would allow filtering out all the values $(x, y) \in T$ such that $p_X(x) = 0$ in one step. If the predicate p_Y is more selective (true for a smaller set of values) then performance would improve under the other attribute order which iterates across Y first.

2.3 Data Structure Abstraction

Users of high-performance computing systems require precise control over data structures. Sparse arrays are conventionally stored in compressed data structures that leave out zero values. These must be used when the total indexing set would be too large to represent otherwise. However, when dense representations are applicable, they offer faster access. Supporting arbitrary data structures in the presence of fusion and hierarchy presents an especially challenging composition problem.

Example 2.2. Suppose $T : X \times Y \to \{0, 1\}$ is a relation to be represented. If X is a set of numeric tuple identifiers, it might be wise to store the first level as a dense array of pointers into the lists of Y values. If X is the set of strings, the set of values $x \in X$ will more likely be stored using a sparse data format. A set of strings might be stored as a sorted array, a B-tree, a trie, a radix tree, or a dense dictionary-encoded set. The popular adaptive radix tree [Leis et al. 2013] combines several of these representations within one data structure. A common link between these data structures is that they implement an ordered traversal interface and efficient lookup.

3 OVERVIEW

The main contribution of our paper is the *indexed stream*, a formally specified intermediate representation for contraction expressions. We present a set of composable operations on indexed streams that can express any contraction expression as well as the optimizations we describe in Section 2 (fusion, hierarchical iteration, and sparse/dense data structures).

To demonstrate the advantages of indexed streams, we built the `Etch` compiler for the contraction expression language that we describe in Section 4. The indexed stream representation simplifies compilation compared to previous work on compiling sparse tensor algebra, which is a special case of contraction expressions. Our compiler is implemented in just 540 lines of Lean code, which is two orders of magnitude less code than the TACO compiler for sparse tensor algebra [Kjolstad et al. 2017] while being more general.

```
-- Variables x, y, z are sparse vectors.
-- They are represented by arrays
--   (_idx, _val) of length _len
--   passed as input.
--   (see right-hand listing)
def out : Var K   := "out"
def x   : i →ₛ K := sparse "x"
def y   : i →ₛ K := sparse "y"
def z   : i →ₛ K := sparse "z"
example := compile out (∑ i: x * y * z)
```

```
K out = 0;
while (x_i < x_len && y_i < y_len && z_i < z_len) {
  bool ready = x_idx[x_i] == y_idx[y_i] &&
               x_idx[x_i] == z_idx[z_i];
  if (ready)
    out += x_val[x_i] * y_val[y_i] * z_val[z_i];
  i index = max(x_idx[x_i], y_idx[y_i], z_idx[z_i]);
  skip(&x_i, x_idx, index, ready);
  skip(&y_i, y_idx, index, ready);
  skip(&z_i, z_idx, index, ready);
}
```

Fig. 2. The Lean code on the left defines a multiway (fused) dot product of sparse vectors and invokes our compiler. The C code on the right is the output of the compiler, modified for readability.

We show an overview of the Etch compiler (Section 7) in Figure 1. The compiler performs two lowering passes to generate imperative code. First, a contraction expression is translated into the indexed stream IR. At this level, input data is represented by typed stream variables, and contraction operations are implemented by stream constructors that build composite streams from simpler ones. The indexed stream IR is then lowered to an imperative loop nest that co-iterates over input data structures. This step is simple and directly motivated by the evaluation semantics of indexed streams from our formal model. The key organizing principle of the compiler is the indexed stream IR. Almost all of the hard work is done in the first step by library code that implements indexed stream constructors.

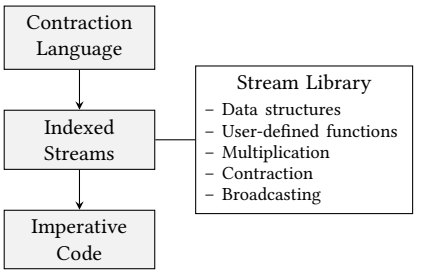

Fig. 1. An overview of the Etch compiler.

Figure 2 depicts an example Etch program and the generated C code. To efficiently compute the contraction of a three-way vector product, the output code simultaneously iterates across all three sparse vectors in a fused loop. When a common index value is found between them (when `ready` is true), the output is updated with the product of their current values. Then, the current maximum index value is used to advance each of the iterators using their `skip` functions. Depending on implementation details, this `skip` function may increment the state variable or use another method such as binary search to quickly advance to the desired index.

To further demonstrate the utility of the indexed stream semantics and to increase trust in its use, we formally prove that it faithfully computes the solutions of contraction expressions, both on paper and using the Lean proof assistant [Moura et al. 2015]. Figure 3 shows the structure of our correctness proof. The *contraction language* $\mathcal{L}$ formalizes the set of contraction expressions, the arrow $\mathcal{L} \to \mathcal{T}$ assigns a formal meaning to each expression, and finally the map $L \to S$ interprets contraction language expressions as indexed streams. The proof shows that stream evaluation, $[\![-]\!]$, makes the diagram commute. We describe indexed streams in Section 5.

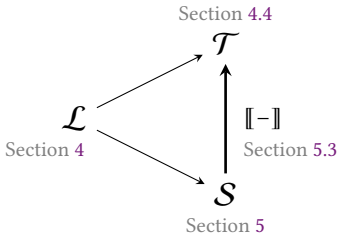

Fig. 3. The structure of our correctness proof, which proves that the indexed streams ($\mathcal{S}$) faithfully compute solutions to contraction problems expressed in our contraction language ($\mathcal{L}$), by relating both to simple functional semantics ($\mathcal{T}$).

4 CONTRACTION EXPRESSION LANGUAGE

We define a simple expression-oriented programming language, $\mathcal{L}$, for *contraction expressions* that compute sums,

$$
\begin{array}{rl}
\text{attribute} & a \in A \\
\text{shape} & S \in 2^A \\
\text{user-defined variable} & v \in V \\
\text{variable typing context} & \tau \,:\, V \to 2^A \\
\text{variable value context} & c \,:\, (v : V) \to I_{\tau(v)} \to K \\
\text{attribute renaming} & \rho \,:\, S \to A
\end{array}
$$

$$
\begin{array}{rl}
e ::= v & \textit{(variable)} \\
\mid e + e \mid e \cdot e & \textit{(arithmetic)} \\
\mid \Sigma_a e & \textit{(contraction)} \\
\mid \Uparrow_a e & \textit{(expansion)} \\
\mid \text{name}_\rho(e) & \textit{(rename)}
\end{array}
$$

(a) Syntax of $\mathcal{L}$

$$
\begin{array}{rl}
v \in V & \implies v : \tau(v) \\
e_1, e_2 : S & \implies e_1 + e_2 : S \\
e_1, e_2 : S & \implies e_1 \cdot e_2 : S \\
a \in S, e : S & \implies (\Sigma_a e) : (S \setminus \{a\}) \\
a \notin S, e : S & \implies (\Uparrow_a e) : (S \cup \{a\}) \\
e : S & \implies \text{name}_\rho(e) : \rho(S)
\end{array}
$$

(b) Typing rules of $\mathcal{L}$

$$
\begin{aligned}
[\![v]\!]^{\mathcal{T}}_c &= c(v) \\
[\![e_1 + e_2]\!]^{\mathcal{T}}_c &= [\![e_1]\!]^{\mathcal{T}}_c + [\![e_2]\!]^{\mathcal{T}}_c \\
[\![e_1 \cdot e_2]\!]^{\mathcal{T}}_c &= [\![e_1]\!]^{\mathcal{T}}_c \cdot [\![e_2]\!]^{\mathcal{T}}_c \\
[\![\Sigma_a e]\!]^{\mathcal{T}}_c &= \sum_{i \in I_a} [\![e]\!]^{\mathcal{T}}_c (a \mapsto i) \\
[\![\Uparrow_a e]\!]^{\mathcal{T}}_c &= [\![e]\!]^{\mathcal{T}}_c \circ \pi_{-a} \\
[\![\text{name}_\rho(e)]\!]^{\mathcal{T}}_c(t) &= [\![e]\!]^{\mathcal{T}}_c(t \circ \rho)
\end{aligned}
$$

(c) Semantics of $\mathcal{L}$

Fig. 4. The syntax, typing rules, and semantics of the language of contraction expressions $\mathcal{L}$

products, and aggregates over objects such as relations and tensors (see Figure 4, whose notation will be defined in this section). We define its denotational semantics in terms of functions defined over tuples. This semantics is a variant of the algebra defined by Green et al., who show that it is complete for relational algebra and matrix algebra. In Section 5, we introduce an alternative stream-based semantics that can be used to efficiently compute with programmatic representations of relations and tensors.

4.1 Language Syntax

Figure 4a shows the syntax of the expression language $\mathcal{L}$ and Figure 4b shows its types. The language includes addition, multiplication, the *contraction operator* Σ_a and the *expansion operator* $\Uparrow_a$. The type system assigns a *shape* to each expression, which is a set of *attributes*. The contraction operator Σ_a aggregates values across an attribute, while the expansion operator $\Uparrow_a$ repeats a value across an attribute. In the following subsections, we explain the ingredients needed for the denotational semantics and make these statements precise.

Variables represent inputs to a contraction expression, but they play two roles. First, variables model input data structures and, second, they can be bound to arbitrary user-defined functions. We discuss these capabilities of our compiler in Section 7.

Example 4.1 (Matrix multiply)**.** Suppose that $x : \{a, b\}$ and $y : \{b, c\}$ are two input variables that represent matrices, where a, b, and c are the attributes of their shapes. The standard notion of *matrix product* is represented by the contraction expression $\Sigma_b(\Uparrow_c x \cdot \Uparrow_a y)$. The $\Uparrow(-)$ operation is used to create two subexpressions with the same shape $\{a, b, c\}$ that are combined using elementwise product $(\cdot)$. The matrix product is computed by summing these products across the b attribute, yielding a result expression that has shape $\{a, c\}$.

4.2 Tuples and Schemas

A *schema* comprises all of the background type information needed to understand a contraction expression. We use the named perspective on relational data [Hall et al. 1975] to represent arrays and relations. In this perspective, a tuple is a map from its set of *attributes* to a set of values. Each

attribute is a unique name. For example, the attributes "time stamp", "url", "content" might be used by tuples storing web crawl data. We refer to the set of attributes on which a tuple is defined as its *shape*. We allow each attribute a to be associated with a distinct set I_a of values, and we call this set an *index set*. We summarize in the following definition:

Definition 4.2 (Schema)**.** A *schema* consists of a finite set A called the *attribute set* and, for each $a \in A$, a totally ordered set I_a called the *index set* for a. A subset $S \subseteq A$ is called a *shape*, and a function $t : (a : S) \to I_a$ is a *tuple*. The set of all tuples of shape S is denoted I_S, which is equivalent to the Cartesian product $\prod_{a \in S} I_a$. We refer to the universal set $I_A \simeq \prod_{a \in A} I_a$ as the *tuple space*.

We give two schema examples:

Example 4.3. To represent crawled webpages, we record tuples with a URL, a timestamp, and webpage content. Thus, the attribute set is $A_{\text{crawl}} = \{\text{url}, \text{ts}, \text{content}\}$ and $I_{\text{url}} = (\text{set of strings})$, $I_{\text{ts}} = \mathbb{N}$, and $I_{\text{content}} = (\text{set of strings})$. The tuple space is $(\text{strings} \times \mathbb{N} \times \text{strings}) \simeq I_{A_{\text{crawl}}}$.

Example 4.4 (A generic matrix)**.** Suppose we have two vector spaces with bases $I_a \coloneqq \{a_1, a_2, \ldots a_m\}$ and $I_b \coloneqq \{b_1, b_2, \ldots, b_n\}$. A linear operator between them can be written as a matrix with one entry for each pair of basis vectors. That is, a matrix assigns a number to each tuple $\{a \mapsto a_i, b \mapsto b_j\}$.

4.3 K-Relations

Our semantics uses a well-known functional representation for tensors and relations based on the *positive algebra* [Green et al. 2007]. The representation generalizes two commonly made observations:

- a relation is an *indicator function* on a Cartesian product of sets (a set of tuples);
- a multi-dimensional array (*tensor*) is a map from coordinate tuples to numeric values.

For example, a subset of the tuple set I_S can be encoded using a function $f : I_S \to \{0, 1\}$, where $f(t) = 1$ means t belongs to the subset. Such a function is a *relation* (no data is associated with a tuple besides its presence). On the other hand, a *multiset* or *bag* is a function $I_S \to \mathbb{N}$: here, $f(t)$ records the number of times t belongs to the multiset. More generally, a function $I_S \to K$, for some set of numbers K, is commonly called a *tensor* or multi-dimensional array. Generalizing from these examples, we restrict our attention to functions whose values come from a semiring.

Definition 4.5 (Semirings)**.** A semiring is a set K equipped with additive and multiplicative structures $(+, 0)$ and $(\cdot, 1)$ that satisfy the axioms of a commutative monoid and a monoid, respectively. They must also satisfy the distributive law $x(y+z) = xy + xz$, $(x+y)z = xz + yz$ and the absorption law $0 \cdot x = x \cdot 0 = 0$ for all $x, y, z \in K$.

Each of the semiring axioms is relevant to our problem domain: zero is used as the default value for tuples that are missing from a sparse object; multiplication satisfying $0 \cdot x = 0$ is useful to combine values while avoiding unnecessary work and preserving sparsity; addition is used to represent aggregation; and the distributive law enables factoring optimizations that perform contractions before products [Aji and McEliece 2000].

To distinguish them from ordinary functions, we refer to functions of the form $I_S \to K$ as *K-relations*. For intuition, they are essentially functions with keyword parameters, and elements of I_S are keyword argument tuples.

Definition 4.6 (K-relation)**.** Given a schema A, semiring K, and shape $S \subseteq A$, a K-relation of shape S is a function $f : I_S \to K$. The *support* of f is the set $\{t \in I_S \mid f(t) \neq 0\}$. Informally, a K-relation is *sparse* if its support is small compared to the cardinality of I_S (which may be infinite). We allow K-relations with infinite support.

$$
\begin{aligned}
M \odot N &\rightsquigarrow M \cdot N \\
(MN)_{ik} &\rightsquigarrow \Sigma_j(\Uparrow_k M \cdot \Uparrow_i N) \\
M + N &\rightsquigarrow M + N \\
\text{broadcast}_i(v) &\rightsquigarrow \Uparrow_i v
\end{aligned}
$$

Fig. 5. Translating matrix algebra into $\mathcal{L}$.

$$
\begin{aligned}
M_{S_1} \bowtie N_{S_2} &\rightsquigarrow (\Uparrow_{S_2 \setminus S_1} M) \cdot (\Uparrow_{S_1 \setminus S_2} N) \\
M_{S_1} \bowtie_p N_{S_2} &\rightsquigarrow \Uparrow M \cdot \Uparrow N \cdot \Uparrow p \\
M \cup N &\rightsquigarrow M + N \\
\sigma_p(M) &\rightsquigarrow p \cdot M \\
\pi_S(M) &\rightsquigarrow \Sigma_{A \setminus S} M
\end{aligned}
$$

Fig. 6. Translating relational algebra into $\mathcal{L}$.

4.4 Language Semantics

The semantics in Figure 4c maps a contraction expression to a K-relation. It is an adaptation of the positive algebra described by Green et al. to become a model of $\mathcal{L}$:

Definition 4.7 (K-relation Algebra $\mathcal{T}$)**.** Given a schema A and semiring K, the *K-relation algebra*, $\mathcal{T}$, consists of the sets $\mathcal{T}_S := I_S \to K$ of K-relations for all $S \subseteq A$. The function $[\![-]\!]_c^{\mathcal{T}}$ maps a contraction expression of shape S to an element of $\mathcal{T}_S$.

The definition of $[\![-]\!]_c^{\mathcal{T}}$ uses several standard operations on K-relations:

- **pointwise operations:** Given two K-relations $f, g : I_S \to K$ of the same shape, any binary operation on K can be applied pointwise. For example, we can define $(f \cdot g)(t) := f(t) \cdot g(t)$.
- **projection:** Given two shapes $S' \subseteq S$, the *projection operator* turns a tuple t of shape S into one of shape S' by simply restricting the domain: for all $a \in S'$, $(\pi_{S'}(t))(a) := t(a)$. A special case is π_{-a}, where $S' = S \setminus \{a\}$.
- **partial application:** Given a K-relation $f : I_S \to K$, attribute $a \in S$, and element $i \in I_a$, the *partial application* of f on i, written $f(a \mapsto i)$, is a K-relation of shape $S \setminus \{a\}$. It is defined by$f(a \mapsto i)(t) := f(\{a \mapsto i\} \cup t)$ for all $t \in I_{S \setminus \{a\}}$.
- **rename:** Given $S \subseteq A$, $f : I_S \to K$, and an inclusion $\rho : S \to A$ satisfying $I_{\rho(s)} = I_s$, the *rename* operation is $\text{name}_\rho(f)(t) := f(t \circ \rho) : I_{\rho(S)} \to K$. This operation does not change the content of f, only its shape.

The summation rule is well-defined only if $[\![e]\!]_c^{\mathcal{T}}$ has finite support. We allow input K-relations with infinite support, but only when they are multiplied by a K-relation with finite support.

In Figure 5 and Figure 6, we give a translation of some core expressions from tensor algebra and relational algebra into $\mathcal{L}$. Note that in every operation involving $\Uparrow$, the set of attributes to expand over can be inferred from the argument shapes and can be omitted.

5 INDEXED STREAMS

In this section we introduce automata that efficiently compute K-relations. These automata, which we call *indexed streams*, can be composed using the operations in the language $\mathcal{L}$ described in Section 4. The automata can also be composed hierarchically, and we use stream-valued streams to compute K-relations with more than one attribute. Although the components of an indexed stream are simple, they are sufficient to model the iteration patterns of practical sparse data structures as well as more complex behavior that results from composing them.

Streams model the traversal of a sequence of *values* associated with *indices*. We assume indices are totally ordered. They may pass through one or more *internal states* where they are not *ready* before producing a value. An indexed stream comes equipped with a *skip* function that advances its state according to a given index value.

Definition 5.1 (Indexed Streams)**.** Formally, given a schema with attribute set A, $a \in A$, and a set R, an *indexed stream of type $a \to_s R$* is a tuple $(\sigma, q_0, \text{index}, \text{value}, \text{ready}, \text{skip})$, where σ is its

internal state space and $q_0 \in \sigma$ its *current state*. The remaining components are functions with the following types and intuitive names:

$$\begin{aligned} \text{index} &: \sigma \to I_a, \\ \text{value} &: \sigma \to R, \\ \text{ready} &: \sigma \to \{0, 1\}, \\ \text{skip} &: \sigma \to (I_a \times \{0, 1\}) \to \sigma. \end{aligned}$$

When clear from context, we normally refer to an indexed stream and its current state using the same symbol; we use lower-case q, r, s, x, y, z for streams.

The components of an indexed stream have a conventional interpretation and several additional properties that they must satisfy in order for the stream to be well-formed. In this section, we will motivate these properties intuitively by way of examples. The full formal details are available in Section 6 and the Lean formalization.

The functions index, value, ready mark a given state with an index, value, and readiness, respectively. When $\text{ready}(q) = 0$ (representing false), the current value is ignored.

The function skip advances the state of the stream. Skip is used to efficiently advance a stream based on the current index of itself or another. Its role is explained when stream multiplication is introduced in the next subsection. At all states, index gives a lower-bound on the index of the next ready state. For the rest of the section, we assume that all of the streams we consider are *monotone*; that is, we assume $\text{index}(q) \le \text{index}(\text{skip}(q, (i, r)))$ for all q, i, r.

Example 5.2 (Dense and Sparse Vectors)**.** Suppose we have an indexing set $I_a = \{i_1, i_2, \ldots, i_n\}$ of size n. A dense vector over this set may be stored using a single array of length n containing values, one per index. A sparse vector may be stored as two separate arrays of the same length `len`: an array `vals` consisting of *only* the nonzero values, and a sorted array `inds` consisting of the strictly increasing sequence of indices corresponding to those values. We represent these using streams as follows:

$$\begin{aligned} \texttt{dense(vals)} &\coloneqq & \texttt{sparse(inds, vals)} &\coloneqq \\ \sigma &= \mathbb{N} & \sigma &= \mathbb{N} \\ q_0 &= 1 & q_0 &= 1 \\ \text{index}(q) &= i_q & \text{index}(q) &= \texttt{inds}[q] \\ \text{value}(q) &= \texttt{vals}[q] & \text{value}(q) &= \texttt{vals}[q] \\ \text{ready}(q) &= q \le n & \text{ready}(q) &= q \le \texttt{len} \\ \text{skip}(q, (i_k, r)) &= \max(q, k + r) & \text{skip}(q, (i_k, r)) &= \arg\min_{q' \ge q}(i_k + r \le \texttt{inds}[q']) \end{aligned}$$

Notice that in either case, when skip is passed the current value of $\text{index}(q)$ and $\text{ready}(q)$, it returns the immediately following state or stays put when $\text{ready}(q) = 0$, which is true when the stream has iterated past all of its values. This immediate successor transition function is important for all streams:

Definition 5.3 (δ)**.** For a stream $(\sigma, q_0, \text{index}, \text{value}, \text{ready}, \text{skip})$, define the *immediate sucessor function* (Figure 7) as:

$$\delta(q) \coloneqq \text{skip}(q, (\text{index}(q), \text{ready}(q))).$$

5.1 Contraction Operators for Streams

In this section, we show how to implement the contraction operators from $\mathcal{L}$ (addition, multiplication, contraction, and expansion) on indexed streams.

5.1.1 Multiplication. The notion of indexed stream allows us to give a universal definition of multiplication that produces efficient computational behavior for the streams arising from sparse tensor algebra and relational algebra.

Since we assume that $x \cdot 0 = 0 \cdot x = 0$, our operator performs the essential *intersection optimization*: it does not produce output at a given state unless both input streams are non-zero for the index value of that state.

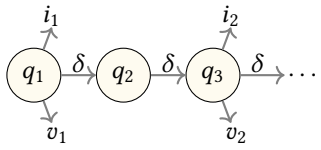

Fig. 7. A depiction of three successive states ($q_1, q_2, q_3 \in \sigma$) of a stream and the associated emitted indices ($i_1, i_2 \in I$) and values ($v_1, v_2 \in R$). The stream is not ready in state q_2 and thus does not emit a value.

Definition 5.4. Given streams $x, y : a \to_s R$ and a product operation $(\cdot) : R \times R \to R$, the *product stream* $(x \cdot y) : a \to_s R$ is defined by

$$\begin{aligned}
\sigma(x \cdot y) &= \sigma_x \times \sigma_y \\
\mathrm{index}(x, y) &= \max(\mathrm{index}(x), \mathrm{index}(y)) \\
\mathrm{value}(x, y) &= \mathrm{value}(x) \cdot \mathrm{value}(y) \\
\mathrm{ready}(x, y) &= \mathrm{ready}(x) \wedge \mathrm{ready}(y) \wedge \mathrm{index}(x) = \mathrm{index}(y) \\
\mathrm{skip}((x, y), i) &= (\mathrm{skip}(x, i), \mathrm{skip}(y, i)) \quad (\forall i \in I_a \times \{0, 1\})
\end{aligned}$$

The key observation used in the definition is that a product cannot be ready unless the inputs are both ready and agree on the index. Since we assume monotonically increasing indices, the maximum of $\mathrm{index}(a), \mathrm{index}(b)$ is a lower bound for the next ready state. Notice that the successor function (Definition 5.3) for this stream simply calls skip on each of its component states, but the index it passes them is the *max* of the two current indices. Thus, when multiply is used to combine two or more streams, the resulting stream implicitly combines information from all of them to direct their subsequent states.

Example 5.5 ($x \cdot y \cdot z$)**.** Consider the running example of a three-way sparse vector multiplication (Figure 2). Suppose our streams are defined as $x := \texttt{sparse}(i_x, v_x)$, $y := \texttt{sparse}(i_y, v_y)$, and $z := \texttt{sparse}(i_z, v_z)$ of type $a \to_s K$ (see Example 5.2). We can unfold the multiply and sparse definitions and simplify to obtain the following stream definition for $x \cdot y \cdot z$:

$$\begin{aligned}
\sigma(x \cdot y \cdot z) &= \mathbb{N} \times \mathbb{N} \times \mathbb{N} \\
\mathrm{index}(x, y, z) &= \max(i_x[x], i_y[y], i_z[z]) \\
\mathrm{value}(x, y, z) &= v_x[x] \cdot v_y[y] \cdot v_z[z] \\
\mathrm{ready}(x, y, z) &= x < \mathtt{len}_x \wedge y < \mathtt{len}_y \wedge z < \mathtt{len}_z \wedge i_x[x] = i_y[y] = i_z[z] \\
\mathrm{skip}((x, y, z), i) &= (\mathrm{skip}(x, i), \mathrm{skip}(y, i), \mathrm{skip}(z, i))
\end{aligned}$$

We define binary addition on streams using the same state space and skip function but using min in the index calculation. The full definition is given in the Lean formalization.

5.1.2 Contraction. The contraction operator Σ_a must produce a stream $\Sigma_a q$ that sums up all the values produced by a stream q. It does this with a small change to the stream's definition: it forgets the index associated with each value.

More precisely, suppose R is a set with addition. Given $q = (\sigma, q, \mathrm{index}, \mathrm{value}, \mathrm{ready}, \mathrm{skip}) : a \to_s R$, the stream $(\Sigma_a\, q) : * \to_s R$ is defined by

$$(\sigma = \sigma,\ \mathrm{index}(q) = *,\ \mathrm{value} = \mathrm{value},\ \mathrm{ready} = \mathrm{ready},\ \mathrm{skip}(q, (*, r)) = \mathrm{skip}(q, (\mathrm{index}(q), r))).$$

Note that the resulting stream is defined over a special indexing set $I_* := \{*\}$ that corresponds to a *dummy attribute* also denoted $*$. This stream is essentially identical to q except that its index is $*$ at all states. We explain how this implements summation in Section 5.3.

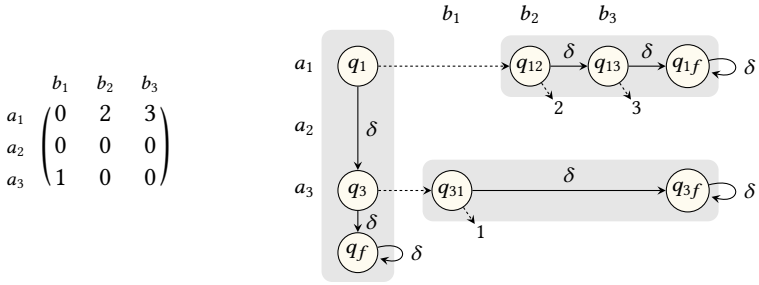

Fig. 8. A sparse matrix and corresponding nested indexed stream representing it. The dashed ⇢ arrow connects a state to its value. States without a value are not ready.

5.1.3 Expansion. Expansion repeats a single value at all of its states. In practice, it does not necessitate copying or recomputing the value; it may simply store a value and make it available repeatedly.

In particular, $\Uparrow_a$ is defined so long as I_a has a minimal element i_0 and a successor function $i + 1 \in I_a$ for all $i \in I_a$. Given a value $v \in R$, the expanded stream $(\Uparrow_a\ v) : a \to_s R$ is defined by

$$(\sigma = I_a, q_0 = i_0, \text{index}(i) = i, \text{value}(i) = v, \text{ready}(i) = 1, \text{skip}(i, (i', r)) = i' + r).$$

That is, the stream is always ready, always returns the value v, and iterates across the indices of I_a.

5.2 Nested Streams

So far we have only discussed streams with a single attribute. To achieve the performance goals we set out, we choose to represent K-relations with multiple attributes using *nested* indexed streams.

As an example, Figure 8 depicts a nested stream representing a sparse matrix. The stream has type $a \to_s b \to_s \mathbb{N}$. The outer stream is depicted by the series of transitions going down the figure. Each row is represented by a stream of type $b \to_s \mathbb{N}$ running from left to right. For example, value(q_3) is the stream with two states $\{q_{31}, q_{3f}\}$. Note that there is no state for the entirely zero row; it is completely omitted from iteration at the outer level. We assume that each stream has a terminal state (shown at the margins) where ready = 0.

In this subsection, we will describe how the stream operations extend naturally to nested streams and then characterize a subset of streams that are well-typed: their sequence of attributes has no duplicates and accords with a global attribute ordering. We use these definitions to define the *stream algebra*, consisting of well-typed streams $\mathcal{S}$ and the stream operations, which gives the alternate semantics for the contraction expression language $\mathcal{L}$.

Nested Stream Operations. The operations defined in the previous section generalize with no difficulty to nested indexed streams. For example, since K has multiplication, the set of streams with type $a_1 \to_s K$ has multiplication; thus the set $a_2 \to_s a_1 \to_s K$ has multiplication, $a_3 \to_s a_2 \to_s a_1 \to_s K$ has multiplication, and so on.

We note that streams are *functorial*: if $f : R \to S$, then there is a function $(\text{map}\ f) : (a \to_s R) \to (a \to_s S)$ defined by composing f with the value function of the stream.

Example 5.6. Map allows us to apply an operator to each value of a stream. Suppose $a, b \in A$ are attributes and $q : a \to_s K$. On the one hand, $\Uparrow_b\ q : b \to_s a \to_s K$, but on the other hand, map $(\Uparrow_b)\ q : a \to_s b \to_s K$. Map can be also iterated:

$$\text{map}^2\ (\Uparrow_c)\ (\text{map}\ (\Uparrow_b)\ q) : a \to_s b \to_s c \to_s K.$$

$$\llbracket v \rrbracket_c^{\mathcal{S}} = c(v) \qquad \llbracket \Sigma_a e \rrbracket_c^{\mathcal{S}} = \mathrm{map}^{\#(a,\tau(e))}\, \Sigma_a\, \llbracket e \rrbracket_c^{\mathcal{S}}$$
$$\llbracket e_1 + e_2 \rrbracket_c^{\mathcal{S}} = \llbracket e_1 \rrbracket_c^{\mathcal{S}} + \llbracket e_2 \rrbracket_c^{\mathcal{S}} \qquad \llbracket \Uparrow_a e \rrbracket_c^{\mathcal{S}} = \mathrm{map}^{\#(a,\tau(e))}\, \Uparrow_a\, \llbracket e \rrbracket_c^{\mathcal{S}}$$
$$\llbracket e_1 \cdot e_2 \rrbracket_c^{\mathcal{S}} = \llbracket e_1 \rrbracket_c^{\mathcal{S}} \cdot \llbracket e_2 \rrbracket_c^{\mathcal{S}} \qquad \llbracket \mathrm{name}_\rho(e) \rrbracket_c^{\mathcal{S}} = (\llbracket e \rrbracket_c^{\mathcal{S}} : \rho(S))$$

Fig. 9. The stream semantics $\mathcal{L} \to \mathcal{S}$. The context c maps a variable of shape S to a stream of shape S. The name operator changes the attribute labels of a stream (its type) without changing its behavior.

That is, $\mathrm{map}^k\, f$ reaches past k layers of the stream type and composes f with the next value function. In this way, we can use map to apply the $(\Sigma, \Uparrow)$ operations to any level of a nested stream.

Stream Algebra. To compute a contraction expression using nested streams we must choose an *attribute ordering*, and we must require that all input streams respect this ordering. In particular, the stream types $a \to_s b \to_s K$ and $b \to_s a \to_s K$ are not equivalent even though the sets of K-relations they represent are equivalent. A particular attribute ordering will enforce that all subexpressions have the same ordering for attributes a and b. This ensures that multiplication (*which requires matching types*) can always be applied to any expressions.

In order to relate streams to contraction expressions and the denotational semantics, we first define the shape of a stream. Recall that the shape of a K-relation is the set of attributes over which it is defined. In a similar way, the shape of a stream is the ordered sequence of attributes that appear in its type, ignoring the dummy attribute ($*$). For example, the stream type $a \to_s b \to_s * \to_s a \to_s K$ has shape $[a, b, a]$. Formally:

Definition 5.7 (Valid Streams)**.** The *stream shape* function τ maps a stream type to a sequence of attributes. It is defined inductively as follows:

$$\tau(K) \coloneqq [] \text{ (the empty sequence)},$$
$$\tau(* \to_s T) \coloneqq \tau(T),$$
$$\tau(a \to_s T) \coloneqq a :: \tau(T).$$

For a stream $q : T$, define $\tau(q) = \tau(T)$.

A *valid stream* is one where $\tau(q)$ is a subsequence of A, the attribute set, as an ordered sequence. That is, the ordering of attributes in the type of q respects the attribute ordering, no attribute appears more than once, and any number of occurrences of the dummy attribute may occur.

Definition 5.8 (Stream Algebra $\mathcal{S}$)**.** Suppose we have a semiring K, schema over attributes A, and a total ordering of A. The *stream algebra*, $\mathcal{S}$, consists of the sets $\mathcal{S}_S$ of valid streams for all $S \subseteq A$:

$$\mathcal{S}_S \coloneqq \{q \mid \tau(q) = S\}.$$

Define $\#(a, S) \coloneqq |\{a' \in S \mid a' < a\}|$ to be the number of attributes in S that come before a in the ordering. Note that if $\tau(q) = S$ and $a \notin S$, $k = \#(a, S)$ is the unique integer such that $\mathrm{map}^k \Uparrow_a q$ is a valid stream: map is needed to insert a at the correct position within S as an ordered sequence. Similarly, if $\tau(q) = S$ and $a \in S$, $\mathrm{map}^{\#(a,S)}\, \Sigma_a\, q$ is well-defined. With these operations and stream multiplication, we can define the *stream semantics* of $\mathcal{L}$ (Figure 9).

Example 5.9 (Matrix multiply)**.** Suppose the streams $x : a \to_s b \to_s K$ and $y : b \to_s c \to_s K$ represent two matrices. Recall that their matrix product contraction expression is $\Sigma_b(\Uparrow_c x \cdot \Uparrow_a y)$. Note that the two expanded subexpressions must have the same shape $[a, b, c]$ in order for the product to be well-typed. Since b is contracted, the *stream type* of the result expression is $a \to_s * \to_s c \to K$, while its shape is $[a, c]$.

5.3 Stream Evaluation

In this final section on the stream semantics, we define the evaluation function $[\![-]\!] : \mathcal{S} \to \mathcal{T}$. This makes formal the intuition we gave before, which is that the meaning of a stream is the sum of its indexed values at each ready state. First we clarify the set over which we sum, then define the sum itself. This requires some care because streams can be nested and can have dummy indices corresponding to attributes that have been contracted.

Definition 5.10 (Finite Streams)**.** A state r is *reachable from* q, written $q \to^* r$, if $r = \delta^k(q), k \geq 0$. A state r such that $r = \delta(r)$ is called *terminal*. If the set of states reachable from q contains a terminal state then it is necessarily finite, and we say that the stream q is finite.

When R is a semiring, there is a natural way of interpreting a pair $(i, v) \in I_a \times R$ as a function of type $I_a \to R$: let $i \mapsto v$ denote the *singleton function*

$$(i \mapsto v)(j) = \begin{cases} v & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

Note that functions from $I_a \to R$ themselves form a semiring under pointwise addition and multiplication, so we can define the following recursive stream evaluation function.

Definition 5.11 (Stream Evaluation: $[\![-]\!]$)**.** Suppose $q \in \mathcal{S}_S$. There are two cases to consider for the type of q:

- $q : a \to_s R$. Then we define $[\![q]\!] = \sum_{r:\, q \to^* r} (\text{index}(r) \longmapsto \text{ready}(r) \cdot [\![\text{value}(r)]\!])$.
- $q : * \to_s R$. Then we define $[\![q]\!] = \sum_{r:\, q \to^* r} (\text{ready}(r) \cdot [\![\text{value}(r)]\!])$.

As a base case, for $v \in R$ we define $[\![v]\!] = v$.

If q has shape S, the resulting function is the curried form of a K-relation of shape S.

Example 5.12 (Sparse Matrix)**.** Recall Figure 8 depicting the nested stream q_1. Its evaluation is a function of type $I_a \to I_b \to \mathbb{N}$, whose value we derive now. First note that the reachable states that are ready for the outer stream are q_1 and q_3, so the value of the stream will be $[\![q_1]\!] = (a_1 \mapsto [\![\text{value}(q_1)]\!]) + (a_3 \mapsto [\![\text{value}(q_3)]\!])$. The first row stream has value $[\![\text{value}(q_1)]\!] = (b_2 \mapsto 2) + (b_3 \mapsto 3)$, and similarly $[\![\text{value}(q_3)]\!] = b_1 \mapsto 1$. Combining these expressions gives the result of $[\![q_1]\!]$, which is a function of type $I_a \to I_b \to \mathbb{N}$ or equivalently a K-relation of type $I_{\{a,b\}} \to \mathbb{N}$.

5.4 Efficient Sparse Computation with Streams

The indexed stream definition can be thought of as an abstract data type specification. Any practical data structure supporting stateful in-order iteration can implement the specification. The semantic framework then enables composition with other data structures to compute arbitrarily complex contraction expressions. In the next section, we will describe our compiler that implements this framework. In the current section, we give one example from matrix algebra and one from relational algebra to further discuss how we approach the issues set out in Section 2.

5.4.1 Matrix Attribute Ordering. Using the example of matrix multiplication, we will discuss how the asymptotic complexity of evaluating a contraction expression using nested indexed streams depends on the chosen attribute ordering. Two strategies for matrix multiplication are the *inner-product* method and the *linear combination of rows*. For the inner product, we have two streams with shapes $x : [a, c]$ and $y : [b, c]$. The contracted index is c, the innermost one, and the contraction expression is $e_1 = \Sigma_c\ (\Uparrow_b x) \cdot (\Uparrow_a y)$. Iteration proceeds across $I_a \times I_b$, the output shape, and an inner loop computes the inner-product of the corresponding row and column of x and y across I_c.

For linear combination of rows, we compute on the two streams $x : [a, b]$ and $y : [b, c]$. The expression is $e_2 = \Sigma_b\ (\Uparrow_c x) \cdot (\Uparrow_a y)$. This algorithm is called linear combination of rows because

the second level of iteration, which traverses I_b, will simultaneously iterate across a particular row of x and the rows of y. This yields one row of output that is a linear combination of rows from y.

In the case where each stream encodes a sparse matrix with n non-empty rows and k non-empty values within a row, evaluating e_1 will involve $O(n^2k)$ transitions. On the other hand, e_2 traverses the rows of x, the intersection of k elements from each row with the rows of y, and finally one row of y, which gives a bound of $O(nk^2)$ assuming the intersection can be computed in time $O(k)$. Sparse data structures can typically perform the intersection in $O(k)$ or $O(k \log k)$, so the latter algorithm is asymptotically faster when $k \in o(n)$.

5.4.2 Worst Case Optimal Joins. Traditional *pairwise* join plans compute complex joins by merging together two relations at a time. This is akin to unfused execution of a contraction expression. The loop structure of a nested indexed stream, however, exactly mirrors the `GenericJoin` algorithm [Ngo et al. 2014], an algorithm that solves for one attribute at a time. This approach discards attribute values that cannot correspond to a solution by considering all relations that involve that attribute at the same iteration level. This approach can have asymptotically superior performance on various join queries. As long as the underlying data structures implement logarithmic or constant time access to a given index and its value, our multiplication operation implements the correct multiway intersection described in this prior work, and hence code generated from the contraction expression meets the worst-case optimal performance bound on arbitrary queries.

6 INDEXED STREAM CORRECTNESS THEOREM

In this section, we give an overview of the proof that the stream operators, such as multiply (defined in Section 5.1.1) and contraction (Section 5.1.2), are correctly defined. In particular, we want to know that if we take the product of two streams using the stream multiplication operator and evaluate the resulting stream (as formally specified in Section 5.3), the evaluation really is the product of the evaluations of the two initial streams. Concretely,

Theorem 6.1. *The function $[\![-]\!] : \mathcal{S} \to \mathcal{T}$ is a homomorphism; that is, for all strictly monotonic lawful streams (see Section 6.2) $q, r \in \mathcal{S}$ of the same shape:*

$$\begin{aligned} [\![q + r]\!] &= [\![q]\!] + [\![r]\!] \\ [\![q \cdot r]\!] &= [\![q]\!] \cdot [\![r]\!] \\ [\![\Sigma_a q]\!] &= \sum_{i \in I_a} [\![q]\!](i) \\ [\![\Uparrow_a q]\!](i) &= [\![q]\!] \end{aligned}$$

Or in other words, $[\![[\![e]\!]_c^{\mathcal{S}}]\!] = [\![e]\!]_c^{\mathcal{T}}$ for all $e \in \mathcal{L}$. We formalize and prove this result [Kovach et al. 2023] using the Lean theorem prover [Moura et al. 2015]. However, we give the relevant definitions here and explain the intuition behind them.

6.1 Lawful Streams

When the `skip` function for a stream is called, it should not affect the evaluation of the stream at any indices after the destination index. We formalize this condition as follows: whenever $(i, r) \in I \times \{0, 1\}$ and $j \in I$ satisfies $(i, r) \leq (j, 0)$ (using the lexicographic ordering on $I \times \{0, 1\}$), skipping to (i, r) does not affect the evaluation at j; i.e., $[\![\text{skip}(q, (i, r))]\!](j) = [\![q]\!](j)$. When this condition is met, we say that the stream is lawful. We formally verify that the skip functions of addition, multiplication, contraction, etc. are lawful assuming the constituent streams are lawful.

```
example (a b : ι₁ →ₛ ι₂ →ₛ R)
  (j : ι₂) : eval (∑ᵢ (a * b)) () j =
    ∑ i in (eval a * eval b).support,
    (eval a i j * eval b i j) :=
by rw Eval.contract'; simp
```

```
example (a b c d : ι₁ →ₛ ι₂ →ₛ ι₃ →ₛ R)  :
  eval (a * (b + c) * d) =
  (eval a) * ((eval b) + (eval c)) * (eval d) :=
by simp
```

Fig. 10. Example proofs automatically synthesized by Lean. In each theorem statement, the LHS is the evaluation of a series of combinators applied to streams, while the RHS consists of various operations on finitely supported functions applied to the evaluations of streams. Notice that nested streams enable us to use the same basic lemmas for vectors, matrices, and more generally, rank n tensors for any n

6.2 Strict Monotonicity

Recall that streams are monotone when $\text{index}(q) \leq \text{index}(\text{skip}(q, (i, r)))$ for all q, i, r. We say that streams are *strictly monotone* when, in addition to being monotone, the stream moves to a state with a strictly larger index when advanced from a ready state.

This requirement is necessary for multiplication because the multiplication combinator eagerly emits a value and advances its constituent streams when their indices match. If a constituent stream emits more than one value associated with the same index, this behavior is incorrect. We formally prove in Lean that addition and multiplication preserve strict monotonicity, so that these combinators may indeed be composed arbitrarily.

6.3 Lean Formalization

We formalize the correctness proofs of our stream model in the Lean theorem prover [Moura et al. 2015]. In particular, we formally verify the soundness of multiplication, binary addition, and contraction. The formal model of streams implemented in Lean closely follows the model presented in this paper with only a few minor changes. First, evaluations of finite streams produce *finitely supported* functions; this ensures that evaluation is always well-defined, even of contracted streams. Moreover, the stream functions are allowed to be partial functions, defined only so long as certain conditions are met. This enforces conditions that are implicitly assumed by the compiler; for example, since calling value when ready is false may produce an out of bounds error, in the verified model, value takes as a parameter a proof that ready is true.

Because these proofs are valid even in the nested case, and because lawful streams are closed under the basic stream operations, we can easily produce proofs that the evaluations of complicated expressions of streams are sound (see Figure 10). These verified proofs give confidence that the underlying model is correct. Combining this with a verification of the compilation step, which translates the streams to imperative code, would result in an end-to-end verified compiler. We leave the verification of the compilation step to future work.

7 THE ETCH COMPILER

We implement a compiler for the contraction language that we call `Etch`[1]. `Etch` transforms contraction expressions from $\mathcal{L}$ into a C-like intermediate representation (IR) that can be compiled by a standard C compiler. In Section 8, we show that the resulting programs match performance of hand-written code for sparse matrix computations and relational queries. The core functionality needed for these benchmarks is implemented in under 540 lines of Lean code.

We describe how the design of the compiler was directly derived from the stream model. We encode indexed streams using concrete *syntactic* indexed streams where each component of an indexed stream translates directly to a program fragment in our C-like IR. Together, these fragments can be compiled into a single program that evaluates a corresponding indexed stream. Each

[1] `Etch` is available at https://github.com/kovach/etch/

```
inductive P
| seq       : P → P → P
| while     : E Bool → P → P
| branch    : E Bool → P → P → P
| skip      : P   -- no-op; unrelated to stream skip
| decl      : Var α → E α → P
| store_var : Var α → E α → P
| store_mem : Var (ℕ → α) → E ℕ → E α → P

inductive E : Type → Type 1
| var    : (v : Var α) → E α
| access : Var (ℕ → α) → E ℕ → E α
| call {α} (op : Op α) (args :
    (i : Fin op.arity) → E (op.argTypes i)) : E α
```

Fig. 11. A simple imperative language P and expression language E. An expression is a variable, array access, or fully-applied function call.

```
structure Op (α : Type _) where
  arity : ℕ
  argTypes : Fin arity → Type
  spec : ((n : Fin arity) → argTypes n) → α
  opName : String

def Op.add [Tagged α] [Add α] : Op α where
  argTypes := ![α, α]
  spec := λ a => a 0 + a 1
  opName := tag_mk_fun α "add"
```

Fig. 12. The user-extensible type of custom operations, allowing users to embed external C definitions within Etch programs. Below: the definition of add in Etch's library. Users can define custom operators in the same way; add is unprivileged.

operation on streams translates naturally to an operation on syntactic streams. Additionally, we have implemented a variety of primitive syntactic indexed streams that allow iteration over different data structures, demonstrating that the system supports data structure abstraction.

7.1 Target Language

Our compiler generates code in a small imperative language that we call P, whose full definition is given in Figure 11. This language supports while loops, if-statements, and assignments to local variables and arrays. It maps directly to C code.

The expression language used within P is parametrized by an open set of *user defined operations*. These allow users to extend the expression language of P with arbitrary C procedures. A user must give an Op definition (Figure 12) that declares the external code and assigns it a type and functional specification. Users are then free to use the operation within their contraction expressions. We use this extension mechanism ourselves to implement all scalar operations needed to define indexed streams, including the semiring operations and index comparisons needed for multiplication and contraction.

7.2 Syntactic Indexed Streams

In the same way that a general program is a syntactic encoding of a computation, a *syntactic indexed stream* is a syntactic encoding of an indexed stream. The key idea is to replace each part of an indexed stream with a syntactic equivalent (Figure 13). We discuss two challenges in this section: (1) deriving this syntactic representation of indexed streams, and (2) encoding the functions used in our previous definitions of streams and stream operators as operations on machine states.

```
structure Stream
(ι : Type _) (α : Type _) where
  S     : Type
  value : S → α
  skip0 : S → E ι → P
  skip1 : S → E ι → P
  ready : S → E Bool
  index : S → E ι
  valid : S → E Bool
  init  : Name → P × S

infixr:25 " →ₛ " => Stream
```

Fig. 13. The fields index, value, ready, and skip0/skip1 correspond to syntactic representations of the stream functions. The fields init and valid initialize and check for termination.

Deriving Syntactic Indexed Streams. We model the runtime state of a C program as consisting of a heap and a set of local variables. We refer to the set of all possible configurations of this state as S. A syntactic indexed stream encodes a stream with S as its state space. We define a semantic function $\texttt{run} : \texttt{P} \to S \to S$ that translates a program in P to a function $S \to S$ that transforms the initial memory state into the final state. Similarly, we have a function $\texttt{eval} : \texttt{E}\ \alpha \to S \to \alpha$ which translates an α-typed expression and a machine

```
def S.mul [HMul α β γ]
(a : ι →s α) (b : ι →s β) : (ι →s γ) where
  S := a.S × b.S
  value p   := a.value p.1 * b.value p.2
  skip0 p i := a.skip0 p.1 i;; b.skip0 p.2 i
  skip1 p i := a.skip1 p.1 i;; b.skip1 p.2 i
  ready p   := a.ready p.1 * b.ready p.2 *
               (a.index p.1 == b.index p.2)
  index p   := .call .max
                 ![a.index p.1, b.index p.2]
  valid p   := a.valid p.1 * b.valid p.2
  init      := seqInit a b

instance [HMul α β γ] :
    HMul (ι →s α) (ι →s β) (ι →s γ) := ⟨S.mul⟩
```

Fig. 14. Our implementation of multiplication for stream IR objects. This definition generalizes to arbitrary nested streams via typeclass search. Compare to the definition in Section 5.1.1.

```
instance base_var [Tagged α] [Add α] :
    Compile (Var α) (E α) where
  compile _ l r := l.store_var (E.var l + r)

instance step [Compile L R] :
    Compile (lvl ι L) (ι →s R) where
  compile n l r :=
    let (init, s) := r.init n
    let (push, pos) := l.push (r.index s)
    init;; P.while (r.valid s)
      (.branch (r.ready s)
        (push;; compile n pos (r.value s);;
         r.skip1 s (r.index s))
        (r.skip0 s (r.index s)))
```

Fig. 15. The core code generation functions. The base case stores an expression into a variable. The inductive case emits a loop to evaluate the outer stream. Any nested streams are handled by the recursive call to `compile`.

state to an α-typed value. Each component of the definition `Stream` is a simple translation of the corresponding stream component. The `run`/`eval` functions allow us to formally relate components of a syntactic indexed stream to the indexed stream it implements.

In order to slightly simplify the compilation function, we split skip : $S \to I \times \{0,1\} \to S$ into two functions `skip0`, `skip1` : $S \to I \to S$ corresponding to either value of $\{0,1\}$. Because states are no longer first class, we use `init` to produce the initial state and `valid` as a termination check. The local variables used by a particular stream are passed to each field as the S parameter; that is, the S component is a static representation of the state space of the stream.

Encoding Primitive Streams and Operators. Encoding the state of a sparse or dense indexed stream (Example 5.2) is straightforward; we assume that streams fit in memory, so we simply need an integer counter to track our position within the stream. In order to define the sparse vector, it is necessary that our machine implementation of the indexing set I supports $(\leq, <)$ operations. For the dense vector, we require the stronger condition that I is encodable as an integer since index values are not explicitly stored but rather used to access locations in the value array.

Some streams can be implicitly represented. Instead of an in-memory representation, some or all of their components are encoded as procedures. For example, the expansion operator does not store multiple copies of its value in memory, but rather returns the constant value each time value is invoked. We use this approach to encode user-defined functions and relations as streams.

The key stream composition operator is multiplication. Its definition on syntactic indexed streams is given in Figure 14 and closely mirrors the earlier definition (Section 5.1.1) on indexed streams. As a convenience, our compiler infers the needed replication operators whenever multiply is used.

7.3 Compilation

The core compilation function, `compile out v`, takes a destination `out` and a value `v` and returns code that computes the value and accumulates it into the destination. We can summarize the intended effect of compile by the following Hoare triple: $\{\texttt{out} \mapsto v\}$ `(compile out q)` $\{\texttt{out} \mapsto v + [\![q]\!]\}$.

This compilation approach is inspired by *destination passing style* [Minamide 1998; Shaikhha et al. 2017], a technique for generating imperative code that helps to avoid redundant allocation and copying. The `compile` function requires that `v` is a syntactic indexed stream or, in the base case, a scalar expression. The type of the destination must be compatible: if `v` is a scalar, it should be a pointer to a scalar accumulator, or if `v` is a stream, it must implement a function that maps an index expression to a sub-destination.

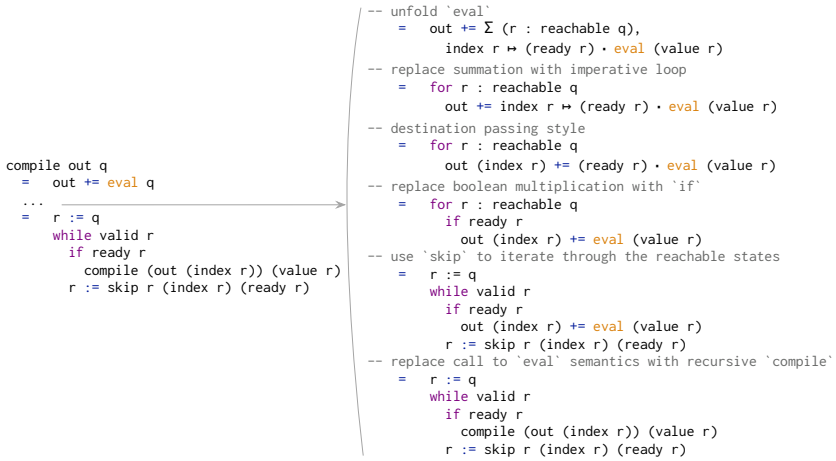

Fig. 16. A simplified version of the code generation function Figure 15 is shown on the left, along with an equational derivation for it on the right.

The compilation function for indexed streams is shown in Figure 15, and a step-by-step intuitive derivation for `compile`, starting from the stream evaluation definition, is given in Figure 16. This can be understood as a code template for a `while` loop. The components of `r`, a syntactic indexed stream, are spliced into the body of the loop where appropriate. Inside the `ready` check, compile is invoked recursively with the current value of the stream and a sub-destination. As in Definition 5.11, there is another almost identical case for contracted streams.

Contraction expressions in `Etch` are parametrized by the choice of scalars, global attribute ordering, and data structure choices. As long as a semiring has a runtime representation and implementations of $(0, 1, +, \cdot)$, it can be used as the scalars for a contraction expression. Our evaluation makes use of boolean, floating point, and $(\min, +)$ scalars.

In addition to a stream expression, compilation requires a global ordering of attributes. This ordering controls the order of loops in the output loop nest. In our evaluation, a very simple heuristic (putting primary keys first when possible) achieves strong performance.

Finally, users may provide their own data structures by implementing the syntactic stream and destination interfaces. As demonstrated by Chou et al. [2018], many storage formats can be decomposed by level. We provide a compositional implementation of the compressed and dense level formats. Our sparse level can be traversed using either naïve iteration or binary search. Additionally, our evaluation uses a hash table format for database query output.

8 EVALUATION

To demonstrate the practical usefulness of indexed streams as an operational model and an intermediate representation, we (1) exhibit a mechanical proof that the stream semantics is correct and (2) evaluate key performance properties of the `Etch` compiler.

Correctness. We prove the correctness of the indexed stream operations by a Lean-checked formal proof that we describe in Section 6. The compositionality of indexed streams made correctness properties easy to state and feasible to prove. We observe that the composition operators are relatively easy to define and understand, while the lawfulness and monotonicity properties (Section 6) are tricky but only need to be checked for *individual* streams. They provide a template for future `Etch` programmers to check: if their new data structure implements skip and meets the necessary

monotonicity properties, they can be sure that composing this data structure with pre-existing nested streams will create no new problems.

Performance. Using the indexed stream IR, `Etch` is implemented in only 540 lines of Lean code , making it about two orders of magnitude smaller than the TACO compiler. It supports everything that the original TACO system [Kjolstad et al. 2017] supported, and it additionally supports format abstractions, iteration reordering [Kjolstad et al. 2019], and user-defined functions [Henry et al. 2021]. Moreover, `Etch` supports relational algebra.

Section 2 describes several challenges of efficient execution of contraction expressions that indexed streams are designed to address. We evaluate how well indexed streams address the following five efficiency goals using both formal and experimental results:

Fusion Indexed streams were expressly designed to fuse arbitrary additive and multiplicative operations across data structures. Section 8.1 shows empirical evidence that the performance of MTTKRP and other kernels match the asymptotic performance of the TACO compiler, which generates fused code [Kjolstad et al. 2017]. The triangle join query in Section 8.2 shows that `Etch` can achieve the worst-case optimal complexity where non-fused implementations do not.

Hierarchical iteration Nested streams (Section 5.2) formalize hierarchical iteration. Most subsequent experiments rely on hierarchical iteration. As an example, TPC-H Query 9 (Figure 19) skips rows whose names do not include the word "green" prior to performing an expensive join. In the filtered SpMV in Figure 21, the filter cuts work on matrix rows that do not satisfy the filter condition.

Control over iteration order Each indexed stream embodies a particular iteration order. We describe in Section 5.4 how two different matrix multiplication stream orders produce algorithms with distinct asymptotic complexities [Ahrens et al. 2022]. The ability to control column order is also crucial for good performance in relational queries, and we show in Section 8.2 that code generated by `Etch` is competitive with mature analytical database software.

Data structures The indexed stream interface is consistent with a wide variety of data structures. Our experiments in this section test both dense and compressed data structures.

Good code quality We demonstrate in this section that indexed streams can be compiled to efficient code with no overhead coming from their high level of abstraction. Section 8.1 shows that `Etch` is competitive with TACO [Kjolstad et al. 2017] for tensor contractions. Section 8.2 shows that `Etch`-generated code outperforms DuckDB [Raasveldt and Mühleisen 2019] and SQLite [Hipp 2020] on all the queries we test. These experiments are not intended to show that the `Etch` compiler is better or worse than other systems (each with their own unique capabilities), but to demonstrate that indexed streams can be compiled to efficient code.

The following subsections provide the empirical evidence that we used to form the above conclusions. We ran experiments on a Linux machine with an Intel i7-8700 processor with 16 GB of memory and swap disabled. The reported data were collected from at least 25 runs where we took the average.

8.1 Sparse Tensor Algebra

The TACO compiler for sparse tensor algebra [Kjolstad et al. 2017] has been demonstrated to compile sparse linear/tensor algebra expressions with comparable performance to hand-optimized code in libraries such as Intel MKL [Intel. 2009] and SPLATT [Smith et al. 2015]. We compare our performance to TACO on synthetic matrices with different percentages of sparsity. We use

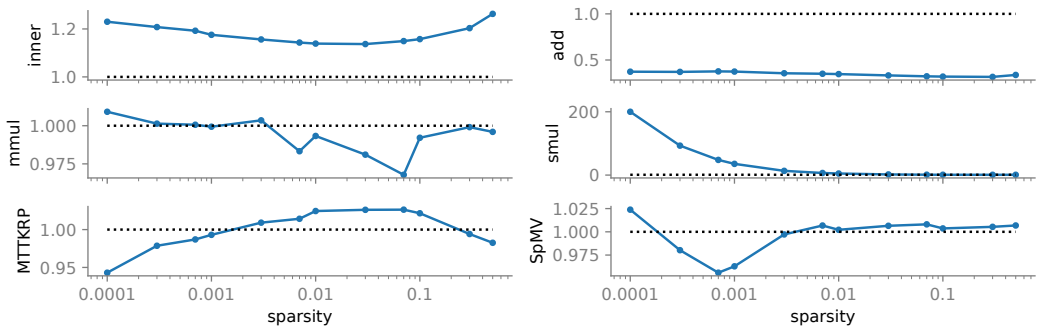

Fig. 17. Sparse tensor algebra expressions, with Etch in blue and TACO in stippled black. Depicts speedup relative to TACO on a linear scale.

synthetic matrices instead of a dataset such as the SuiteSparse repository [Davis and Hu 2011], as they let us sweep over different sparsity percentages to demonstrate that Etch can generate algorithms with suitable asymptotic complexity.

Figure 17 plots the performance of sparse tensor algebra expressions generated by Etch (in solid blue) as speedup normalized to TACO (stippled black). The expressions are taken from the original TACO paper [Kjolstad et al. 2017]. MTTKRP is the matricized tensor times Khatri–Rao product, `inner` is a matrix inner product, `mmul` is CSR matrix–matrix multiplication, and `smul` is DCSR matrix–matrix multiplication. The logarithmic x-axis contains matrices of increasing sparsity. The results show that Etch is within a factor of 0.75 to 1.2 of the runtime of TACO (lower is better), on all cases except for sparse matrix addition and `smul`. For sparse matrix addition, the evaluation demonstrates correct asymptotic behavior relative to TACO. Etch suffers from a 2–3× constant factor difference in performance because TACO uses a more refined method for loop generation. The `smul` expression is significantly faster than TACO due to our use of binary search in the skip function. This gives an asymptotic improvement with respect to sparsity level.

Finally, we use Etch to generate code for both the inner product and linear combination of rows matrix multiplication algorithms from Section 5.4. We run them on a $10\,000 \times 10\,000$ matrix with 200 000 nonzeros and the asymptotic complexity advantage of the linear combination of rows algorithm led it to be 40 times faster (9.77 s vs. 0.24 s).

8.2 Relational Algebra

We compare the performance of Etch on relational algebra expressions to DuckDB [Raasveldt and Mühleisen 2019] and SQLite [Hipp 2020]. Both are efficient in-process SQL database engines: SQLite is a traditional row-oriented DBMS while DuckDB is designed for online analytical processing (OLAP). DuckDB implements an optimized vectorized query execution engine, while Etch compiles queries to machine code. Kersten et al. [2018] found that vectorized and compiled query execution methods perform similarly on a wide range of queries, so DuckDB is a reasonable baseline. See Figure 18 for an overview of other differences between the three systems. The experiment is designed to test two claims:

(1) the Etch compilation method can generate competitive query evaluation code without any abstraction overhead, and
(2) we empirically meet the worst-case optimal asymptotic complexity for a nontrivial query.

In particular, we do not intend to evaluate Etch as a database management system. Production database software has many additional design constraints, such as transaction processing, whereas

System	Execution model Data model
DuckDB	interpreted (vectorized) column-based
SQLite	interpreted (bytecode) row-based
Etch	compiled (Clang -O3) column-based

Fig. 18. Relational algebra systems used in evaluation.

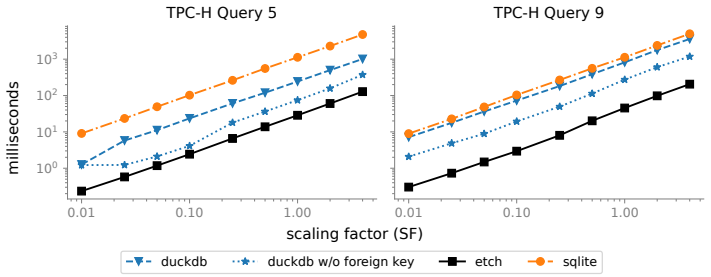

Fig. 19. TPC-H queries. At SF=1, queries 5 and 9 use 7.7 and 8.5 million rows of data respectively; the largest join in both queries is between two relations with 1.5 and 6 million rows.

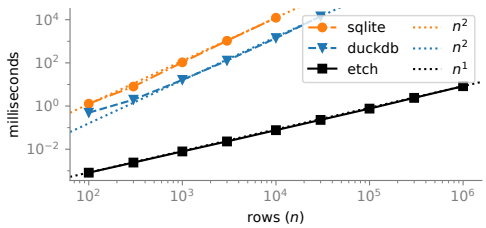

Fig. 20. Triangle query [Ngo et al. 2014]

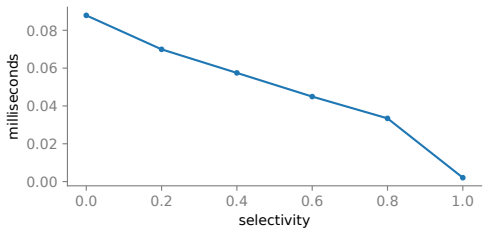

Fig. 21. Filtered SpMV

in this experiment Etch uses static data structures optimized for analytics of data sets at rest. To make the comparison more fair, we (a) restrict execution to a single thread, (b) load all data into memory, (c) delete columns irrelevant to the query, (d) prepare queries before repeated execution, and (e) add indices with the same column ordering as Etch. All these changes advantage SQLite, changes (b)–(d) advantage DuckDB, and change (a) advantages Etch.

We evaluate three relational algebra queries: two queries in the standard TPC-H benchmark suite [TPCH 2022] and a triangle cyclical query. We manually translate TPC-H queries 5 and 9 from SQL to contraction expressions. Q5 joins together seven distinct relations with 10 columns, while Q9 joins six relations with 12 columns. When doing so, we choose the particular data formats per table (dense vs. sparse columns) and a column ordering that is appropriate for the query. These decisions are analogous to those made by a query optimizer in a DBMS. Additionally, Q9 requires a timestamp-to-year conversion function not built into Etch, so we define a custom operator based on `gmtime_r()`. Q9 also requires a substring matching function, which we encode as a boolean-valued indexed stream. These extensions do not require modifications to the Etch compiler or library.

Figure 19 shows the relative performance on TPC-H queries 5 and 9. The size of the TPC-H data set can be linearly scaled by a scaling factor (SF). Prior work [Boncz et al. 2014] has identified data access locality and join performance as the bottlenecks of Q5 and Q9 respectively, two areas where Etch's data structures and fused join algorithms let it outperform the general-purpose databases. Figure 19 shows that Etch exceeds the performance of SQLite by at least 24× and DuckDB by 1.6× across a wide range of scales. We include a variant of DuckDB with foreign key constraints removed to encourage the use of Hash Joins over Index Joins, which improves on its default performance.

The log–log plot in Figure 20 shows the systems' performance on the triangle query $\sum_{a,b,c} R(a,b) \cdot S(b,c) \cdot T(c,a)$, a motivating example for multiway join methods. Ngo et al. [2014] proved that

fused multiway joins can solve this family of query instances in $\Theta(n)$ time,[2] the best possible, while any pair-wise join method must iterate over an intermediate result of size $\Theta(n^2)$. In Figure 20, we empirically show that the fused indexed stream scales linearly while SQLite and DuckDB scale quadratically.

8.3 Fused Tensor and Relational Algebra

In our final experiment, we demonstrate the benefits of fusing across tensor and relational operations. We designed a small expression that combines a sparse matrix-vector multiplication with a relational selection/filter on the vector entries. Such an expression could, for example, be used in a PageRank computation [Page et al. 1999] where we want to leave out pages with a low score. Figure 21 shows that the time to compute the filtered-vector SpMV goes to zero as the filter selectivity approaches 100%. This experiment demonstrates the benefit of fusing sparse linear and relational algebra.

9 RELATED WORK

This paper proposes a formal indexed stream operational semantics for contraction expressions, a proof of its correctness, and a compiler. We discuss prior work on contractions, related DSL systems and compilers, and work on stream languages.

Contraction Formulations. Sparse tensor algebra [Kjolstad et al. 2017], databases [Shaikhha et al. 2018], factorized and marginalized probability distributions [Abo Khamis et al. 2016; Aji and McEliece 2000], weighted graphs [Mattson et al. 2013], and formal languages [Elliott 2019] can all be represented as vectors or higher-order sparse tensors by choosing the underlying set of scalars appropriately. Such representations are also conducive to algebraic restatements of many algorithms [Abo Khamis et al. 2016]. Moreover, these restatements can enable application of specialized fusion techniques such as worst-case optimal join methods [Ngo et al. 2018; Schleich et al. 2019; Veldhuizen 2014] and factorization techniques that perform asymptotically better on some queries. By introducing a flexible and composable abstract data type for such general sparse computations, we hope to make these advanced algorithms more widely accessible across domains.

Compilers for Sparse Tensor Algebra. Recent work has showed how to compile [Bik et al. 2022; Kjolstad et al. 2017; Tian et al. 2021] arbitrary sparse tensor algebra expressions to fused code on sparse and dense data structures. Our work generalizes these compilers by also effectively supporting relations. Chou et al. [2018] shows how to extend sparse tensor algebra compilers with new sparse data structures and Henry et al. [2021] shows how to extend them with user-defined functions. The indexed stream model provides a rigorous method to ensure that new data structures and functions are valid without sacrificing performance. Finally, Kjolstad et al. [2019] introduce compiler optimizations to reorder iteration and introduce temporaries, while Senanayake et al. [2020] introduce tiling and parallelizing optimizations. Our indexed streams can express iteration reordering and temporaries, but we leave tiling and parallelization as future work.

Execution Systems for Relational Algebra. Execution systems for relational algebra [Codd 1970] used in database management systems have flourished since INGRES [Held et al. 1975] and System R [Astrahan et al. 1976]. More recent work, such as the HyPer DBMS [Kemper and Neumann 2011], shows the benefits of code generation through composing hand-written templates for different types of algorithms. Several libraries also provide relational algebra support for data retrieval and analysis, including SQLite [Hipp 2020] and Python pandas [McKinney 2010]. Recently, Aberger et al. [2017] showed how to generate fused code for inner join expressions. Finally, researchers

[2]We evaluate on three copies of the relation $\{0\} \times [n] \cup [n] \times \{0\}$, which has an output size of $\Theta(n)$.

have shown how to reduce dense tensor algebra to relational algebra [Aberger et al. 2018; Yuan et al. 2021]. Our work can handle both dense and sparse tensor algebra and relational algebra.

Compilers for Array Programs. There is a large body of work on functional languages that model dense arrays as functions, including Halide [Ragan-Kelley et al. 2012], Futhark [Henriksen et al. 2017], Lift [Steuwer et al. 2017]. These systems use pure functional representations to support equational reasoning and optimization for array programs. Liu et al. [2022] express low-level optimizations on dense array programs as verified source-to-source transformations of a high level functional language. Shaikhha et al. [2022] exhibit a functional language for sparse tensors and relations. Our system has several salient differences: we produce fused code automatically; we provide a less restrictive data structure abstraction; we guarantee in-order iteration when desired, which has positive performance consequences; and we mechanically verify our semantics.

Stream Programs and Stream Fusion. Stream-fusion [Coutts et al. 2007; Kiselyov et al. 2017], one-dimensional stream-based programming models [Halbwachs et al. 1991; Thies et al. 2002], and synchronous dataflow languages [Caspi and Pouzet 1995; Colaço et al. 2006] have been used in a wide range of applications from embedded signal-processing to database query evaluation. Our work describes higher-dimensional streams that are augmented with an increasing index value. Indexed streams are related to *hierarchical maps* in the same way that standard streams are related to *lists*, so they are composable in more general ways.

Nested Data Parallelism. Prior work on nested data parallelism [Blelloch 1992] has described systems for extracting parallelism from computations on nested, irregular data structures. This line of work has provided a language-based framework for reasoning formally about time complexity [Blelloch and Greiner 1996], integration with Haskell [Chakravarty et al. 2007; Jones 2003], and applications in teaching the design of parallel algorithms. This work has limited application to problems involving simultaneous iteration of multiple sparse data structures with differing structure, as is the case in relational processing and tensor algebra. In future work, we hope to investigate similar reasoning tools and automatic parallelization for our work.

10 CONCLUSION

We introduced the indexed stream operational semantics for contraction expressions. Our model hides details of fusion and sparse data structures beneath a high-level expression language. The stream model has benefited from several iterations of design driven by our mechanical proof effort and the desire to match existing high-performance systems. Our work towards formalization, first on paper and then in Lean, revealed many conceptual issues and continually forced us to simplify the model. As future work, we plan to build on the Lean formalization of our correctness theorem to mechanically verify the `Etch` compiler, providing a stronger correctness guarantee and a modular proof for others to extend. We also plan to design and verify a scheduling language for `Etch`.

ACKNOWLEDGMENTS

We thank Olivia Hsu, Shiv Sundram, Rohan Yadav, Bobby Yan, Manya Bansal, Ajay Brahmakshatriya, and Matthew Sotoudeh for feedback on drafts. We also thank Kyle Miller for sharing explanations of Lean techniques. Finally, we thank our anonymous reviewers and our shepherd, Adam Chlipala, for invaluable comments and suggestions. This work was in part supported by the National Science Foundation under Grant CCF-2143061 and CCF-2216964. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

SOFTWARE AVAILABILITY

The latest version of the `Etch` compiler is available at https://github.com/kovach/etch/. Benchmarking code and formal proofs used for evaluation are available online [Kovach et al. 2023].

REFERENCES

Christopher R. Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. 2018. LevelHeaded: A unified engine for business intelligence and linear algebra querying. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 449–460. https://doi.org/10.1109/ICDE.2018.00048

Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)* 42, 4 (2017), 1–44. https://doi.org/10.1145/3129246

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*. ACM, 13–28. https://doi.org/10.1145/2902251.2902280

Peter Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. 2022. Autoscheduling for Sparse Tensor Algebra with an Asymptotic Cost Model. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 269–285. https://doi.org/10.1145/3519939.3523442

Srinivas M. Aji and Robert J. McEliece. 2000. The generalized distributive law. *IEEE Transactions on Information Theory* 46, 2 (March 2000), 325–343. https://doi.org/10.1109/18.825794

Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. 1976. System R: Relational Approach to Database Management. *ACM Trans. Database Syst.* 1, 2 (June 1976), 97–137. https://doi.org/10.1145/320455.320457

Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler Support for Sparse Tensor Computations in MLIR. *ACM Trans. Archit. Code Optim.* 19, 4, Article 50 (Sept. 2022), 25 pages. https://doi.org/10.1145/3544559

Guy E Blelloch. 1992. *Nesl: A Nested Data-Parallel Language.* Technical Report CMU-CS-92-103. Carnegie Mellon Univ.

Guy E Blelloch and John Greiner. 1996. A provable time and space efficient implementation of NESL. *ACM SIGPLAN Notices* 31, 6 (1996), 213–225. https://doi.org/10.1145/232629.232650

Peter Boncz, Thomas Neumann, and Orri Erling. 2014. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Performance Characterization and Benchmarking*, Raghunath Nambiar and Meikel Pöss (Eds.). Springer International Publishing, Cham, 61–76. https://doi.org/10.1007/978-3-319-04936-6_5

Paul Caspi and Marc Pouzet. 1995. A Functional Extension to Lustre. In *Eighth International Symposium on Languages for Intentional Programming (ISLIP '95)*, M. A. Orgun and E. A. Ashcroft (Eds.). World Scientific, Sydney, Australia.

Manuel MT Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. 2007. Data parallel Haskell: a status report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming (DAMP '07)*. 10–18.

Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 123 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276493

Edgar Frank Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (June 1970), 377–387. https://doi.org/10.1145/362384.362685

Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. 2006. Mixing Signals and Modes in Synchronous Data-Flow Systems. In *Proceedings of the 6th ACM/IEEE International Conference on Embedded Software* (Seoul, Korea) *(EMSOFT '06)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/1176887.1176899

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion: From Lists to Streams to Nothing at All. *SIGPLAN Not.* 42, 9 (Oct. 2007), 315–326. https://doi.org/10.1145/1291220.1291199

Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25. https://doi.org/10.1145/2049662.2049663

Conal Elliott. 2019. Generalized Convolution and Efficient Language Recognition. (2019). arXiv:1903.10677 [cs.PL]

Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD–SIGACT–SIGART Symposium on Principles of Database Systems* (Beijing, China) *(PODS '07)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/1265530.1265535

Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous data flow programming language Lustre. *Proc. IEEE* 79, 9 (1991), 1305–1320. https://doi.org/10.1109/5.97300

Patrick Hall, Peter Hitchcock, and Stephen Todd. 1975. An Algebra of Relations for Machine Computation. In *Proceedings of the 2nd ACM SIGACT–SIGPLAN Symposium on Principles of Programming Languages* (Palo Alto, California) *(POPL '75)*.

Association for Computing Machinery, New York, NY, USA, 225–232. https://doi.org/10.1145/512976.512998

G. D. Held, M. R. Stonebraker, and E. Wong. 1975. INGRES: A Relational Data Base System. In *Proceedings of the May 19–22, 1975, National Computer Conference and Exposition* (Anaheim, California) *(AFIPS '75)*. Association for Computing Machinery, New York, NY, USA, 409–416. https://doi.org/10.1145/1499949.1500029

Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 556–571. https://doi.org/10.1145/3062341.3062354

Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, and Fredrik Kjolstad. 2021. Compilation of sparse array programming models. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–29. https://doi.org/10.1145/3485505

Richard D Hipp. 2020. SQLite. https://www.sqlite.org/index.html

Intel. 2009. Intel Math Kernel Library: Reference Manual.

Simon Peyton Jones. 2003. *Haskell 98 language and libraries: the revised report.* Cambridge University Press.

Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 195–206. https://doi.org/10.1109/ICDE.2011.5767867

Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask. *Proc. VLDB Endow.* 11, 13 (Sept. 2018), 2209–2222. https://doi.org/10.14778/3275366.3284966

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. ACM, 285–299. https://doi.org/10.1145/3093333.3009880

Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor Algebra Compilation with Workspaces. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. ACM/IEEE, 180–192. https://doi.org/10.1109/CGO.2019.8661185

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29. https://doi.org/10.1145/3133901

Scott Kovach, Praneeth Kolichala, Tiancheng Gu, and Fredrik Kjolstad. 2023. *Artifact for "Indexed Streams: A Formal Intermediate Representation for Fused Contraction Programs"*. https://doi.org/10.5281/zenodo.7809339

Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 38–49. https://doi.org/10.1109/ICDE.2013.6544812

Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022. Verified Tensor-Program Optimization via High-Level Scheduling Rewrites. *Proc. ACM Program. Lang.* 6, POPL, Article 55 (Jan. 2022), 28 pages. https://doi.org/10.1145/3498717

Tim Mattson, David Bader, Jon Berry, Aydin Buluc, Jack Dongarra, Christos Faloutsos, John Feo, John Gilbert, Joseph Gonzalez, Bruce Hendrickson, et al. 2013. Standards for graph algorithm primitives. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–2. https://doi.org/10.1109/HPEC.2013.6670338

Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman (Eds.). 56–61. https://doi.org/10.25080/Majora-92bf1922-00a

Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together at Last. *Proc. VLDB Endow.* 11, 1 (Sept. 2017), 1–13. https://doi.org/10.14778/3151113.3151114

Yasuhiko Minamide. 1998. A functional representation of data structures with a hole. In *Proceedings of the 25th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL '98)*. 75–84. https://doi.org/10.1145/268946.268953

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *International Conference on Automated Deduction*. Springer, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *International Conference on Automated Deduction*. Springer, 625–635. https://doi.org/10.1007/978-3-030-79876-5_37

Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case optimal join algorithms. *Journal of the ACM (JACM)* 65, 3 (2018), 1–40. https://doi.org/10.1145/3180143

Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew strikes back: New developments in the theory of join algorithms. *ACM SIGMOD Record* 42, 4 (2014), 5–16. https://doi.org/10.1145/2590989.2590991

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report 1999-66. Stanford InfoLab.

Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics (TOG)* 31, 4 (2012), 1–12. https://doi.org/10.1145/2185520.2185528

Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. 2019. A layered aggregate engine for analytics workloads. In *Proceedings of the 2019 International Conference on Management of Data*. 1642–1659. https://doi.org/10.1145/3299869.3324961

Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A Sparse Iteration Space Transformation Framework for Sparse Tensor Algebra. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 158 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428226

Amir Shaikhha, Mohammad Dashti, and Christoph Koch. 2018. Push versus pull-based loop fusion in query engines. *Journal of Functional Programming* 28 (2018). https://doi.org/10.1017/S0956796818000102

Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. 2017. Destination-Passing Style for Efficient Memory Management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing* (Oxford, UK) *(FHPC 2017)*. Association for Computing Machinery, New York, NY, USA, 12–23. https://doi.org/10.1145/3122948.3122949

Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. 2022. Functional collection programming with semi-ring dictionaries. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–33. https://doi.org/10.1145/3527333

Shaden Smith, Niranjay Ravindran, Nicholas D Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and parallel sparse tensor-matrix multiplication. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 61–70. https://doi.org/10.1109/IPDPS.2015.27

Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 74–85. https://doi.org/10.1109/CGO.2017.7863730

William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*. Springer, 179–196. https://doi.org/10.1007/3-540-45937-5_14

Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. 2021. A High-Performance Sparse Tensor Algebra Compiler in Multi-Level IR. (2021). arXiv:2102.05187 [cs.DC]

Transaction Processing Performance Council. TPCH. 2022. *TPC Benchmark H Standard Specification*. Revision 3.0.1. Transaction Processing Performance Council. https://www.tpc.org/tpch/

Todd L Veldhuizen. 2014. Leapfrog Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *Proc. 17th International Conference on Database Theory (ICDT)* (Athens, Greece). OpenProceedings.org, 96–106. https://doi.org/10.5441/002/icdt.2014.13

Binhang Yuan, Dimitrije Jankov, Jia Zou, Yuxin Tang, Daniel Bourgeois, and Chris Jermaine. 2021. Tensor Relational Algebra for Distributed Machine Learning System Design. *Proc. VLDB Endow.* 14, 8 (April 2021), 1338–1350. https://doi.org/10.14778/3457390.3457399

Received 2022-11-10; accepted 2023-03-31