A Compiler for Fused Relational Operations on Multisets

JAMES DONG, Stanford University, USA
FREDRIK KJOLSTAD, Stanford University, USA

We describe a comprehensive compilation approach for relational algebra, centered on an abstract loop-based intermediate representation (IR) that can express fused implementations of relational algebra operators on both sets and multisets. The loops are abstracted away from physical data structures thus making it easier to optimize. We show how to lower relational algebra query plan trees to the IR, including the complex operators that are used in production database systems, such as outer joins, non-equi joins, and differences. We then show how to compile this IR to efficient C++ code that co-iterates over the physical data structures present in relational algebra expressions. Finally, we show that our approach is portable across data structures. Since our approach can fuse across disparate operators, it achieves a 3.8× speedup (0.8–12.1×) compared to Hyper on selected LSQB benchmarks and worst-case optimal triangle queries. Our compiler also generates code of high quality: it has similar sequential performance to Hyper on TPC-H with a 1.0× speedup (0.4–4.3×) and is approaching their parallel performance with a 0.6× speedup (0.2–1.8×).

CCS Concepts: • **Software and its engineering → Compilers**.

Additional Key Words and Phrases: relational algebra, databases, sparse compilation, multisets, bags

ACM Reference Format:
James Dong and Fredrik Kjolstad. 2026. A Compiler for Fused Relational Operations on Multisets. *Proc. ACM Program. Lang.* 10, PLDI, Article 205 (June 2026), 25 pages. https://doi.org/10.1145/3808283

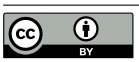
1 Introduction

Database management systems lower SQL operations to a logical query plan consisting of a tree of relational algebra operators like joins, filters, and aggregation, which is then optimized by a logical query planner, e.g., to push filtering operations before others [Chaudhuri 1998; Hellerstein et al. 2007]. A physical query planner then selects implementations for operators in the query plan from a library of implementations, such as sort-merge and hash joins, filters, and fused combinations.

By selecting from a fixed number of hand-written operator implementations, database systems leave performance on the table. Since there is a limit to the number of variants that implementers can feasibly write, we cannot support all operators on all combinations of data structures. Nor can we support many fused implementations that optimize across multiple operators. Fused operators can yield better temporal locality and, as demonstrated by the worst-case optimal join literature [Ngo et al. 2012; Veldhuizen 2013; Wang et al. 2023], can provide asymptotically better performance than unfused variants. The lack of fusion can therefore have serious performance consequences.

There are two relevant notions of *fusion*: operator fusion and loop fusion. Operator fusion combines multiple operators to produce a combined operation implementation. The main mechanism for optimization is producer-consumer loop fusion, which combines a loop that produces values with a loop that consumes those values, reducing or eliminating temporary storage and improving temporal locality. This type of fusion commonly occurs in pipeline and worst-case optimal join

Authors' Contact Information: James Dong, Stanford University, Stanford, USA, dongj@stanford.edu; Fredrik Kjolstad, Stanford University, Stanford, USA, kjolstad@cs.stanford.edu.



ACM 2475-1421/2026/6-ART205
https://doi.org/10.1145/3808283

optimization. For example, a multi-way join ($A \bowtie B \bowtie C$) eliminates the need to materialize the intermediate storage for $A \bowtie B$ and leads to asymptotic improvement for cyclic queries (Section 2).

However, the irregular nature of relational algebra makes it challenging to automatically fuse arbitrary combinations of relational operators. There are many different types of relational operators, such as selections, projections, group-bys, and joins (e.g., inner, outer, and non-equi joins). Compounding the problem, relations can be stored in a diverse set of irregular data structures, such as row and column stores, hash maps, tries, and B-trees. Fused code may therefore need to co-iterate over an arbitrary number of irregular data structures and support a large set of join types and other operations. Finally, in many practical databases relations are stored as multisets [Lamperti et al. 2001], which have more complex semantics than sets and thus more complicated code generation.

We describe a compiler approach to generating fused implementations of any relational algebra operations. In addition to generating fused code for pipelines and inner joins, which have been explored in prior work [Aberger et al. 2017; Kemper and Neumann 2011; Kovach et al. 2023; Shaikhha et al. 2022; Wang et al. 2023], our approach supports fusion of outer, left, and right joins, non-equi joins, differences, intersections, and unions, with set and multiset semantics. Table 1 compares our approach with the closest prior work. The approach is centered on a new intermediate representation called ALIR that can represent the fused execution of any set of relational operators. We describe how to automatically lower these operators to fused ALIR. The IR represents relational operations as loops that iterate over set/multiset expressions of the values in relational attributes (columns). By abstracting the loop domains as set/multiset expressions over attributes, we can specify the physical data structures in a separate data structure specification language. This data structure language generalizes the prior work on tensor format representations [Chou et al. 2018] to relational data structures as varied as row stores, column stores, tries, B-trees, and hash maps. Finally, building on prior work on sparse tensor algebra compilation [Kjolstad et al. 2017; Kovach et al. 2023; Root et al. 2024], we show how to compile the ALIR IR consisting of both abstract loops and data structure specifications into efficient fused low-level C++ code. Our contributions are:

(1) An intermediate representation (ALIR) that can represent fused relational algebra expressions as nested loops, including non-equi joins, outer joins, differences, and multisets.
(2) A lowering approach from relational algebra to ALIR.
(3) A data structure representation abstraction that can represent multiset relation storage.
(4) An iteration machine optimization machinery for generating efficient low-level co-iteration code for loops over multiset operators.

To demonstrate our compilation approach, we built a prototype compiler from relational algebra physical query plans to C++ code. We show that our approach can generate efficient code without sacrificing generality or data structure portability: compared to Hyper the performance of our generated code on TPC-H is on average 1.0× (0.4–4.3×) in sequential execution, and 0.61× (0.2–1.8×) in parallel execution. However, the benefit of our system is its ability to generate fused code. On four LSQB graph query benchmarks, which provide opportunities for fusion across different types of operators, we achieve an average speedup of 3.8× in parallel (0.8–12.1×) over Hyper.

2 Background and the Requirements of Modern Database Systems

To motivate our approach, we compare it to existing approaches to fusion of the natural relational algebra. We then analyze what is required to generalize them to extended relational algebra on multisets that is used in modern production database systems. First, we begin by building up the scope of fusion made possible by these approaches, for each describing the key contributions that make fusion possible. Next, we outline the extensions to relational algebra that we address. Finally, we survey the challenges of these extensions and generalizing to multisets.

Table 1. Comparison of different approaches to worst-case optimal join code generation.

	Fusion							
	Inner joins	Multi-attr. iteration	Non-equi joins	Outer joins	Differences	Intersections and Unions	Multiset Semantics	Data Structure Portability
EmptyHeaded [Aberger et al. 2017]	✓	✗	✗	✗	✗	✗	✓	Limited
Indexed Streams [Kovach et al. 2023]	✓	✗	✗	✗	✗	✓	✗	✓
Free Join [Wang et al. 2023]	✓	✓	✗	✗	✗	✗	✓	✗
SDQL [Shaikhha et al. 2022]	✓	✓	✗	✗	✗	✗	✓	✗
Our approach	✓	✓	✓	✓	✓	✓	✓	✓

First, we give some background on relational algebra and multisets. A *relation* is a set of tuples of some fixed length, where each element of a tuple is called an attribute. Operations on relations include set operations ($\cup, \cap, \times$, etc.), *projection* π, which restricts each tuple to a subset of attributes, and *selection* σ, which filters tuples according to a predicate. A particularly important set of operators are *joins*. For example, an inner join $\bowtie$ is defined as $R \bowtie_\theta S \coloneqq \sigma_\theta(R \times S)$. In an equi-join, θ is attribute equality. When no predicate is specified, $\bowtie$ indicates the natural join, which is equi-join on attributes with the same name. Other types of joins are discussed in Section 2.2.

Multisets allow duplicate tuples by extending binary set membership to a *multiplicity* in $\mathbb{N}$, which we notate $\#M(x)$ (multiplicity of x in M). We define $A \cup B = C$ with multiplicity $\#C(x) = \max(\#A(x), \#B(x))$, and similarly $\#(A \cap B)(x) = \min(\#A(x), \#B(x))$. We then define difference as $\#(A - B)(x) = \max(\#A(x) - \#B(x), 0)$ and disjoint union or sum as $\#(A + B)(x) = \#A(x) + \#B(x)$. We define the Cartesian product of multiset relations as $\#(A \times B)(\vec{a}, \vec{b}) = \#A(\vec{a}) \cdot \#B(\vec{b})$.

2.1 Fusion in Systems and Compilers for Relational and Sparse Tensor Algebra

Systems that are capable of fusing relational algebra include the HyPer compiler [Kemper and Neumann 2011], worst-case-optimal join (WCOJ) [Ngo et al. 2012; Veldhuizen 2013] systems such as EmptyHeaded [Aberger et al. 2017] and Free Join [Wang et al. 2023], SDQL [Shaikhha et al. 2022], and the Indexed Streams compiler [Kovach et al. 2023] that builds on ideas from sparse tensor algebra compilation [Kjolstad et al. 2017]. Table 1 compares our approach to these systems.

The HyPer compiler [Kemper and Neumann 2011] can fuse a linear pipeline of hash joins, filters, and projections. These pipelines are delineated by pipeline breaker operations that require an entire relation to be materialized. For example, a hash join requires the lookup side to be hashed, forming a pipeline breaker; the iterator side continues the pipeline. Pipeline fusion is thus insufficient to fuse cyclic queries such as $A(a, b) \bowtie B(b, c) \bowtie C(c, a)$ [Ngo et al. 2014], which cannot be linearized.

Worst-case optimal join systems [Aberger et al. 2017; Wang et al. 2023] can fuse an arbitrary combination of inner joins, achieving better asymptotic complexity than binary join systems on cyclic queries. These systems expose fusion opportunities by decomposing joins by the attributes joined. For example, the triangle join $A(a, b) \bowtie B(b, c) \bowtie C(c, a)$ has three joined attributes. For each attribute, a fused loop is generated that iterates over the intersection of the two joined relations. The loop structure forms a hierarchy where the values of an attribute to be considered are those which match all values of preceding attributes, requiring relations to be stored in a trie-like structure.

Indexed streams [Kovach et al. 2023] generalize fusion to include union operations in addition to intersections. Indexed streams are built using combinators that combine two streams to create a new fused stream along with a code generation algorithm that generates imperative code. In addition to fusing more operations, indexed streams also supports multiple data structures through a data structure specification language, and different relations can be stored in different data structures.

2.2 Challenges with Extended Relational Algebra

In this section we describe key extensions to the relational algebra that our compiler supports, as well as the challenges imposed by these extensions. The natural (standard) relational algebra [Codd

1970] consists of set operators (union, intersection, difference, and Cartesian product) along with the relational operators projection π, rename ρ, selection σ, aggregation G, and join $\bowtie$ [Garcia-Molina et al. 2008]. By extending to multisets, the semantics of these operators are changed, and one must also support the disjoint union +. Our approach supports multisets as well as the following extensions that are required by SQL in most commercial databases [Garcia-Molina et al. 2008]. The extended projection operator, which also generalizes renaming, allows arbitrary computation on attributes, e.g. $\pi_{x+y}(R)$, which returns a single-attribute relation. The grouping or aggregation operator, ${}_{y,z}\mathrm{G}_{\Sigma(x)}(R)$ gathers the tuples for each unique value of the grouping attributes (y, z), and performs an aggregate operator (here summation over x). Finally, the outer join operators $\rtimes\ltimes$ (full outer join), $\rtimes$ (left join), and $\ltimes$ (right join) extend the standard *inner* join operator $\bowtie$ by augmenting the result of an inner join with NULL values (abbreviated as $\varnothing$) so that all tuples in the left (for left joins), right, or both (full outer joins) are represented (Figure 1).

These extended operators pose challenges for existing approaches to fusing relational algebra. Outer joins introduce several problems to worst-case optimal joins: First, full outer joins require iterating over the union of joined attributes. This is not supported by WCOJ systems, as they only iterate over intersections. Second, outer joins necessitate handling NULL values. While indexed streams would support fusing the outer join $A \rtimes\ltimes_{x=y} B$ when A and B only have a single attribute each, it cannot represent the case where A or B has any additional attributes, which would require producing NULL values. Finally, existing WCOJ systems do not support fusing non-equi joins, also due to their loop structure.

A

x	y
1	1
2	2

B

y	z
2	3
3	4

Inner $A \bowtie B$

x	y	z
2	2	3

Full $A \rtimes\ltimes B$

x	y	z
1	1	$\varnothing$
2	2	3
$\varnothing$	3	4

Left $A \rtimes B$

x	y	z
1	1	$\varnothing$
2	2	3

Right $A \ltimes B$

x	y	z
2	2	3
$\varnothing$	3	4

Fig. 1. Binary equi-joins semantics.

Furthermore, NULL values introduce another complication to approaches like Indexed Streams [Kovach et al. 2023] that build on sparse tensor algebra compilation techniques [Kjolstad et al. 2017]. That line of work sidesteps explicitly handling fill values, since they can be eliminated by algebraic simplification. Extended relational algebra necessitates this handling, since NULL values are part of the output tuple. More importantly, the implication of this combined with fusion means that this NULL handling must be part of loop domains. Consider the expression $(A \rtimes\ltimes B) \cup C$. In order to fuse the union operator, the structure of $A \rtimes\ltimes B$ must be the same as that of C. So a relational compiler must handle NULLs within loop bodies, and they must also be part of the iteration structure itself.

Multiset handling also complicates fusion. First, relational algebra over multisets introduces a new operator, disjoint union +, and adds the additional semantics of multiplicity to all relational operators. For example, if A and B each have one attribute x, the expressions $A \cap B$ and $A \bowtie B$ are identical when using set semantics. This is not the case with multiset semantics: $\#(A \cap B)(x) = \min(\#A(x), \#B(x))$, while $\#(A \bowtie B)(x) = \#A(x) \cdot \#B(x)$. Second, fusing multiset operators introduces new challenges. Consider a fused loop over the expression $(A - B) \cap C$, where A, B, C each have one attribute.

```
i_A, i_B, i_C ← 0, 0, 0
while i_A < |A| ∧ i_B < |B| ∧ i_C < |C| do
  m ← min(A[i_A], B[i_B], C[i_C])
  if A[i_A] = B[i_B] = C[i_C] = m then
    omit m
  else if A[i_A] = C[i_C] = m then
    insert m into output
  if A[i_A] = m, i_A ← i + 1
  advance i_B, i_C similarly …
iterate over A ∩ C …
```

Fig. 2. TACO code for $(A - B) \cap C$

A code generation approach like TACO [Kjolstad et al. 2017] with the general expression extension of Henry et al. [2021] produces the code in Figure 2. The code in the figure has the following behavior (visualized in Figure 20) for the input $A = \{\!\{1, 1\}\!\}, B = C = \{\!\{1\}\!\}$, where double braces indicate a multiset. In the first iteration, all three relations contain the current value 1, so the value is omitted. However, in the next iteration, all three iterators are advanced. Since only A contains another 1, the value is not produced, and thus no values are produced by this loop. However, the value of $(A - B) \cap C$ is $\{\!\{1\}\!\}$. From this, we see that multiset iteration is not a straightforward extension of existing techniques for set iteration, as they fail to correctly account for duplicates.

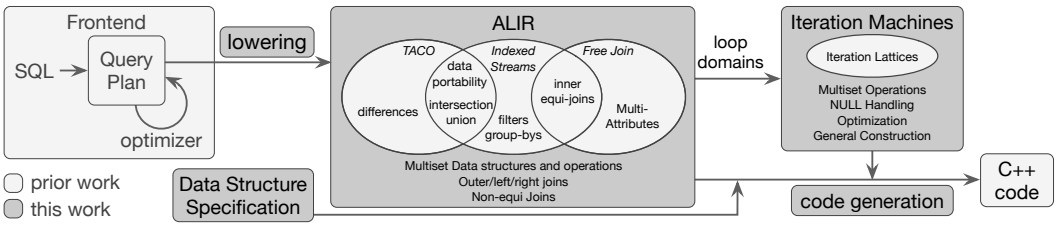


Fig. 3. Overview of our compilation approach. The frontend is outside the scope of this work.

3 Overview

We present a compiler approach that lowers relational algebra in the form of a physical query plan along with a data specification for each relation to C++ code. Figure 3 gives an overview of the approach and how it relates to prior work, which is shown in light gray. On the left, an SQL query is lowered to relation algebra in the form of a logical query plan. This plan is then optimized by a query optimizer and turned into a physical query plan where the different parts of the graph are pinned to specific implementations. These initial steps have been explored in prior work [Chaudhuri 1998; Tu and Ré 2015] and are outside the scope of our paper.

The compiler approach that we describe in this paper takes as input the pre-optimized query plan produced by a query optimizer along with a data specification that describes the storage and indexing structures for each relation. The data specifications could either be determined by users or by an automatic index selection system.

The first step lowers the physical query plan to an IR where abstract loops iterate over set expressions of the relational attributes (columns). This IR, called the Abstract Loop IR (ALIR), unifies the capabilities of the TACO IR [Kjolstad et al. 2019], the indexed stream IR [Kovach et al. 2023], and Freejoin [Wang et al. 2023], while also generalizing to support multiset semantics, all join types, and non-equi joins whose predicates go beyond simple equalities. The final code generation step lowers the ALIR to C++ code that computes the query by co-iterating over the storage of the relations. The code generation process uses an IR we call iteration machines that generalize the iteration lattices from prior work [Kjolstad et al. 2017] to support multisets and the iteration over and production of null values. We achieved this generality by reformulating lattices to state machines, which also let us define an optimal minimization algorithm.

4 Relation Storage Data Model

How relations are stored is an important and complex part of efficiently executing relational queries [Abadi et al. 2008; Selinger et al. 1979]. The simplest relational storage is a list of tuples in memory. Performant databases, however, use several other advanced data structures or auxiliary indexes to improve performance. In order to effectively fuse arbitrarily complex operations, we define a data model that lets us break down accesses into per-attribute (per-column) granularity. Each attribute of a relation thus has its own data storage, and implementation details of how data is accessed is abstracted by a general interface.

These attribute data structures are linked together in an abstract structure known as a *coordinate tree* [Kjolstad et al. 2017] shown in Figure 4. We extend coordinate trees to relations with arbitrarily-typed attributes, and to multisets by adding a new level for duplicates. We extend the iterator model using the skip (`skipto`) operation [Kovach et al. 2023] for worst-case optimal joins, and the `reset` operation for Cartesian products. We also note that this model is similar to trie data structures used in WCOJ systems [Veldhuizen 2013].

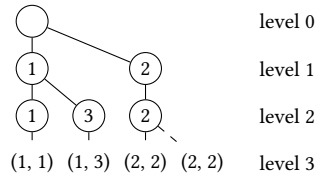


Fig. 4. A coordinate tree. Nodes represent multiset elements and paths represent tuples.

Our approach represents a relation R with attributes $a_1, \ldots, a_r$ as a consisting of one level for each attribute, in some predefined order. In a coordinate tree, attributes are stored hierarchically, where level 0 represents all of R, and the children at level i represent the distinct values of attribute a_i of elements that match all previous attributes, with the last level holding duplicates.

The storage description for a relation consists of layer storage for each layer in the coordinate tree. Layer storage describes how the data in each level can be accessed. For example, a layer may be stored as a sorted list of values along with a reference to the storage for the next level, a hash table from attribute value to row indices, or a dense set over all possible attribute values. This is done by a small set of primitive routines, defined for each layer storage implementation.

These routines are grouped into two *capabilities*: lookup and iteration, and which capabilities a layer has influences how iteration is performed for a given set of layers. For each set of layers, the layers with iterator capability are iterated simultaneously, and the remaining layers must have lookup capability. The choice of which layers use which capability depends both on the desired performance characteristics and the structure of the computation, as not every choice of layer capabilities can be used to generate correct code (see Appendix D in the supplemental material).

Lookup. Layers with the lookup capability have the interface of a membership query. The following routines are required for a layer of type T to have the lookup capability:

- **`init`**: $() \longrightarrow ()$ initializes a layer.
- **`present`**: $T \longrightarrow \text{Bool}$ returns whether or not a given value is present in a layer.
- **`value`**: $() \longrightarrow V$, where V is the type of the related values, retrieves the related values and is only required for the last layer if there are related values.

Iteration. Layers with the iteration capability (iterators) have the interface of an iterator [Kovach et al. 2023] over the set of layer's values, which is required to be sorted if there are any other iterator layers participating in co-iteration, all with the same sort order. Iteration capability requires the following routines (Figure 5 gives an example implementation for tries):

- **`init`**: $() \longrightarrow ()$ initializes the iterator.
- **`curval`**: $() \longrightarrow T$ returns the value of the current element.
- **`advance`**: $() \longrightarrow ()$ advances the iterator to the next element.
- **`present`**: $T \longrightarrow \text{Bool}$, defined as $\texttt{present}(v) \coloneqq \textbf{let}\ c \leftarrow \texttt{curval}()\ \textbf{in}\ c == v$, has the same function as for lookup capability, but it is sufficient to check the current element.
- **`valid`**: $() \longrightarrow \text{Bool}$ returns whether the iterator is not yet exhausted.
- **`skipto`**: $T \longrightarrow ()$ advances the iterator to the first element greater than or equal to the given value, which for many data types is more efficient than repeatedly calling `advance`.
- **`reset`**: $() \longrightarrow ()$ resets to the beginning (optional); only required for Cartesian products.
- **`value`**: $() \longrightarrow V$ retrieves the related values if they exist.

This framework for describing relation storage suffices to represent many common storage types, such as row and column storage, tries, hash and B(+)-tree indices, as well as dense and sparse matrices. Figure 6 visualizes these data structures for the abstract coordinate tree in Figure 4.

Related Values and Primary Keys. In many relations, a small number of attributes form the relation's *primary key*, which are typically the attributes that query implementations need to iterate over. The other attributes, which we call *related values*, are not typically iterated over but instead treated as additional data to be looked up [Garcia-Molina et al. 2008]. The values in the primary key are unique, so related values form a chain under the primary key attributes. Therefore, it is not necessary to iterate over these attributes. Instead, they are attached to the storage for primary keys, and iteration state is associated only with the primary key storage.

```
class TrieStorage(IteratorStorage):
  def __init__(self, node_name, varname, emitter, has_children, parents, pfx):
    super().__init__(node_name, emitter, '')
    # init fields varname, indname, has_children, parents, pfx, ty...
    self.i = pfx + varname + '_i' # .val, .end, .idx...
    self.lower = '0'; self.upper = f'{varname}.size()'
    if len(parents) > 0: self.lower, self.upper = self.parents[-1].bounds()
  def bounds(self): return f'{self.idx}[0]', f'{self.idx}[1]' # trie specific
  def init(self):
    self.emit(f'{self.ty} *{self.i} = &{self.varname}[0] + {self.lower};')
    self.emit(f'{self.ty} *{self.end} = &{self.varname}[0] + {self.upper};')
    if self.has_children:
      self.emit(f'size_t *{self.idx} = &{self.indname}[0] + {self.lower};')
  def advance(self):
    self.emit(f'{{ ++{self.i};')
    if self.has_children: self.emit(f' ++{self.idx};')
    self.emit(f' }}')
  def valid(self): return f'{self.i} < {self.end}'
  def curval(self): return f'*{self.i}'
  def skipto(self, expr):
    self.emit(f'{self.i} = gallop_lower({self.i}, {self.end}, {expr});')
    if self.has_children:
      self.emit(f'{self.idx} = &{self.indname}[{self.i} - {self.varname}];')
```

Fig. 5. Example implementation of trie storage layer.

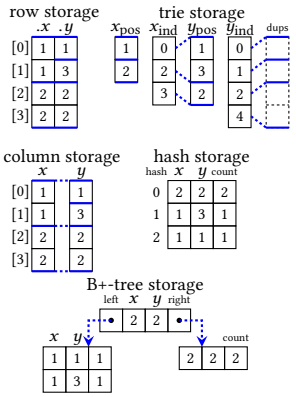


Fig. 6. Concrete data structure layouts for the abstract coordinate tree in Figure 4.

5 Abstract Loop IR

While the relational algebra is capable of expressing many types of computation across several domains, it does not let you express fused code. Operators are inherently separate constructs and naive execution thus results in many temporary relations. To address this issue, prior work on worst-case optimal joins has largely adopted a nested loop-based representation [Ngo et al. 2014; Veldhuizen 2013; Wang et al. 2023] that supports multi-way inner equi-joins.

Like prior work, our Abstract Loop IR (ALIR) representation is based on loops and focused on expressing fusion. Our loops, however, are more general than the loops in prior work and supports all equi and non-equi joins (not just inner equi-joins), multiset operations, union, intersections, and differences. Moreover, our IR is abstracted from physical data and our code generator generates specialized fused code that co-iterates over different combinations of physical data structures.

5.1 Basic structure

The syntax of ALIR (Figure 7) consists of nested loops whose iteration domain is a set or multiset expression over one or more levels from relation coordinate trees. ALIR has four types of statements:

Emit statements $(a, b, \dots)$ produce a tuple that is added to the result relation. They consist of a parenthesized list, where each element corresponds to an attribute in the result and is computed as a scalar expression of values from input relations and loop variables.

Aggregation/assignments such as $T[x].z \mathrel{+}= y$, consist of a relation, optional scalar expressions for indices, updated attribute, and scalar expression. The statement updates a value in a temporary relation, with the indices specifying the tuple to be modified.

Let statements such as **let** $T \coloneqq S_1$ **in** S_2 consist of a temporary relation name and two statements. S_1 computes a temporary relation T that is provided as an input relation to S_2.

Loops have a loop variable, a domain specified as a multiset expression, and a body. Loops iterate through the elements in the loop domain and execute the body for each element. Nested loop domains are implicitly correlated by coordinate tree traversal (see Section 5.2).

For example, the following inner join produces a tuple for every pair of tuples in A and in B that match on shared attributes: $R = A \bowtie B$ over relations $R(x, y, z)$, $A(x, y)$, and $B(y, z)$. Figure 8 shows the join represented in ALIR. However, this is not the only way to encode this computation in ALIR. For example, the loops can be reordered arbitrarily without changing the semantics of the program, though such orderings must be consistent with the structure of the coordinate trees.

$$
\begin{aligned}
\langle stmt \rangle ::= \ & \textbf{for } \alpha \in \langle mexpr \rangle \ \langle stmt \rangle \ldots \\
& \mid \ (\langle expr \rangle, \ldots) \\
& \mid \ R\,[\,\langle expr \rangle, \ldots].\alpha \circ\!= \langle expr \rangle \\
& \mid \ \textbf{let } R(\alpha, \ldots) \coloneqq \langle stmt \rangle \textbf{ in } \langle stmt \rangle \ldots \\
\\
\langle expr \rangle ::= \ & \alpha \mid (\text{constant}) \mid \langle expr \rangle \circ \langle expr \rangle \\
& \mid \ R.\alpha \quad \mid \ \ldots
\end{aligned}
$$

$$
\begin{aligned}
\langle mexpr \rangle ::= \ & R.\alpha \quad \mid \quad R[\ \alpha = \langle expr \rangle\ ] \\
& \mid \ \{\,\alpha \mid \langle expr \rangle\,\} \quad \mid \quad \{\,\langle expr \rangle\,\} \\
& \mid \ \langle mexpr \rangle \ \langle mop \rangle \ \langle mexpr \rangle \\
& \mid \ \overline{\langle mexpr \rangle} \\
& \mid \ \langle mexpr \rangle \ (\langle mexpr \rangle) \\
& \mid \ \langle mexpr \rangle \cup \varnothing \\
\\
\langle mop \rangle ::= \ & \cup \mid \cap \mid - \mid + \mid \times
\end{aligned}
$$

Fig. 7. Grammar for ALIR. R represents a relation, and α is an attribute.

```
for x ∈ A.x
  for y ∈ A.y ∩ B.y
    for z ∈ B.z
      for dups ∈ A × B
        (x, y, z).
```

Fig. 8. Inner join in ALIR.

5.2 Nested loops and coordinate tree traversal

In a loop nest that iterates over multiple relational attributes, the domains of inner loops implicitly depend on the loop variables of outer loops. For example, the loop for y in Figure 8 only iterates over the values of $A.y$ within the subtree corresponding to the current value of x. We use the term *subtree* to denote the restriction of a relation by the values of outer loops. Since the current subtree is determined syntactically, we leave this traversal dependency implicit in the IR.

In our IR, we allow intermediate transformations to have arbitrary attribute orderings; a permuted ordering simply traverses a transformed coordinate tree based on the new loop ordering. While the IR allows this flexibility while maintaining semantics, we place restrictions on the final IR used for code generation based on how relations are laid out in physical storage. As some storage types restrict the order attributes can be accessed, the coordinate tree abstraction has the same restriction. We require that the final IR used for code generation must match the coordinate trees. An optimizer should therefore be aware of the specified data layout and either ensure the loops are generated accordingly, or it should generate temporary storage to transform the data layout.

As discussed in Section 4, it is not always desirable to model all attributes of a relation in the coordinate tree. Besides primary keys, multi-attribute iteration is a performance optimization [Wang et al. 2023]. In ALIR, such trailing attributes can be accessed using $R.\alpha$ within the innermost loop. As a result of truncated iteration, lookups are not guaranteed to be over iterated attributes. For example, a loop may iterate over $A.x$ but look up $B.y$ using $A.y$ as follows: $B[y = A.y]$.

5.3 Loop domains

The for loops in ALIR iterate over loop domains consisting of set or multiset expressions. ALIR supports the following multiset operators, with standard semantics [Syropoulos 2001]: $\cup$ (union), $\cap$ (intersection), $-$ (difference), $+$ (concatenation), and set complement. ALIR also supports Cartesian products ($\times$), which are only used in the innermost dups loop (see Section 6). When extended to other loops, it becomes the natural join $\bowtie$ and operates on unique values (e.g., $\{\!\{1, 1, 2\}\!\} \times \{\!\{1, 2, 2\}\!\} = \{\!\{1, 1, 2, 2\}\!\}$). We define $\cup$ as having *maximum* multiplicity, not the sum, which we denote using $+$.

In addition to input relations, ALIR also supports predicate relations and singletons. Predicate relations only support lookup, not iteration, and are notated using set-builder notation. Singletons (notated $\{v\}$) hold a single value v and, unlike predicate expressions, support iteration.

In ALIR, there is special notation for certain operations that are necessary for full handling of relational algebra with multisets. First, the notation $e_1(e_2)$ notates the relations in e_2 as index-only relations. Such relations are used only for traversing the coordinate tree and do not contribute to the output tuples. The operator is necessary because ALIR loops express both the computation domain and iteration over coordinate trees. For example, the expression $A.x(B.x)$ is used when a loop should iterate over the values of $A.x$, but we also need to traverse $B.x$ in that coordinate tree in order to introduce a dependency for inner loops. Index-only relations do not require the layer to have lookup capability: it is also possible to perform co-iteration as usual, but cases where the current value is only present in the index-only relation are discarded.

ALIR's NULL-padding operator ($\cup \varnothing$) produces a single NULL value when a relation is empty. This is a property that spans all loops involving R; such loops are either NULL-padded (NP loops) or not (join condition). Exactly one of the following cases must hold: either no NULLs are produced and all loops involving R correspond to a tuple in R, or all NP loops produce NULL and the join condition produces a sub-tuple that does not belong to any tuple in R.

Consider the ALIR loop example to the right, where the inner loop is NP, and $A.x \cup B.x$ is the join condition. The first case corresponds to the output tuples $(x, y) \in A$, and the second case to the tuples (x, NULL) for $x \in B.x - A.x$.

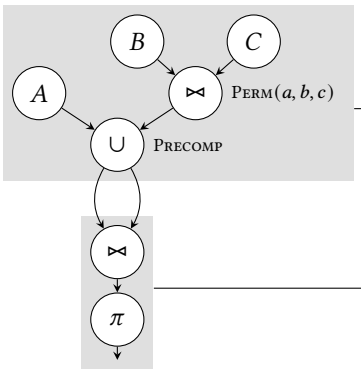
```
for x ∈ A.x ∪ B.x
  for y ∈ A.y ∪ ∅
    (x, y)
```

Code generation (Section 8) requires all NP loops to occur after the join condition. For example, the expression $A.x \cup B.x$ (where both A and B can then be NULL-padded) has three such cases: when $x \in A.x \cap B.x$, when $x \in A.x - B.x$, and when $x \in B.x - A.x$. An inner loop over $A.y \cup \varnothing$ is thus able to determine whether it is empty based on which branch is being generated.

The semantics of the NULL-padding operator does not require this restriction, though, and a loop nest without this structure can be transformed into two separate loops with the same semantics, by breaking it down into the two cases. First, the transformed code generates all non-NULL tuples, by appending $\cap R$ to the join condition and removing the $\cup \varnothing$ operator. Next, it generates the NULL tuples by subtracting R from the join condition. Since the identity $X \cap R + X - R = X$ holds for multisets X, R, the behavior of the transformed loops is equivalent to the original.

6 Lowering from Relational Algebra

The lowering procedure from relational algebra to ALIR lowers a *physical query plan* with annotations that influence the lowering process. This query plan is a graph of standard relational operators, where each operator may be annotated with any combination of Perm(π, ψ), Precomp, as in Figure 9, and loop tags (Appendix E in the supplemental material). Perm determines loop order and Precomp is a pipeline breaker that splits the query plan into sections that are lowered separately and joined together. The lowering procedure then applies rules for each type of operator, proceeding from inputs to outputs. These rules are given in Appendix A in the supplemental material. The Perm and Precomp rules introduce nondeterminism into the lowering process, as they can be applied to any expression, making rule selection nondeterministic. These rules must be specified in the query plan and are discussed in detail below. Some rules, such as Op and Concat, overlap; the most specific rule (here Concat) is used by default unless a similar annotation is given.




```
let T ≔
  for x ∈ A.x ∪ B.x
    for y ∈ A.y ∪ (B.y ∩ C.y)
      for z ∈ A.z ∪ C.z
        for dups ∈ A ∪ (B × C)
          (x, y, z)
in
  for x ∈ T.x ∪ T′.z
    for y ∈ T.y
      for z ∈ T′.y
        for dups ∈ T × T′
          (x, y, z)
```

Fig. 9. Example physical query plan with Perm and Precomp annotations for $\pi_{T.x,T.y,T'.y}(\rho_T(A \cup B \bowtie_y C) \bowtie_{T.x=T'.z} \rho_{T'}(A \cup B \bowtie_y C))$ (ρ renaming the temporary relation). T' denotes another cursor for the same T.

```
for x ∈ A.x(C.x)
  for y ∈ (A.y ∩ B.y)(C.y)
    for z ∈ B.z(C.z)
      for dups ∈ (A × B) − C
        (x, y, z).
```

Fig. 10. ALIR for $(A \bowtie B) - C$.

```
let T(Σx, y) ≔
  for x ∈ A.x
    for y ∈ A.y
      T[y].Σx += A.x
in
  for Σx ∈ T.Σx
    (Σx²).
```

Fig. 11. $\pi_{\Sigma x^2}({}_yG_{\Sigma(x)}A)$.

Duplicate handling. When evaluating relational algebra expressions with multiset semantics, we must take care to properly handle duplicate entries. For example, while a multiset intersection matches on pairs of duplicates and produces the same number as in the smallest operand (e.g., $A(x) \cap B(x) = \{\!\{1,1\}\!\} \cap \{\!\{1,1\}\!\} = \{\!\{1,1\}\!\}$), a multiset inner join produces the Cartesian combination of matches (e.g., $A(x) \bowtie B(x) = \{\!\{1,1\}\!\} \bowtie \{\!\{1,1\}\!\} = \{\!\{1,1,1,1\}\!\}$). The reason is that inner joins are defined in terms of a filtered Cartesian product: $A(x) \bowtie_{A.x=B.x} B(x) \coloneqq \sigma_{A.x=B.x}(A(x) \times B(x))$. Our approach to producing Cartesian combinations of duplicates for join operators is to place a duplicates loop as the last loop level. The duplicates loop iterates over the Cartesian combination of matching duplicate entries from the two sides of join operators, thus generating the correct number of duplicates. Thus, the difference between a relational multiset intersection operator and a multiset inner join is clearly reflected in the ALIR loops. For example, the expression $A(x,y) \bowtie B(y,z) - C(x,y,z)$ requires a duplicates loop to be computed without temporary storage, as shown in Figure 10. Note that C is an index-only relation in every attribute except the duplicates loop.

6.1 Non-join operations

Projection is represented by replacing the tuple output statements. For example, the ALIR for $\pi_{(x,z+1)}(A)$, where A has attributes (x,y,z), replaces the tuple (x,y,z) with the tuple in the projection, $(x, z+1)$. The replacement rule is $\dfrac{\Gamma \vdash e \rightsquigarrow \textbf{for} \ldots (\vec{z}) \qquad \Gamma \vdash e : (\vec{x})}{\Gamma \vdash \pi_{\vec{y}}(e) \rightsquigarrow \textbf{for} \ldots (\vec{y}[\vec{z}/\vec{x}])}$[Proj], where $\Gamma \vdash e \rightsquigarrow P$ states that given schema Γ, the expression e is lowered to program P. The rule is read as follows: To lower the expression $\pi_{\vec{y}}(e)$, first lower e. Then lower $\pi_{\vec{y}}$ by replacing the actual value z for each attribute x specified in the projection expression y as the output tuple and reusing the loop structure of e.

Selection operations are represented by an intersection with a relation that encodes the predicate as a set builder. For example, the ALIR loop for the attribute y in $\sigma_{y \text{ is even}}(B)$ has the domain $y \in B.y \cap \{y \mid y \text{ is even}\}$. To lower a selection σ_ϕ, an intersection with $\{x_k \mid \phi\}$ is inserted at a loop where all variables are defined.

To represent aggregation in ALIR, the tuple output is replaced with a compound assignment. For example, Figure 11 shows the ALIR code for the relational expression $\pi_{(\Sigma x^2)}({}_y\mathrm{G}_{\Sigma(x)}(A))$, which first aggregates the sum of x grouping by the values of y and then squares the sum, discarding y. This first performs the aggregation into a temporary relation T, then performs the projection to produce the final output. In general, aggregation is always performed into a temporary or the result, as intermediate values must be stored somewhere. Thus, for a fully-fused expression, aggregation must occur at the outermost level. As such, the Agg rule is a pipeline breaker, similar to Precomp.

6.2 Inner joins

We represent inner equi-joins using a loop structure that is the same as prior work on worst-case optimal joins [Kovach et al. 2023; Ngo et al. 2012; Veldhuizen 2013]. First, all attributes in the joined relations are nested into loops, and then loops for joined attributes are fused using the intersection operator. For example, the triangle join $A(x,y) \bowtie B(y,z) \bowtie C(x,z)$ is shown in Figure 12. The resulting loops each iterate over the intersection of the corresponding attributes in the two relations for each joined attribute (x,y,z).

```
for x ∈ A.x ∩ C.x
  for y ∈ A.y ∩ B.y
    for z ∈ B.z ∩ C.z
      (x, y, z).
```

Fig. 12. ALIR for triangle join.

6.3 Outer joins

It is essential to handle the full set of outer joins (full outer join, left join, right join, and fused combinations thereof) to support the complete extended relational algebra supported by most commercial relational database management systems and a key generalization of our work.

```
for y ∈ A.y ∪ B.y
  for x ∈ A.x ∪ ∅
    for z ∈ B.z ∪ ∅
      (x, y, z).
```

Fig. 13. ALIR for $A \mathbin{⟗} B$.

To represent outer joins in ALIR, we only need to consider what the requisite domain is for each nested loop. Non-join attributes are either unchanged (both sides in inner join, LHS of left join, RHS of right join, neither side in full outer join) or are possibly augmented with an additional NULL value. The join dimension (which combines both joined attributes) simply follows the semantics of the corresponding join type: the intersection of the joined attributes for inner join, union for full outer join, and only the left/right relation for left/right join, though left/right joins require special handling as described below. For example, a full outer join $A \mathbin{⟗} B$ can be expressed using the ALIR in Figure 13, using the special $\cup\, \emptyset$ notation.

```
for y ∈ A.y(B.y)
  for x ∈ A.x
    for z ∈ B.z ∪ ∅
      (x, y, z).
```

Fig. 14. $A \mathbin{⟕} B$.

Left or right joins illustrate the necessity of index-only relations, relations whose values are needed but will not be iterated over. Consider the ALIR in Figure 14 for the left join $A \mathbin{⟕} B$. We would like the loop $z \in B.z \cup \emptyset$ to only iterate over the values of $B.z$ for the tuples where $B.y = A.y$. Without the index-only relation indicated in red, $B.y$ is not present in the IR, so the correct ALIR must use index-only relations to traverse the coordinate tree.

6.4 Non-equi joins

ALIR can also handle non-equi joins in full generality (joins over an arbitrary predicate, and not just equality). An inner join $A \bowtie_\theta B$ with an arbitrary predicate θ is equivalent to $\sigma_\theta(A \times B)$. Using this, an example non-equi join $A \bowtie_{A.y<B.y} B$ can be naturally expressed in ALIR, as shown in Figure 15 (left). There is one loop for each attribute, and the filter $A.y < B.y$ is inserted at the outermost valid location, which is the loop for $B.y$, resulting in the domain $B.y \cap \{y \mid y_1 < y\}$.

However, a problem arises with outer non-equi joins, which cannot be described using a simple translation. The semantics of such joins is that in addition to the tuples produced by the corresponding inner join, tuples from the left relation (for left and full outer joins) and/or right relation (for right and full outer joins) are included if they would not otherwise be present, with the missing values substituted with NULLs. The results of each type of join for the predicate $A.y < B.y$ on the data $A = \{\!\{(1,1),(3,3)\}\!\}, B = \{\!\{(1,2),(3,4)\}\!\}$ are shown below:

$A \bowtie_{A.y<B.y} B$

x	A.y	B.y	z
1	1	3	4

$A \mathbin{⟕}_{A.y<B.y} B$

x	A.y	B.y	z
1	1	3	4
3	3		

$A \mathbin{⟖}_{A.y<B.y} B$

x	A.y	B.y	z
1	1	3	4
		1	2

$A \mathbin{⟗}_{A.y<B.y} B$

x	A.y	B.y	z
1	1	3	4
3	3		
		1	2

Temporary storage is unavoidable to represent these operations. Our system decomposes them into an inner join computing into a temporary that is used to compute the full join, adding in missing values. For example, the left join $A \mathbin{⟕}_{A.y<B.y} B$ generates the ALIR shown in Figure 15 (center). This differs in duplicate handling from a projected left join, which results in the duplicate expression dups $\in T \times A \cup \emptyset$ that produces extraneous duplicates for every duplicate tuple in A.

```
for x ∈ A.x
  for y1 ∈ A.y
    for y2 ∈ B.y ∩ {y | y1 < y}
      for z ∈ B.z
        for dups ∈ A × B
          (x, y1, y2, z).
```

```
for x ∈ A.x
  for y1 ∈ A.y
    let T(y2, z) ≔
      σ_{A.y<B.y}(B)
    in
      for y2 ∈ T.y2 ∪ ∅
        for z ∈ T.z ∪ ∅
          for dups ∈ T ∪ ∅
            (x, y1, y2, z).
```

```
let T(x, y1, y2, z) ≔
  A ⋈_{A.y<B.y} B
in
  A ⟕_{A.y<B.y} B using T…
  for y2 ∈ B.y(T.y2)
    for z ∈ B.z − T.z
      for dups ∈ B
        (∅, ∅, y2, z).
```

Fig. 15. Three non-equi joins for the expression $A.y < B.y$. From left to right: inner, left, and full outer.

The same idea can be used for full outer joins, but when augmenting the result with B, the outermost loop is $A.x(T.y) \cup \emptyset$. As the presence of $\emptyset$ depends on values only produced in a later loop, this cannot be used to generate code. The lowering process fixes this by splitting the loop in two, with iteration order switched for the second. This is shown in Figure 15 (right), and is the result of eliminating the invalid padding by the transformation described in Section 5.3.

6.5 Precomputation and Permutation

Although a primary goal of our work is to express fused loops, many situations benefit from not fusing, or partially fusing, loops, such as when avoiding excess recomputation. ALIR supports such cases through precomputation of temporary relations. Precomputation computes a temporary relation that is then used in another computation, and has the effect of unfusing loops. For example, the expression $\sigma_{x^2+y^2=r^2}(A) \bowtie B$ that computes a filtered version of A joined with B could be computed by first creating a temporary relation $T(x, y)$ corresponding to $\sigma_{x^2+y^2=r^2}(A)$, and then performing an inner join between T and B. This is shown in Figure 16. This can improve performance as we can skip all work in the inner loops for values of y where no matching x exists. The storage of temporary relations must also be specified in the storage description separately from the IR.

```
let T(x, y) ≔
  for x ∈ A.x
    for y ∈ A.y ∩
        {y | x² + y² = r²}
      (x, y).
in
  for y ∈ T.y ∩ B.y
    for z ∈ B.z
      for x ∈ T.x
        (x, y, z).
```

Fig. 16. ALIR for $\sigma_{x^2+y^2=r^2}(A) \bowtie B$ with temporary T.

The Precomp rule applies to any *function* of an expression $f(e)$; therefore, an expression which is used multiple times can be precomputed once, and the result used multiple times, such as in Figure 9. Then if the inner expression is lowered to P, and $f(T)$, where T is a new temporary relation, is lowered to Q, we have that the original expression $f(e)$ can be lowered to **let** $T \coloneqq P$ **in** Q.

Similarly, loop ordering can greatly impact performance, and we use a similar transformation rule to permute loops. The Perm rule allows for permuting loops arbitrarily, with the predicates permuted separately such that variables are not used outside of where they are defined.

6.6 Set semantics

ALIR supports set semantics automatically, as sets are multisets and all multiset operations applied to sets have equivalent semantics. However, if relations are known to be sets, then certain operations, such as set complement, become possible, and multiset-specific handling can be omitted. These simplifications are performed as a post-processing step after lowering is completed.

In particular, there are two main changes for set semantics compared to multiset semantics. First, during lowering to iteration machines, the multiset-specific "not ready" behavior can be replaced by omitter behavior, which is slightly cheaper and can reduce the size of the minimal iteration machine. Also, iteration machines only involving sets allow the complement to be defined.

Second, in the ALIR itself, the duplicates loop can be simplified and often omitted. For a set relation R, if R is required in all parent loops ($D - R$ is empty for all domains D), then it can be replaced by a singleton set 1. Then by applying identities $1 \times 1 = 1 \cup 1 = 1 \cap 1 = 1$, the duplicate loop domain can be simplified, and if the resulting domain is 1, then it can be removed entirely. For queries involving concatenation or differences, as well as queries that mix set and multiset relations, the duplicates loop cannot be eliminated but can often be simplified.

7 Iteration Machines

A key feature of ALIR is that loops iterate over set/multiset expressions of relational attributes. This design abstracts loops from the physical data structures that store the relations and make it easy to express fusion, as shown by the loop over the x attributes of three relations in Figure 17. This loop could, for instance, result from a fused combination of an inner and an outer join. To efficiently

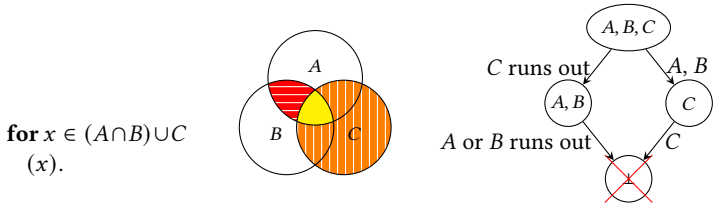


Fig. 17. Fused loop, Venn diagram, iteration lattice for $(A \cap B) \cup C$.

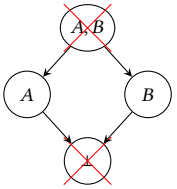


Fig. 18. $A \oplus B$

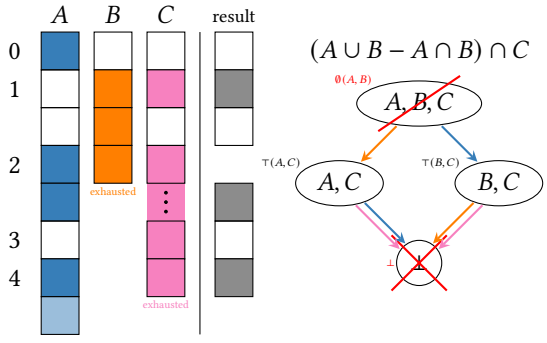
execute it, we must lower it to efficient low-level code that co-iterates over actual physical data structures, such as a column store or a trie.

A key part of this lowering algorithm, described in Section 8, is an intermediate representation called *iteration machines* that is generated from ALIR loop domains. Iteration machines are deterministic finite automata (DFAs) that represent how optimized co-iteration over sorted relational attributes transitions to progressively simpler types of iteration as relations run out of values, as well as what tuples are emitted based on what relations contain a particular value. They thus capture and generalize the intuition behind the conventional implementation of binary merge-join, where loops exit as soon as any side runs out of values. We explain the semantics and construction of iteration machines in this section and show how to use them to generate code in Section 8.

7.1 Background on Iteration Lattices

Iteration machines are inspired by the iteration lattices developed by Kjolstad et al. [2017], but are simplified and generalized to support relational algebra and multiset semantics. Unlike iteration lattices, iteration machines can have attributes over non-iterable spaces (e.g., the set of all strings), have an optimal minimization algorithm, and support the iteration domain of operators such as multiset difference and and disjoint union/concatenation. Figure 19 shows an iteration machine and its execution for a complex multiset expression that could not be represented by iteration lattices.

We first give some background on iteration lattices that also applies to iteration machines. We then give a new but equivalent presentation of these ideas as DFAs and a floor function to motivate the novel ideas in iteration machines. Figure 17 shows an iteration lattice. Starting from the top, each node represents a set of relations that must be considered. For example, A, C can be treated the same as C, so it is not included. Edges are inserted between nodes for transitions when a relation runs out of values. Henry et al. [2021] extended iteration lattices to general array programming,




```
int *A_i, *A_end, *B_i, *B_end, *C_i, *C_end;
while (A_i < A_end && A_i < B_end) { // A, B, C
  int min_i = min(*A_i, *B_i, *C_i);
  if (*A_i==min_i&&*B_i==min_i&&*C_i==min_i) {
    ++A_i; ++B_i; ++C_i;
  } else if (*A_i == min_i && *C_i==min_i) {
    emit(min_i);
    ++A_i; ++C_i; if (*B_i == min_i) ++B_i;
  } else if (*B_i == min_i && has_C) {
    emit(min_i);
    ++B_i; ++C_i; if (*A_i == min_i) ++A_i;
  } else {
    if(...)++A_i;if(...)++B_i;if(...)++C_i; } }
while (A_i < A_end) { // A, C
  int min_i = min(*A_i, *C_i);
  if (*A_i == min_i && *C_i == min_i) {
    emit(*A_i); ++A_i; ++C_i; }
  else { if(...) ++A_i; if(...) ++C_i; } }
// ... B, C analogous to A, C
```

Fig. 19. An iteration machine for a loop over $(A \cup B - A \cap B) \cap C$, where A, B, C have trie storage (Figure 5).

Fig. 20. Iteration state for an incorrect (left) and a correct (right) IM. Red bars are positioned before the current value of each iterator. For each IM, we show the initial state and the state after one iteration.

including set complements, introducing *omitter* nodes that influence case handling. Figure 18 includes an omitter node A, B that indicates that this case is not part of the iteration domain.

Within a loop, the compiler must generate different code, splitting each iteration into multiple cases depending on what relations contain the current value. This process of *case handling* involves determining what cases need to be handled, the conditions for each case, and how iteration proceeds for each case. For any possible set of input relations that contain a given value, an iteration lattice assigns a representative state (node). This is the key reason why iteration lattices are useful for code generation: iteration lattices precisely capture the behavior of an expression by splitting the space of combinations of input relations (Venn diagram) into disjoint spaces each represented by a state. Then if one looks at the node corresponding to the inputs that have not dropped out, only cases corresponding to reachable nodes need to be considered.

For example, in the example of $(A \cap B) \cup C$, the set of inputs $\{A, C\}$ can be handled exactly the same as the set $\{C\}$, as once B runs out of values, it does not matter whether a value is in A or not. Therefore, we assign both of these sets the representative of $\{C\}$. And by looking at the iteration lattice, it is clear that this case only needs to be checked in states (A, B, C) and (C).

For an iteration lattice and set of inputs, the representative is given by the *floor function*, defined precisely in Appendix C. The transition function δ for an iteration lattice is uniquely determined from the set of states using the floor function.

To give intuition for what this function represents, we look at the Venn diagram in Figure 17. We color in this Venn diagram according to the nodes of Figure 17 for $A \cap B \cup C$, bottom up. We first color C orange, then $A \cap B$ (representing (A, B)) red, and finally $A \cap B \cap C$ yellow. These colors represent which node is a representative at any given point in the Venn diagram: even though $A \cap C$ is not present in the IM, we see that it is covered by its representative C.

Each node in an iteration lattice may be labeled with one of two possible behaviors: producer (input that produces a value) or omitter (input that does not not produce a value). Iteration machines introduce a third behavior, $\emptyset$ (not ready to produce output, but values may remain), discussed in the following section and used in construction. Let $\mathcal{I}(n)$ represent the label of the node n, where $\top$ represents producer, $\bot$ omitter, and $\emptyset$ not ready. Then we can define the interpretation of M at value p as $[\![M]\!](p) = \mathcal{I}(\text{floor}(p))$.

7.2 Iteration control

Applying this procedure directly to multisets works well for multiset union and intersection (see Section 2 for definitions), but a problem arises when we try to handle difference. To illustrate this problem, we look at the expression $(A - B) \cap C$ on the inputs $A = \{\!\{1, 1\}\!\}, B = \{\!\{1\}\!\}, C = \{\!\{1\}\!\}$. The correct result should be $(A - B) \cap C = \{\!\{1\}\!\}$. A minimal iteration lattice for this expression has nodes $\{(A, B, C), (A, C), ()\}$ and interpretation $\mathcal{I}(A, B, C) = \bot, \mathcal{I}(A, C) = \top, \mathcal{I}() = \bot$.

But applying this iteration machine produces the incorrect behavior shown in Figure 20 (left). First, since 1 appears in all three inputs, it falls into representative (A, B, C), which has interpretation $\bot$. To proceed, all three inputs are incremented, since they all contain 1. However, the next state has representative $()$, since a second 1 is only contained in A; since $\mathcal{I}() = \bot$, the output is $\{\!\{\}\!\}$.

This problem arises due to misalignment resulting from empty spaces from the difference operator. To fix this, we can treat these empty spaces separately, and instead fast-forward until a value is ready or we run out of values; equivalently, we delay processing the inputs not involved in the difference until we are past the empty space. These semantics cannot be modeled in an iteration lattice, as they do not encode iterator advancement in their semantics; in every case, all iterators containing the minimum value are incremented. To allow such semantics, we encode advancement in the behavior of iteration machines. For this example, we can assign the behavior $\emptyset(A, B)$ to (A, B, C), meaning that only A and B should be incremented, producing the correct output. The "not ready" behavior $\emptyset$ has the same execution semantics as $\bot$, but it must be distinguished from $\bot$ in IM construction (discussed in the next section).

Disjoint union. In many cases, disjoint union is not combined with other operators and the iteration order does not matter; in such cases, it is best to use two separate loops (the Concat rule). When this does not apply, the disjoint union operator must be handled in the iteration machine. It is treated similarly to union, except that the behavior of nodes where both sides are present only advances one side, leaving the other side to be iterated in the next iteration. This requires $\top$ to have a $\vec{N}$ parameter, like $\emptyset(\vec{N})$.

Multiple cursors. Multiple cursors are necessary in situations where the same input needs to be both advanced and not advanced. For example, consider the expression $(A - B) \cup (C - A)$. In the (A, C) state, the right-hand side of the union is not ready, so we only want to advance the inputs on the right-hand side (A and C). However, A is also present on the left-hand side. To solve this, we must split iteration over A into multiple cursors.

7.3 Generation

Iteration machines can be generated bottom-up, from single set to expressions, using the product construction for DFAs. In particular, the product construction simplifies generation compared to iteration lattices, as getting the correct behavior does not require any special case handling.

Segment rule The iteration machine for a single set A has two nodes: (A) and $()$, where $\mathcal{I}(A) = \top(A), \mathcal{I}() = \bot$. (See Section 7.2 for a description of the notation $\top(\vec{N})$.)

Complement rule The complement of an iteration machine can obtained by simply reversing the behavior of all nodes ($\top$ to $\bot$ and vice versa.) Since complement is not well-defined for multisets, we do not need to handle $\emptyset$.

Product construction To generate the iteration machine for a binary operator $E_1 \circ E_2$, we use a product construction similar to that for DFA intersection and union described below.

Depending on the operator $\circ$, the interpretation for a resulting node $u_1 \cup u_2$ in the product construction is given by $\mathcal{I}_1(u_1) \circ \mathcal{I}_2(u_2)$, defined as follows:

- $\emptyset(\vec{N}) \circ \emptyset(\vec{M}) = \emptyset(\vec{N} \cup \vec{M})$.
- $\emptyset(\vec{N}) \circ v = v \circ \emptyset(\vec{N}) = \emptyset(\vec{N})$, for $v \in \{\top(\vec{M}), \bot\}$, except $\emptyset(\vec{N}) \cap \bot = \bot$.
- $\top(\vec{N}) + \top(\vec{M}) = \top(\vec{N})$. Otherwise, $+$ behaves the same as $\cup$.
- $\top(\vec{N}) - \top(\vec{M}) = \emptyset(\vec{N} \cup \vec{M})$. In set semantics, $\top - \top = \bot$, so that complement is well-defined.
- $\top(\vec{N}) \cap \top(\vec{M}) = \top(\vec{N}) \cup \top(\vec{M}) = \top(\vec{N} \cup \vec{M})$.
- $\top(\vec{N}) \cup \bot = \bot \cup \top(\vec{N}) = \top(\vec{N}) - \bot = \top(\vec{N})$.
- Otherwise, $\mathcal{I}_1 \circ \mathcal{I}_2$ is given by the standard interpretation of $\cup, \cap, -$, etc.

Observe that $\emptyset$ has distinct *construction* semantics from $\bot$ in that it delays all other relations, implementing the iteration control in Section 7.2. The Cartesian product $\times$ behaves the same as intersection, but instead of $u_1 \cup u_2$, the node is instead labeled with $\{R_1 \times R_2 \mid R_1 \in u_1, R_2 \in u_2\}$. These nodes require special handling in iteration.

Algorithm 1 Product construction for IM generation

procedure PRODUCT($N_1, \mathcal{I}_1, N_2, \mathcal{I}_2, \circ$)
  $Q, N, \mathcal{I}, E \leftarrow \{(\top_1, \top_2)\}, \{\}, \{\}, \{\}$ ▷ $\top$ is the maximum element; edge info is used for minimization
  while Q is not empty **do**
    $u_1, u_2 \leftarrow \text{pop}(Q)$
    $l \leftarrow u_1 \cup u_2$, or $\{R_1 \times R_2 \mid R_1 \in u_1, R_2 \in u_2\}$ if $\circ$ is $\times$
    skip to next pair if $l \in N$; otherwise add l to N
    $\mathcal{I}(l) \leftarrow \mathcal{I}_1(u_1) \circ \mathcal{I}_2(u_2)$
    for $r \in l$ **do** ▷ where an input $r \in l$ if r is contained in any product
      $v_1, v_2 \leftarrow \text{floor}_1(u_1 - \{r\}), \text{floor}_2(u_2 - \{r\})$
      add edge $(u_1, u_2) \rightarrow (v_1, v_2)$ to E
      $Q \leftarrow Q \cup \{(v_1, v_2)\}$
  return $N, \mathcal{I}, E$

Algorithm 1 shows product construction. It performs a breadth-first search on the product DFA by simultaneously traversing the left and right iteration machines. For each pair u_1, u_2, the result node is labeled with the union $u_1 \cup u_2$ (if not product), and its interpretation is given above. Then, we proceed on each transition r in $u_1 \cup u_2$. The algorithm terminates when all nodes are discovered.

7.4 Optimization

Consider the two iteration machines (IMs) in Figure 21. Both IMs represent the same expression $A \cap B$, but the second has more nodes. To speed up iteration and reduce the generated code, we want to ensure that a minimal number of nodes is generated. To do this, our system runs a minimization step after product construction.

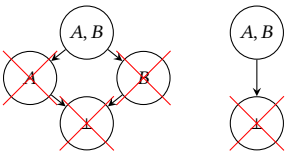


Fig. 21. Two distinct IMs for $A \cap B$, with omitter nodes crossed out. Edge labels and self-loops are omitted.

Consider a node n. It can be removed only if removing it produces a valid IM M', and for all p, $\text{floor}(p) = n \implies (\mathcal{I}'(\text{floor}'(p)) = \mathcal{I}(\text{floor}(p)))$, where $\text{floor}'(p)$ is the node corresponding to p in M'. In fact, this is true whenever $\mathcal{I}(n) = \mathcal{I}'(\text{floor}'(n))$. Observe that removing a node can never cause another node below it (i.e. a subset) to be removable, so removability only flows upwards. Thus, we can minimize iteration machines by traversing from bottom to top; the full algorithm with proof of optimality is in Appendix C.

To process index-only relations, we do the reverse of this process, adding in nodes as necessary to distinguish the specified relations for every node; see Henry et al. [2021] and Appendix E.

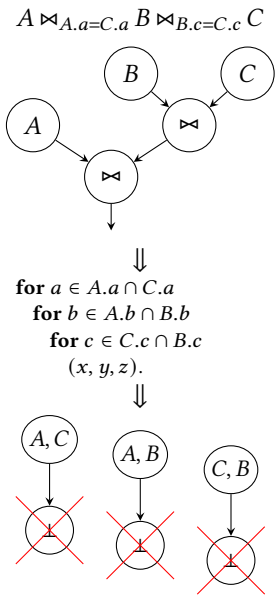
8 Code Generation

To generate executable code from ALIR, two pieces are necessary: an iteration machine for each loop domain, which captures the structure of the generated loops; and the data representation of each relation. Given this information, our system can generate optimized C++ code.

Figure 22 shows an end-to-end example of compiling a triangle join, given a physical query plan and trie storage specification as shown in Figure 5. The compiler lowers the query plan to ALIR, generates an iteration machine for each loop domain, and applies Algorithm 2 generate C++ code.

8.1 Loop generation

Algorithm 2 shows the code generation algorithm for loops. After generating the iteration machine, the algorithm initializes variables for each relation. Then, for every state in the iteration machine in topological order, a loop is generated with the condition corresponding to the required iterators for that state. During iteration, the current value being processed is indicated by the minimum value (*min*) of all iterators. Next, galloping (Appendix E) is performed on relevant iterators to skip



⇒

```
int *ab_a_i = &ab_a[0] + 0; int *ab_a_end = &ab_a[0] + ab_a.size();
size_t *ab_a_ind_i = &ab_a_ind[0] + 0;
int *ac_a_i = &ac_a[0] + 0; int *ac_a_end = &ac_a[0] + ac_a.size();
size_t *ac_a_ind_i = &ac_a_ind[0] + 0;
while (ab_a_i < ab_a_end && ac_a_i < ac_a_end) {
  int aa = std::min(*ab_a_i, *ac_a_i);
  int gallop_aa = std::max(*ab_a_i, *ac_a_i);
  if (aa != gallop_aa) {
    ab_a_i = gallop_lower(ab_a_i, ab_a_end, gallop_aa);
    ab_a_ind_i = &ab_a_ind[ab_a_i - ab_a];
    ac_a_i = gallop_lower(ac_a_i, ac_a_end, gallop_aa);
    ac_a_ind_i = &ac_a_ind[ac_a_i - ac_a]; continue; }
  if (*ab_a_i == aa && *ac_a_i == aa) {
    // vars...
    while (ab_b_i < ab_b_end && bc_b_i < bc_b_end) {
      int ab = std::min(*ab_b_i, *bc_b_i);
      /* ...gallop ab, bc... */
      if (*ab_b_i == ab && *bc_b_i == ab) {
        // vars...
        while (bc_c_i < bc_c_end && ac_c_i < ac_c_end) {
          int bc = std::min(*bc_c_i, *ac_c_i);
          /* ... gallop bc, ac ... */
          if (*bc_c_i == bc && *ac_c_i == bc) {
            emit(aa, ab, bc);
            ++bc_c_i; ++ac_c_i; }
          else { /* dead with galloping */ }
        ++ab_b_i; ++bc_b_i; ++bc_b_ind_i; }
      else { /* dead */ }
    ++ab_a_i; ++ab_a_ind_i; ++ac_a_i; ++ac_a_ind_i; }
  else { /* dead */ }
```

Fig. 22. End-to-end code generation example for triangle join (duplicates loop omitted to reduce the generated code). Left: query plan, ALIR, iteration machines. Right: generated C++ code, simplified for brevity.

Algorithm 2 Code generation for loops

procedure GENERATELOOP(loop domain D, body, storage description for each layer)
  $M \leftarrow$ generate iteration machine for D
  emit $\langle R.\texttt{init}() \rangle$ for each layer R
  for each non-empty state S in M in topological order **do**
    emit **while** $(\bigwedge_{R \in \text{iterators} \cap S} \langle R.\texttt{valid}() \rangle)$ {
    compute minimum value m of $\langle R.\texttt{curval}() \rangle$ for each iterator layer
    compute galloping value g (Subsection **??**)
    if $g \neq -\infty$ **then**
      emit **if** $(m \neq g)$ { advance all iterators to g using `skipto`; **continue** }
    for each reachable state T from S in topological order **do**
      emit **if** $(\bigwedge_{R \in T} \langle R.\texttt{present}(m) \rangle)$ {
      if $\mathcal{I}(T) = \top(\vec{R})$ **then**
        emit body for state T
        advance iterators in $\vec{R}$
      else ▷ $\mathcal{I}(T) = \bot, \emptyset$
        advance appropriate iterators (either all iterators in T or $\vec{R}'$ if $\mathcal{I}(T) = \emptyset(\vec{R}')$.
      emit }
    emit }

past values with forward recursive doubling. Finally, the algorithm emits a series of branches to determine correct behavior based on which relations contain *min*. Within each branch, the loop body is emitted if the corresponding state is a producer, and the appropriate iterators are advanced.

As described in Section 7, multiple concrete loops may be generated for each abstract loop. Each state in the iteration machine results in one loop, unless no producer node is reachable. By ordering the loops in topological order, control flow moves automatically between loops as relations run out of values. With this ordering, the algorithm ensures that the IM edges are followed correctly.

8.2 Case handling

Case handling is similar to loop generation. For each iteration machine state, the generated code has a branch that tests if the current iteration state matches. Since the current value being iterated over is *min*, it tests each input relation R using the method R.`present`(min). A branch matches if all relations in its label are present and the current state does not match a more specific state. To handle overlapping states, it is sufficient to test them from most specific to least specific. The code only needs to handle states reachable from the state corresponding to the current loop, as unreachable states involve an exhausted input. By the structure of iteration machines, a state A is reachable from B only if $A \subseteq B$.

At the end of each case, we need to advance the appropriate iterators. For $\bot$ case, this is every iterator in the corresponding state, and for $\top(\vec{R})$ and $\emptyset(\vec{R})$, this is $\vec{R}$. Product relations (as produced by the product construction for $\times$) require special handling; see `Product.Next` [Root et al. 2024].

9 Evaluation

The purpose of our compiler system is to enable fusing relational operations before lowering to code that operates on specific data structures. Our compilation strategy has four advantages:

(1) The code generator can generate efficient code for any relational algebra operation. In addition the generated code generate code is competitive with leading database systems on the TPC-H queries for which they were optimized (Section 9.1).
(2) The code generator can generate code that is fused through two or more joins to generate worst-case optimal join implementations (Section 9.2).
(3) The code generator generates nested loops that iterate hierarchically over attributes, which enables filter operations to skip over many rows at a time (Section 9.3).
(4) The code generator can generate code that is portable across data structures, which allows for flexibility depending on the application (Section 9.4).

All benchmarks were run on dual Intel Xeon E5-2640 v4 CPUs with 251 GiB of memory. Benchmarks are single-threaded where not otherwise indicated.

To evaluate our approach, we implemented a prototype backend from ALIR and data representations to optimized C++ code using the techniques in Sections 7 and 8. This prototype does not currently support the frontend lowering from query plans to ALIR, but we implemented a frontend that lowers TPC-H SQL queries to ALIR. We then manually optimized this code and hand-wrote ALIR code for other queries than TPC-H. For each database system, we added primary key indices to account for the difference in data representation. Next, to run the queries, we ran our code generator on the ALIR, which produces C++ code. The time taken to generate C++ code from our Python DSL ranged from 0.04s to 0.14s (geomean: 0.06s; see Appendix B), which includes file I/O. Our prototype code generator is written in Python and no effort has been made to optimize its performance, so the compilation speed could be further optimized as necessary.

The generated C++ code was then compiled using g++ 10.5.0 to produce the final executable. Compilation times are excluded from the results. Queries are run three times and averaged. The time spent loading data into our data representation is excluded. Scale factors used for evaluation were 5 for TPC-H and 10 for LSQB. Data for additional scale factors is present in Appendix F.

9.1 Generality and Baseline Performance

Our compiler approach is designed to provide support for fusion and data structure portability across any relational operator. To demonstrate that ALIR does not introduce high overhead in providing these capabilities, we demonstrate that the performance of the generated code is comparable to modern database systems on TPC-H [Council 2022] queries for which they are optimized.

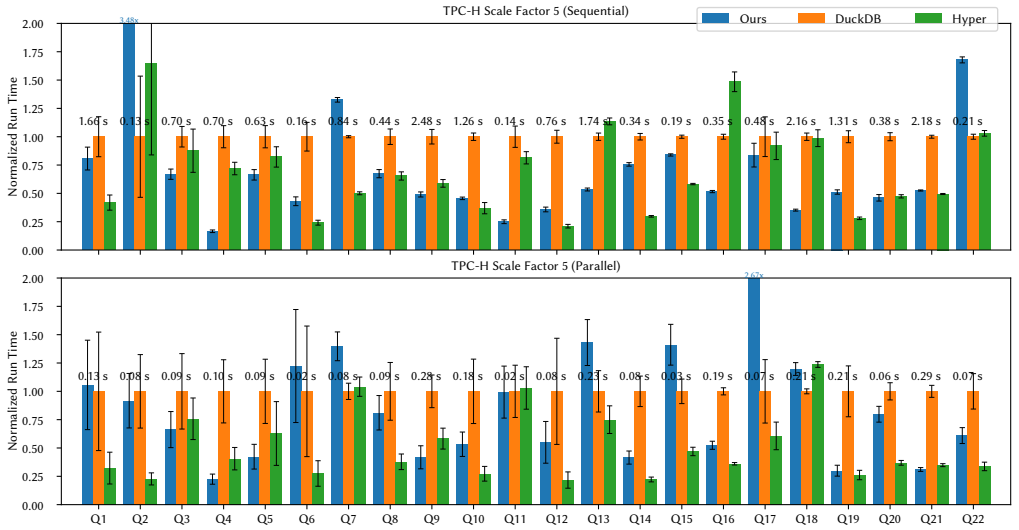


Fig. 23. TPC-H results for scale factor 5. All results normalized to DuckDB.

We evaluate TPC-H queries using our system, DuckDB, Hyper, and hand-written TPC-H queries [Palkar 2017]. As shown in Figure 23, we outperform all databases in 10/22 of the benchmarks when run sequentially, achieving 0.29–4.34× speedup (geomean: 0.96×) compared to the best alternative, with geomean 1.00× compared to Hyper. In parallel, we outperform the other databases in 6 of 22 TPC-H benchmarks, achieving 0.23–1.80× speedup (geomean: 0.60×, vs Hyper: 0.61×).

To demonstrate our compiler's ability to efficiently fuse diverse operations, we also evaluated four queries from the LSQB [Mhedhbi et al. 2021] benchmark, against DuckDB [Raasveldt and Mühleisen 2019] and Hyper [Kemper and Neumann 2011] using the Tableau Hyper API. These queries include diverse operations such as unions, non-inner joins, and anti-joins. Our code generator generates fully fused code, i.e. code that does not require temporary storage, for all queries except Q4 which can be fused but benefits from partial fusion. The results are shown in Figure 24 for both sequential and parallel execution. The LSQB queries end with a count operation in order to not stress client applications. While the other database systems materialize the output tuples, our system optimizes this away and only returns the count. To make the comparison more informative, we configured our compiler to emit each tuple into a temporary buffer. If we let it optimize away this operation, its performance increases by up to 7.5×: sequential speedup is 7.5× for Q6 and 2.9× for Q9 and parallel speedup is 4.8× and 2.6×. Q2/Q4 speedups are within variance.

For query 2, our sequential code generated by our approach is slower than Hyper (0.66×), but achieves a 2.84× parallel speedup. For query 4, sequential runtime is the same as Hyper, but we are slower in parallel (speedup: 0.82×). Queries 6 and 9 are significantly faster: Q6 has a 6.43×/7.61× sequential/parallel speedup, and Q9 has a 14.58×/12.12× sequential/parallel speedup.

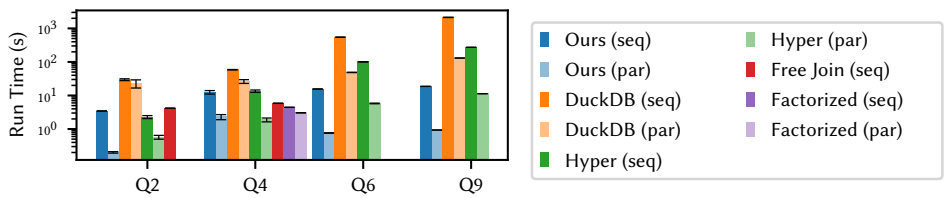


Fig. 24. LSQB results for scale factor 10.

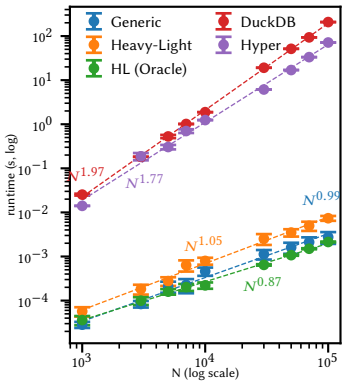


Fig. 25. Triangle query.

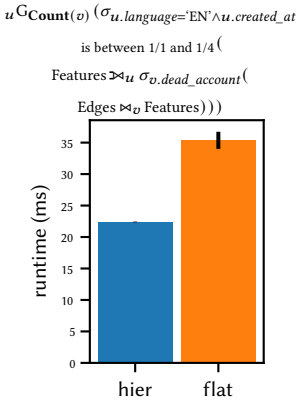


Fig. 26. Hierarchical iteration comparison.

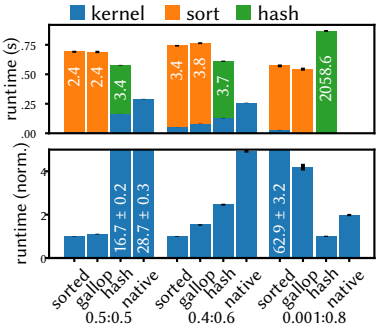


Fig. 27. Joins using intersection. Error bars per segment. Labels in the top chart are profitability cutoff. Bottom chart is normalized to fastest kernel.

We attribute the significant speedup in queries 6 and 9 to a more efficient query plan that uses a multi-way join and hierarchical iteration to improve query performance. The query plans generated by both Hyper and DuckDB compute a binary join between **Person_knows_Person** and **Person_hasInterest_Tag** before performing another join on **Person_knows_Person**. In contrast, our query plan defers iterating through tags until all three people have been determined, and for Q9 it uses a multi-way join with the anti-join on **knows**.

We also compared our approach to Free Join [Wang et al. 2023]. Free Join does not support non-equi joins or non-join operations, so it only applies to queries 1–5 (Q6 requires that person3 ≠ person1). Our approach is 1.21× faster on Q2 and 0.47× slower on Q4. The slowdown on Q4 is due to their use of a factorized output representation, which is orthogonal to the ideas presented in this paper. If we modify the code generated from ALIR to use a similar representation, we get a 1.32× speedup compared to Free Join. These results are mainly due to overhead rather than a different query plans, but we note that our approach fully fuses Q4 while Free Join requires two queries.

9.2 Fusion and Triangle Joins

In addition to the fusion in TPC-H and LSQB queries above, we explicitly demonstrate the advantage of fusion by evaluating our approach on the triangle-finding query [Ngo et al. 2014]: $A(a,b) \bowtie B(b,c) \bowtie C(c,a)$, where $A,B,C = \{(1,i) \mid i \in 1,\ldots,n\} \cup \{(i,1) \mid i \in 1,\ldots,n\}$, which requires $\Theta(n^2)$ runtime for any join-based execution strategy. However, our approach can generate code with optimal asymptotic efficiency $\Theta(n)$ for this query, as shown in Figure 25. As shown, we significantly outperform both DuckDB and Hyper, which exhibit quadratic scaling.

Our ALIR-to-C++ code generator can generate both versions of the triangle join proposed by Ngo et al. [2014]. We compare the two algorithms in Figure 25: the Heavy-Light algorithm, described in Appendix E, and the Generic Join algorithm, described in Figure 12. The heavy-light version is slightly slower (0.47×) than Generic Join, but this is entirely due to the cost of lookups. When hash table lookups are replaced with an arithmetic oracle ($1 \le a,b \le N, \min(a,b) = 1$), heavy-light becomes slightly faster (1.29×) due to the decreased complexity of the loops and because all branches are predictable. This demonstrates the utility of having dynamic selection of data structures.

9.3 Hierarchical Iteration

To clearly show the performance advantage of hierarchical iteration, we compared flat and hierarchical iteration on a simple query, shown in Figure 26, on the Twitch Gamers network [Rozemberczki and Sarkar 2021]. For this query, the filter on u is able to remove a large portion of edges (selectivity

1/159.9), which allows us to skip processing them entirely using hierarchical iteration. For this evaluation, we used column storage for both queries; trie storage would show a greater performance difference for this query, as the non-matching rows could be directly skipped over.

9.4 Data Structures

Databases support several internal data structures for different access patterns. For example, sort-merge join requires both inputs to be sorted and hash join requires one input to be hashed. We compared these algorithms in our system by varying the storage of the joined relations. As shown in Figure 27, we compared sort-merge join, sort-merge join with galloping, and hash join. The results show the runtime of intersecting two relations A and B, each sampling dN elements from $[1, N]$, for three choices of densities $d_A : d_B$. In all cases, $N = 10{,}000{,}000$. The native bar shows a hash join using a C++ `unordered_map`, for when a user already has the data in memory, but not in the system's preferred data structure. The labels in the top chart show the profitability cutoff, i.e. the number of runs before total runtime with conversion becomes profitable over native.

Sort-merge join has excellent performance when the input sizes are similar, but it is unable to exploit sparsity when one of the inputs is very sparse, as shown by the 0.001:0.8 results. Galloping alleviates this problem at a modest performance cost, most visible with a slight imbalance (0.4:0.6) exhibiting worst-case behavior for the cache and branch predictor. Hash join can exploit sparsity in all cases, but it has a higher constant factor than sort-merge join and is slightly slower to build.

One application of data structure portability is the ability to integrate seamlessly with existing systems. For this example, we have added a storage description for C++ `unordered_maps` with lookup capability. This takes a total of 6 lines of Python code. As shown in Figure 27, converting between storage formats can have a significant runtime cost, especially if the data is only used once or a few times, but in all cases the native performance is significantly worse than some other implementation. Depending on the user's requirements, they can choose whether or not to convert based on how many times the query is run for each conversion.

10 Related Work

Our paper describes a relational algebra compiler that can fuse general relational algebra with worst-case optimal join algorithms, which builds on recent work on sparse tensor algebra compilation.

Database management systems. Relational database management systems started with System R [Chamberlin et al. 1981] and Ingres [Stonebraker et al. 1976]. The Volcano model [Graefe 1994] proposed a query execution model where relational operators are organized in a tree where parents request tuples from children. Many modern query execution engines improve on this model in various ways. X100/VectorWise [Boncz et al. 2005] uses vectorized execution [Kersten et al. 2018] on columns to optimize performance. HyPer [Kemper and Neumann 2011] breaks down queries into pipelines of operators separated by pipeline breakers, then compiles these pipelines into native code interspersed with handwritten kernels, using a push-based execution strategy. Peloton [Menon et al. 2017; Pavlo et al. 2017] combines the advantages of vectorization and pipeline fusion using relaxed operator fusion, which splits pipelines into vectorized stages.

Our work generalizes pipeline compilation by allowing fusion across any set of operators. In particular, we allow for fusion of multiple joins as well as other operators such as union and intersection that are traditionally pipeline breakers. However, our work does not explicitly address vectorized execution, and we leave this for future work. Loop transformations for tensor algebra compilation have previously been addressed by Senanayake et al. [2020], and applying this to the relational domain could enable generation of vectorized query code, and a similar approach to Peloton using relaxed fusion could help exploit vectorization opportunities.

Worst-case optimal joins. Worst-case optimal joins (WCOJ), beginning with Generic Join [Ngo et al. 2012], demonstrated that joins over multiple relations can have asymptotically better performance on some queries, such as the triangle query, than traditional query plans with only binary joins. Several databases, such as EmptyHeaded [Aberger et al. 2017] and Umbra [Freitag et al. 2020; Neumann and Freitag 2020] have adopted worst-case optimal joins for their query engines. Leapfrog triejoin [Veldhuizen 2013] gives a simple worst-case optimal join algorithm based on tries, which has been adopted by other approaches that use similar data structures [Kovach et al. 2023; Shaikhha et al. 2022]. Free Join [Wang et al. 2023] improves on Generic Join by allowing iteration over multiple attributes in a single loop, decreasing the preprocessing needed to execute joins. These WCOJ systems all generate code as a series of nested loops, matching the basic structure of our IR.

SDQL [Shaikhha et al. 2022] proposes the use of semi-ring dictionaries to achieve complex fusion of relational and linear algebra optimizations using simple primitives. Its fusion capabilities support the pipeline fusion of HyPer in addition to horizontal and vertical loop fusion for inner joins. By focusing on dictionaries, however, it does not support co-iteration and can thus only synthesize hash joins and not variants of merge joins. While outer hash-joins can be expressed, they require temporary storage and thus cannot be fused. Differences are explicitly not supported, and multiset union/intersection cannot be expressed using the fixed semiring structure.

Our IR enables the compilation of a larger set of fusable operations than are supported by the prior work on worst-case optimal joins, as demonstrated by Table 1. This generality is enabled by the general set of loop domains expressible using ALIR and supported by our code generation algorithms. Compared to prior work on worst-case optimal joins, our IR is more general both in what it can express in its loop domains and in the placement of statements such as aggregate updates and precomputation, enabling a greater degree of fusion.

Sparse Tensor Algebra Compilation. Our work builds on prior work on sparse tensor algebra compilation [Kjolstad et al. 2017]. Multiplications and additions in sparse tensor algebra lead to co-iteration over tensor coordinates similar to loops for inner and outer joins. Chou et al. [2018] extended TACO's coordinate tree abstraction with support for more data structures. We extend the coordinate tree model to represent relations over multisets and show how to represent common database storage formats in this model.

The basic iteration lattice (initially merge lattice) was first introduced in TACO [Kjolstad et al. 2017]. Henry et al. [2021] extended the TACO compilation model with support for non-linear operations, adding the ability to iterate over set complements of tensor coordinates. Root et al. [2024] showed how to represent shape operators using four primitives (collapse, concat, split, slice). Kovach et al. [2023] developed a stream-based model and showed how to represent worst-case optimal joins but is limited to inner equi-joins. Our work generalizes these ideas to all the relational algebra operators used in modern database systems, multiset semantics, and provides encodings of all relational algebra operations. In particular, the generalized iteration lattice [Henry et al. 2021] is insufficient to represent differences in non-iterable universes and does not supports multisets.

11 Conclusion

We have shown how to compile the major operations supported by real-world industry analytics query engines without giving up operator fusion or data structure portability. We hope our work can serve as a stepping stone towards query execution engines that provide full support for modern query languages, support sophisticated optimizations, and are portable across data structures.

Acknowledgements

This work was supported in part by PRISM, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. This work was in part supported by the National Science Foundation under Grant CCF-2143061. This work was also supported by Qualcomm through the Stanford Portal Center.

Data Availability Statement

The source code for our compiler system is available in the accompanying artifact [Dong and Kjolstad 2026]. The artifact includes a compiler for the Python-based ALIR DSL to C++, lowering system from relational algebra to ALIR, some optimization passes, and all code for benchmarks used. It also includes all benchmarking scripts and instructions for running the benchmarks to reproduce the figures in the evaluation. Performance results may vary based on the hardware used.

References

Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: how different are they really?. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 967–980. https://doi.org/10.1145/1376616.1376712

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

Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *Cidr*, Vol. 5. 225–237. https://www.cidrdb.org/cidr2005/papers/P19.pdf

Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. 1981. A History and Evaluation of System R. *Commun. ACM* 24, 10 (oct 1981), 632–646. https://doi.org/10.1145/358769.358784

Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Seattle, Washington, USA) *(PODS '98)*. Association for Computing Machinery, New York, NY, USA, 34–43. https://doi.org/10.1145/275487.275492

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 F Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (1970), 377–387.

Transaction Processing Performance Council. 2022. TPC Benchmark™ H. https://www.tpc.org/tpch/

James Dong and Fredrik Berg Kjolstad. 2026. *Artifact for A Compiler for Fused Relational Operations on Multisets.* https://doi.org/10.5281/zenodo.19662739

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

Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book* (2 ed.). Prentice Hall Press, USA.

G. Graefe. 1994. Volcano— An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.* 6, 1 (feb 1994), 120–135. https://doi.org/10.1109/69.273032

Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. 2007. Architecture of a Database System. *Found. Trends Databases* 1, 2 (Feb. 2007), 141–259. https://doi.org/10.1561/1900000002

Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, and Fredrik Kjolstad. 2021. Compilation of Sparse Array Programming Models. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 128 (oct 2021), 29 pages. https://doi.org/10.1145/3485505

Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. IEEE Computer Society, USA, 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

Fredrik Kjolstad, Willow Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor algebra compilation with workspaces. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 180–192. https://dl.acm.org/doi/10.5555/3314872.3314894

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (oct 2017), 29 pages. https://doi.org/10.1145/3133901

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

Gianfranco Lamperti, Michele Melchiori, and Marina Zanella. 2001. On Multisets in Database Systems. In *Multiset Processing*, Cristian S. Calude, Gheorghe PĂun, Grzegorz Rozenberg, and Arto Salomaa (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 147–215. https://doi.org/10.1007/3-540-45523-X_9

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 (sep 2017), 1–13. https://doi.org/10.14778/3151113.3151114

Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. 2021. LSQB: a large-scale subgraph query benchmark. In *Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (Virtual Event, China) *(GRADES-NDA '21)*. Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. https://doi.org/10.1145/3461837.3464516

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

Hung Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case Optimal Join Algorithms. *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 37–48. https://doi.org/10.1145/2213556.2213565

Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Rec.* 42, 4 (feb 2014), 5–16. https://doi.org/10.1145/2590989.2590991

Shoumik Palkar. 2017. TPC-H Benchmarks. https://github.com/sppalkia/tpch-benches

Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems. In *CIDR*, Vol. 4. 1. https://www.cidrdb.org/cidr2017/papers/p42-pavlo-cidr17.pdf

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

Alexander J Root, Bobby Yan, Peiming Liu, Christophe Gyurgyik, Aart J.C. Bik, and Fredrik Kjolstad. 2024. Compilation of Shape Operators on Sparse Arrays. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 312 (Oct. 2024), 27 pages. https://doi.org/10.1145/3689752

Benedek Rozemberczki and Rik Sarkar. 2021. Twitch Gamers: a Dataset for Evaluating Proximity Preserving and Structural Role-based Node Embeddings. arXiv:2101.03091 [cs.SI]

P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '79)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/582095.582099

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, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. 2022. Functional collection programming with semi-ring dictionaries. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 89 (April 2022), 33 pages. https://doi.org/10.1145/3527333

Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. 1976. The Design and Implementation of INGRES. *ACM Trans. Database Syst.* 1, 3 (sep 1976), 189–222. https://doi.org/10.1145/320473.320476

Apostolos Syropoulos. 2001. Mathematics of Multisets. In *Multiset Processing*, Cristian S. Calude, Gheorghe PĂun, Grzegorz Rozenberg, and Arto Salomaa (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 347–358. https://doi.org/10.1007/3-540-45523-x_17

Susan Tu and Christopher Ré. 2015. DunceCap: Query Plans Using Generalized Hypertree Decompositions. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 2077–2078. https://doi.org/10.1145/2723372.2764946

Todd L. Veldhuizen. 2013. Leapfrog Triejoin: a worst-case optimal join algorithm. arXiv:1210.0481 [cs.DB]

Yisu Remy Wang, Max Willsey, and Dan Suciu. 2023. Free Join: Unifying Worst-Case Optimal and Traditional Joins. *Proc. ACM Manag. Data* 1, 2, Article 150 (June 2023), 23 pages. https://doi.org/10.1145/3589295

Received 2025-11-13; accepted 2026-04-03