A semi-ring dictionary query language for data science

, 48 pages, . ©
doi:

A Semi-ring Dictionary Query Language for Data Science

AMIR SHAIKHHA[1], MATHIEU HUOT[2], SHIDEH HASHEMIAN[1], AMIRALI KABOLI[1], ALEX MASCOLO[1], MILOS NIKOLIC[1], JACLYN SMITH[3], DAN OLTEANU[4]

[1] University of Edinburgh, [2] MIT, [3] University of Oxford, [4] University of Zurich
(e-mail: amir.shaikhha@ed.ac.uk,mhuot@mit.edu,shideh.hashemian@ed.ac.uk, amirali.kaboli@ed.ac.uk, a.mascolo@ed.ac.uk, milos.nikolic@ed.ac.uk,jaclyn.smith@cs.ox.ac.uk,olteanu@ifi.uzh.ch)

Abstract

This article introduces semi-ring dictionaries, a powerful class of compositional and purely functional collections that subsume other collection types such as sets, multisets, arrays, vectors, and matrices. We developed SDQL (Semi-ring Dictionary Query Language), a statically typed language that can express relational algebra with aggregations, linear algebra, and functional collections over data such as relations and matrices using semi-ring dictionaries. Furthermore, thanks to the algebraic structure underlying these dictionaries, SDQL unifies a wide range of optimizations commonly used in databases (DB) and linear algebra (LA). As a result, SDQL enables efficient processing of hybrid DB and LA workloads, by putting together optimizations that are otherwise confined to either DB systems or LA frameworks. We show experimentally that a handful of DB and LA workloads can take advantage of the SDQL language and optimizations. SDQL can compete with or outperform a host of systems that are state of the art in their own domain: in-memory DB system DuckDB for (flat, non-nested) relational data using both traditional query operators and worst-case optimal joins; SciPy for LA workloads; sparse tensor compiler TACO; the Trance nested relational engine; and the in-DB ML engines LMFAO and Morpheus for hybrid DB/LA workloads over relational data.

1 Introduction

The development of domain-specific languages (DSLs) for data analytics has been an important research topic across many communities for more than 40 years. The DB community has produced SQL, one of the most successful DSLs based on the relational model of data (Codd, 1970). For querying complex nested objects, the nested relational algebra (Buneman *et al.*, 1995) was introduced, which relaxes the flatness requirement of the relational data model. The PL community has built language-integrated query languages (Meijer *et al.*, 2006) and functional collection DSLs based on monad calculus (Roth *et al.*, 1988). Finally, the HPC community has developed various linear algebra frameworks for tensors (Vasilache *et al.*, 2018; Kjolstad *et al.*, 2017).

The data science pipeline comprises multiple stages spanning various data collections, including (nested) relations, graphs, and matrices (tensors). Yet, each stage is implemented as a stand-alone component specialized for a particular type of workload. This isolated design results in three main issues. First, the carefully crafted structure of the data is lost through the pipeline, which limits structure-specific optimizations. Second, the data processing components do not benefit from the optimizations developed in the other ones. Finally, there is no way to optimize across the boundary of different components.

The main contribution of this article is SDQL, a purely functional language that is simple, canonical, efficient, and expressive enough for hybrid database (DB) and linear algebra (LA) workloads. In this language, the data is presented as dictionaries over semi-rings, which subsume collection types such as sets, multisets, arrays, and tensors.

Furthermore, SDQL unifies optimizations with inherent similarities that are otherwise developed in isolation. Consider the following relational and linear algebra expressions:

$$Q(a,d) = \Gamma^{\#}_{a,d} R_1(a,b) \bowtie R_2(b,c) \bowtie R_3(c,d)$$

$$N(i,l) = \Sigma_{j,k} M_1(i,j) \cdot M_2(j,k) \cdot M_3(k,l)$$

The expression Q computes the number of paths between each two nodes (a,d) via the binary relations R_1, R_2, and R_3. The expression N computes the matrix representing the multiplication chain of matrices M_1, M_2, and M_3. These expressions are optimized as:

$$Q'(a,c) = \Gamma^{\#}_{a,c} R_1(a,b) \bowtie R_2(b,c) \qquad Q(a,d) = \Gamma^{\#}_{a,d} Q'(a,c) \bowtie R_3(c,d)$$

$$N'(i,k) = \Sigma_j M_1(i,j) \cdot M_2(j,k) \qquad N(i,k) = \Sigma_k N'(i,k) \cdot M_3(k,l)$$

The similarity between these two is not a coincidence; in both cases, two intermediate results are factored out (Q' and N'), thanks to the opportunity provided by the distributivity law. This is because of the semi-ring structure behind both relational and linear algebra: natural number and real number semi-rings. These optimizations are known as *pushing aggregates past joins* (Yan & Larson, 1994) and *matrix chain ordering* (Cormen *et al.*, 2009), respectively.

Contributions. This article makes the following contributions.

- We introduce dictionaries with semi-ring structure (Section 2.3). Semi-ring dictionaries realize the well-known connection between relations and tensors (Abo Khamis *et al.*, 2016).
- We introduce SDQL, a statically typed and functional language over such dictionaries. The kind and type system of SDQL keep track of the semi-ring structure (Section 2). SDQL can be used as an intermediate language for data analytics; programs expressed in (nested) relational algebra (Section 3) or linear algebra-based languages (Section 4) can be translated to SDQL.[1]
- The unified formal model provided by SDQL allows tighter integration of data science pipelines that are otherwise developed in loosely coupled frameworks for different domains. This makes SDQL particularly advantageous for hybrid workloads such as in-DB machine learning and linear algebra over nested biomedical data; SDQL can uniformly apply loop optimizations (including vertical and horizontal loop fusion, loop-invariant code motion, loop factorization, and loop memoization) inside and across the boundary

[1] In this article, by (nested) relational and linear algebra, we mean the corresponding sets of operators presented in Figures 4-7.

of different domains. We also show how we can synthesize efficient query processing algorithms (e.g., hash join and group join) based on these optimizations (Section 5).
- Thanks to the compositional structure of semi-ring dictionaries, SDQL unifies alternative representations for relations: row/columnar vs. curried layouts, and tensors: coordinate (COO) vs. compressed formats (Section 6).
- Using these compositional data layouts, we show how advanced algorithms in databases and linear algebra can be expressed on top of SDQL (Section 7). More specifically, we show the implementation of worst-case optimal join algorithms and tensor algebra over compressed tensor data-layouts.
- We give operational semantics and sound denotational semantics using 0-preserving functions between K-semi-modules and use them for the correctness of SDQL optimizations (Section 8).
- We implemented a prototype compiler and runtime for SDQL (Section 9). We show experimentally (Section 10) that SDQL can be competitive with or outperforms a host of systems that are state-of-the-art in their own domain and that are not designed for the breadth of workloads and data types supported by SDQL. SDQL achieves similar performance to the in-memory DB system DuckDB, while it is faster than a state-of-the-art implementation of worst-case optimal joins. It is on average 2× faster than SciPy for sparse LA and has similar performance to taco for sparse tensors. For nested data, it outperforms the Trance nested relational engine by up to an order of magnitude. For hybrid DB/LA workloads over flat relational data, SDQL has, on average, slightly better performance than the in-DB ML engines LMFAO and Morpheus.

Relation to the Prior Publication. This article extends a previously published paper in OOPSLA'22 (Shaikhha *et al.*, 2022) as follows:

- A new section introducing a worst-case optimal join and a sparse tensor algebra algorithm that leverage compositional data layouts (Section 7).
- The operational semantics for SDQL (Section 8.4), which is used to prove the correctness of our optimizations (Section 8.5).
- Extended experiments with DuckDB query processing (Section 10.2).
- Further experiments comparing our own implementation of a worst-case optimal join algorithm with a state-of-the-art implementation and with DuckDB (Section 10.2).
- Improved experimental results for sparse tensor algebra (Section 10.3).

Motivating Example. The following setting is used throughout the article to exemplify SDQL. Biomedical data analysis presents an interesting domain for language development. Biological data comes in a variety of formats that use complex data models Committee (2005). Consider a biomedical analysis focused on the role of mutational burden in cancer. High tumor mutational burden (TMB) has been shown to be a confidence biomarker for cancer therapy response Fancello *et al.* (2019); Chalmers *et al.* (2017). A subcalculation of TMB is gene mutational burden (GMB). Given a set of genes and variants for each sample, GMB associates variants to genes and counts the total number of mutations present in a given gene per tumor sample. This analysis provides a basic measurement of how impacted a given gene is by somatic mutations, which can be used directly as a likelihood

<table>
<tr><th>Core Grammar</th><th>Description</th></tr>
<tr><td>e ::= sum(x in e) e | { e -> e, ... }</td><td>Dictionary Aggregation & Construction</td></tr>
<tr><td> | { }$_{T,T}$ | e(e)</td><td>Empty Dictionary, Dictionary Lookup</td></tr>
<tr><td> | < a = e, ... > | e.a |</td><td>Record Construction, Field Access</td></tr>
<tr><td> | let x = e in e | x |</td><td>Variable Binding & Access</td></tr>
<tr><td> | if e then e else e | not e</td><td>Conditional, Negation</td></tr>
<tr><td> | e + e | e * e | promote$_{S,S}$(e)</td><td>Addition, Multiplication, Scalar Promotion</td></tr>
<tr><td> | n | r | false | true | c</td><td>Numeric, Boolean, and Other Constants</td></tr>
<tr><td>T ::= { T -> T } | < a:T, ... >
  | S | U</td><td>Dictionary, Record,
Scalar, and Enum Types</td></tr>
<tr><td>S ::= int | real | bool | [Table 1]</td><td>Scalar Semi-Ring Types</td></tr>
<tr><td>U ::= string | dense_int</td><td>String and Dense Integer Types</td></tr>
<tr><td>K ::= Type | SM(S)</td><td>Ordinary & Semi-Module Kinds</td></tr>
</table>

Fig. 1: Grammar of the core part of SDQL. Scalar numeric operations (e.g., `sin`) are omitted for brevity.

measurement for immunotherapy response Fancello *et al.* (2019), or can be used as features to predict patient response to therapy or the severity of the patient's cancer.

The biological community has developed countless DSLs to perform such analyses Masseroli *et al.* (2015); Team (2020); Voss *et al.* (2017). Modern biomedical analyses also leverage SQL-flavoured query languages and machine learning frameworks for classification. An analyst may need to use multiple languages to perform integrative tasks, and additional packages downstream to perform inference. The development of generic solutions that consolidate and generalize complex biomedical workloads is crucial for advancing biomedical infrastructure and analyses.

This article shows the above tasks can be framed in SDQL and benefit from optimized execution.

2 Language

SDQL is a purely functional, domain-specific language inspired by efforts from languages developed in both the programming languages (e.g., Haskell, ML, and Scala) and the databases (e.g., AGCA (Koch *et al.*, 2014) and FAQ (Abo Khamis *et al.*, 2016)) communities. This language is appropriate for collections with *sparse* structure such as database relations, functional collections, and sparse tensors. Nevertheless, SDQL also provides facilities to support dense arrays.

Figure 1 shows the grammar of SDQL for both expressions (`e`) and types (`T`). We first give a background on semi-ring structures. Then, we introduce the kind and type systems of SDQL (cf. Figure 2). Afterwards, we continue by introducing semi-ring and iteration constructs. Finally, we show how arrays and sets are encoded in SDQL.

2.1 Semi-Ring Structures

Semi-ring. A semi-ring structure is defined over a data type `S` with two binary operators + and `*`. Each binary operator has an identity element; $\mathbf{0}_S$ is the identity element for + and $\mathbf{1}_S$ is for `*`. When clear from the context, we use `0` and `1` as identity elements. Furthermore, the following algebraic laws hold for all elements `a`, `b`, and `c`:

```
a + (b+c) = (a+b) + c    0 + a = a + 0 = a       1 * a = a * 1 = a
a + b = b + a            a * (b*c) = (a*b) * c   0 * a = a * 0 = 0
a * (b+c) = a*b + a*c    (a+b) * c = a*c + b*c
```

The last two rules are distributivity laws, and are the base of many important optimizations for semi-ring structures (Aji & McEliece, 2000). Semi-rings with commutative multiplications (`a*b=b*a`) are called commutative semi-rings.

Semi-module. The generalization of commutative semi-rings for containers results in a semi-module. A semi-module over a semi-ring of data type `S` (a `S`-semi-module) is defined with an addition operator between two semi-modules, and a multiplication between a semi-ring element and the semi-module. An example is the vector of real numbers with vector addition and scalar-vector multiplication. The following laws hold for all the elements `u` and `v` in a `S`-semi-module:

```
a * (u + v) = a * u + a * v        (u + v) * a = u * a + v * a
(a + b) * u = a * u + b * u        (a * b) * u = a * (b * u)
```

Tensor product. For two types `T1` and `T2` that are `S`-semi-modules, the tensor product $T1 \otimes_S T2$ is another `S`-semi-module. It comes equipped with a canonical map which we also denote using $*: T1 \times T2 \rightarrow T1 \otimes_S T2$ with the following laws for all elements `u1,u2:T1` and `v1,v2:T2`:

```
u1 * (v1+v2) = u1*v1 + u1*v2      (u1+u2) * v1 = u1*v1 + u2*v1
(u1*a) * v1 = u1 * (a*v1)         1 * u1 = u1
```

2.2 Kind System and Type System

Figure 2 shows the kind/type system of SDQL. The types with a semi-ring structure have the kind `SM(S)`; semi-ring dictionaries with `S`-semi-module value types are also `S`-semi-modules (i.e., they have the kind `SM(S)`). However, dictionaries with value types of the ordinary kind `Type` are of kind `Type`. Similar patterns apply to records.

Example 1. Both types `{ string -> int }` and `<c: int>` are of kind `SM(int)`. However, the types `{string -> string}` and `<d: string>` are of kind `Type`.

The addition of two expressions requires both operands to have the same type of kind `SM(S)`. This means that the body of summation also needs to have a type of kind `SM(S)`. The type system rules for the multiplication operator are defined inductively. Multiplying a scalar with a dictionary results in a dictionary with the same keys, but with the values multiplied with the scalar value. Multiplying a dictionary with another term also results in a dictionary with the same keys, and values multiplied with that term. Note that the multiplication operator is not commutative in general.[2] The typing rules for the multiplication of record types are defined similarly.

[2] To be more precise, the scalar * is commutative, but the tensor product * is commutative up to reordering.

Kind System: $T :: K$

$$\frac{}{S::SM(S)} \qquad \frac{\forall\, i.\ Ti::SM(S)}{\langle a1{:}T1,\dots,an{:}Tn\rangle::SM(S)} \qquad \frac{T1::K \quad T2::SM(S)}{\{T1\to T2\}::SM(S)}$$

$$\frac{T1,\ T2 ::SM(S)}{T1\otimes_S T2::SM(S)} \qquad \frac{}{U::Type} \qquad \frac{\exists\, i.\ Ti::Type}{\langle a1{:}T1,\dots,an{:}Tn\rangle::Type} \qquad \frac{T1::K \quad T2::Type}{\{T1\to T2\}::Type}$$

Type System: $\Gamma \vdash e : T$

$$\frac{c: T}{\Gamma \vdash c: T} \qquad \frac{x: T \in \Gamma}{\Gamma \vdash x: T} \qquad \frac{\Gamma \vdash e1: T1 \quad \Gamma,\ x: T1 \vdash e2: T2}{\Gamma \vdash \mathbf{let}\ x = e1\ \mathbf{in}\ e2: T2}$$

$$\frac{\Gamma \vdash e1: \mathbf{bool} \quad \Gamma \vdash e2: T \quad \Gamma \vdash e3: T}{\Gamma \vdash \mathbf{if}(e1)\ \mathbf{then}\ e2\ \mathbf{else}\ e3: T} \qquad \frac{\Gamma \vdash e: S1}{\Gamma \vdash \mathbf{promote}_{S1,\ S2}(e): S2}$$

$$\frac{\Gamma \vdash e1: \{T1\to T2\} \quad \Gamma,\ x: \langle \mathbf{key}{:}T1, \mathbf{val}{:}T2\rangle \vdash e2{:}T3 \quad T3::SM(S)}{\Gamma \vdash \mathbf{sum}(x\ \mathbf{in}\ e1)\ e2: T3}$$

$$\frac{\Gamma \vdash k1: T1 \quad \Gamma \vdash v1: T2 \quad \dots \quad \Gamma \vdash kn: T1 \quad \Gamma \vdash vn: T2}{\Gamma \vdash \{\ k1 \to v1,\ \dots,\ kn \to vn\ \}: \{\ T1 \to T2\ \}} \qquad \frac{}{\Gamma \vdash \{\}_{T1,T2}:\ \{T1 \to T2\}}$$

$$\frac{\Gamma \vdash e1: \{\ T1 \to T2\ \} \quad \Gamma \vdash e2: T1}{\Gamma \vdash e1(e2): T2} \qquad \frac{\Gamma \vdash e: \mathbf{bool}}{\Gamma \vdash \mathbf{not}\ e: \mathbf{bool}}$$

$$\frac{\Gamma \vdash e1: T1 \quad \dots \quad \Gamma \vdash en: Tn}{\Gamma \vdash \langle a1{=}e1,\dots,an{=}en\rangle: \langle a1{:}T1,\dots,an{:}Tn\rangle} \qquad \frac{\Gamma \vdash e: \langle a1{:}T1,\dots,ak{:}Tk\rangle}{\Gamma \vdash e.ai: Ti}$$

$$\frac{\Gamma \vdash e1,\ e2: T \quad T::SM(S)}{\Gamma \vdash e1\ +\ e2: T} \qquad \frac{\Gamma \vdash e1: T1 \quad \Gamma \vdash e2: T2 \quad T1,\ T2 ::SM(S)}{\Gamma \vdash e1\ *\ e2: T1\otimes_S T2}$$

Definition of $\otimes_S$: $\forall\, i.\ Ti :: SM(S)$

$$S \otimes_S T1 \triangleq T1 \qquad \{T1 \to T2\} \otimes_S T0 \triangleq \{T1 \to T2 \otimes_S T0\}$$

$$\langle a1{:}T1,\dots,an{:}Tn\rangle \otimes_S T0 \triangleq \langle a1{:}T1\otimes_S T0,\dots,an{:}Tn\otimes_S T0\rangle$$

Fig. 2: Kind System and Type System of SDQL.

Example 1 (Cont.). Assume a dictionary term `d` with type `{string->int}`, and a record term `r` with type `<c: int>`. The type of the expression `d*r` is `{string->int}`$\otimes_{\texttt{int}}$`<c: int>`, which is `{ string -> <c: int> }`, as can be confirmed by the typing rules.

2.3 Semi-Ring Constructs

Scalars. Values of type `bool` form the *Boolean Semi-Ring*, with disjunction and conjunction as binary operators, and `false` and `true` as identity elements. Values of type `int` and `real` form *Integer Semi-Ring* ($\mathbb{Z}$) and *Real Semi-Ring* ($\mathbb{R}$), respectively. Table 1 shows an extended set of semi-rings for scalar values. Both addition and multiplication only support elements of the same scalar type.

Promotion. Performing multiplications between elements of different scalar data types requires explicitly *promoting* the operands to the same scalar type. Promoting a scalar term `s` of type `S1` to type `S2` is achieved by $\mathbf{promote}_{S1,S2}(s)$.

Dictionaries. A dictionary with keys of type `K`, and values of type `V` is represented by the data type `{ K -> V }`. The expression `{ k_1 -> v_1, ..., k_n -> v_n }`, constructs a dictionary of `n` elements with keys `k_1, ..., k_n` and values `v_1, ..., v_n`. The expression $\{\}_{K,V}$ constructs an empty dictionary of type `{ K -> V }`, and we might drop the type subscript when it is clear from the context. The expression `dict(k)` performs a lookup for key `k` in the dictionary `dict`.

If the value elements with type `V` form a semi-ring structure, then the dictionary also forms a semi-ring structure, referred to as a semi-ring dictionary (SD) where the addition

Table 1: Different semi-ring structures for scalar types.

Name	Type	Domain	Addition	Mult.	Zero	One	Ring
Real Sum-Product	`real`	$\mathbb{R}$	$+$	$\times$	`0`	1	✓
Int. Sum-Product	`int`	$\mathbb{Z}$	$+$	$\times$	`0`	1	✓
Nat. Sum-Product	`nat`	$\mathbb{N}$	$+$	$\times$	`0`	1	✗
Min-Product	`mnpr`	$(0, \infty]$	min	$\times$	∞	1	✗
Max-Product	`mxpr`	$[0, \infty)$	max	$\times$	`0`	1	✗
Min-Sum	`mnsm`	$(-\infty, \infty]$	min	$+$	∞	`0`	✗
Max-Sum	`mxsm`	$[-\infty, \infty)$	max	$+$	$-\infty$	`0`	✗
Max-Min	`mxmn`	$[-\infty, \infty]$	max	min	$-\infty$	$+\infty$	✗
Boolean	`bool`	$\{T, F\}$	$\vee$	$\wedge$	`false`	`true`	✗

is point-wise, that is the values of elements with the same key are added. The elements of an SD with $\mathbf{0}_\mathtt{V}$ as values are made implicit and can be removed from the dictionary. This means that two SDs with the same set of `k_i` and `v_i` pairings are equivalent regardless of their $\mathbf{0}_\mathtt{V}$-valued `k_j`s.

The multiplication `dict * s`, where `dict` is an SD with `k_i` and `v_i` as keys and values, results in an SD with `k_i` as the keys, and `v_i * s` as the values. For the expression `s * dict`, where `s` is not an SD and `dict` is an SD with keys `k_i` and values `v_i`, the result is an SD with `k_i` as keys and `s * v_i` as values. Note that the multiplication operator is not commutative by default.

Example 2. Consider the following two SDs: `{ "a"->2, "b"->3 }` named as `dict1` and `{ "a"->4, "c"->5 }` named as `dict2`. The result of `dict1+dict2` is `{ "a"->6, "b"->3, "c"->5 }`. This is because `dict1` is equivalent to `{ "a"->2, "b"->3, "c"->0 }` and `dict2` is equivalent to `{ "a"->4, "b"->0, "c"->5 }`, and element-wise addition of them results in `{ "a"->2+4, "b"->3+0, "c"->0+5 }`.

The result of `dict1 * dict2` is `{ "a"->2 * dict2, "b"->3 * dict2 }`. The expression `2 * dict2` is evaluated to `{ "a"->2*4, "c"->2*5 }`. By performing similar computations, `dict1 * dict2` is evaluated to `{ "a"->{ "a"->8, "c"->10 }, "b"->{ "a"->12, "c"->15 } }`. On the other hand, `dict2 * dict1` is `{ "a"->4 * dict1, "c"->5 * dict1 }`. After performing similar computations, the expression is evaluated to `{ "a"->{ "a"->8, "b"->12 }, "c"->{ "a"->10, "b"->15 } }`.

Records. Records are constructed using `< a_1 = e_1, ..., a_n = e_n >` and the field `a_i` of record `rec` can be accessed using `rec.a_i`. When all the fields of a record are `S`-semi-modules, the record also forms an `S`-semi-module.

Example 1 (Cont.). Assume the dictionary `d` with the value `{ "a"->2, "b"->3 }`, and the record `r` with the value `< c=4 >`. The expression `d * r` is evaluated as `{ "a" -> <c=8>, "b" -> <c=12> }`.

2.4 Dictionary Summation

The expression `sum(x in d) e` specifies iteration over the elements of dictionary `d`, where each element `x` is a record with the attribute `x.key` specifying the key and `x.val` specifying the value. One can alternatively use the syntactic sugar `sum(<k,v> in d) e` that binds `k` to `x.key` and `v` to `x.val` (cf. Figure 3). This iteration computes the summation of the result

<table>
<tr><th>Extension</th><th>Definition</th><th>Description</th></tr>
<tr><td>if e_0 then e_1</td><td>if e_0 then e_1 else $\mathbf{0}_{\mathrm{T}}$ where e_1: T</td><td>Single Conditional</td></tr>
<tr><td>{ e_0, ..., e_k }</td><td>{ e_0 -> true,..., e_k -> true }</td><td>Set Construction</td></tr>
<tr><td>dom(e)</td><td>sum(x in e) { x.key }</td><td>Key Set of Dictionary</td></tr>
<tr><td>sum(<k,v> in e)
e_1</td><td>sum(x in e)
let k=x.key in let v=x.val in e_1</td><td>Sum Paired Iteration</td></tr>
<tr><td>range(dn)</td><td>{ 0 -> true, ..., dn-1 -> true }</td><td>Range Construction</td></tr>
<tr><td>[| e_0,...,e_k |]</td><td>{ 0 -> e_0, ..., k -> e_k }</td><td>Array Construction</td></tr>
<tr><td>{ T }</td><td>{ T -> bool }</td><td>Set Type</td></tr>
<tr><td>[| T |]</td><td>{ dense_int -> T }</td><td>Array Type</td></tr>
</table>

Fig. 3: Extended constructs of SDQL.

of the expression `e` using the corresponding addition operator, and by starting from an appropriate additive identity element. In the case that `e` has a scalar type, this expression computes the summation using the corresponding scalar addition operator. If the expression `e` is an SD, then the SD addition is used.

Example 1 (Cont.). Consider the expression `sum(x in d) x.val` where `d` is a dictionary with value of `{ "a" -> 2, "b" -> 3 }`. This expression is evaluated to `5`, which is the result of adding the values `(2 + 3)` in dictionary `d`. Let us consider the expression `sum(<k,v> in d) { k -> v * 2 }`, with the same value as before for `d`. This expression is evaluated to `{ "a" -> 4, "b" -> 6 }`, which is the result of the addition of `{ "a" -> 2*2 }` and `{ "b" -> 3*2 }`.

2.5 Set and Array

Collection types other than dictionaries, such as arrays and sets, can be defined in terms of dictionaries (cf. Figure 3). Arrays can be obtained by using *dense integers* (`dense_int`), which are continuous integers ranging from 0 to k , as keys and the elements of the array as values. Sets can be obtained by using the elements of the set as keys and Booleans as values. Arrays and sets of elements of type `T` are represented as `[| T |]` and `{ T }`, respectively.

3 Expressiveness for Databases

This section analyzes the expressive power of SDQL for database workloads. We start by showing the translation of relational algebra to SDQL (Section 3.1). Then we show the translation of nested relational calculus to SDQL (Section 3.2), followed by the translation of aggregations (Section 3.3).

3.1 Relational Algebra

Relational algebra (Codd, 1970) is the foundation of many query languages used in database management systems, including SQL. In general, a relation $R(a_1, ..., a_n)$ (with set semantics) is represented as a dictionary of type { $<a_1$: A_1, ..., a_n: $A_n>$ -> bool } in SDQL. Figure 4 shows the translation rules for the relational algebra operators. SDQL

Name	Translation
Selection	$[\![\sigma_p(\mathrm{R})]\!]$ = `sum(x in` $[\![\mathrm{R}]\!]$ `) if p(x.key) then { x.key }`
Projection	$[\![\pi_f(\mathrm{R})]\!]$ = `sum(x in` $[\![\mathrm{R}]\!]$ `) { f(x.key) }`
Union	$[\![\mathrm{R} \cup \mathrm{S}]\!]$ = $[\![\mathrm{R}]\!]$ + $[\![\mathrm{S}]\!]$
Intersection	$[\![\mathrm{R} \cap \mathrm{S}]\!]$ = `sum(x in` $[\![\mathrm{R}]\!]$ `) if` $[\![\mathrm{S}]\!]$`(x.key) then { x.key }`
Difference	$[\![\mathrm{R} - \mathrm{S}]\!]$ = `sum(x in` $[\![\mathrm{R}]\!]$ `) if not` $[\![\mathrm{S}]\!]$`(x.key) then { x.key }`
Cartesian Product	$[\![\mathrm{R} \times \mathrm{S}]\!]$ = `sum(x in` $[\![\mathrm{R}]\!]$ `) sum(y in` $[\![\mathrm{S}]\!]$ `) { concat(x.key, y.key) }`
Join	$[\![\mathrm{R} \bowtie_\theta \mathrm{S}]\!]$ = $[\![\sigma_\theta(\mathrm{R} \times \mathrm{S})]\!]$

Fig. 4: Translation from relational algebra (with set semantics) to SDQL.

can also express different variants of joins including outer/semi/anti-joins. The explanation of the relational algebra and various join operators can be found in the extended technical report Shaikhha *et al.* (2021*a*).

Example 3. Consider the following data for the `Genes` input, which is a flat relation providing positional information of genes on the genome:

Genes	name	desc	contig	start	end	gid
	NOTCH2	notch receptor 2	1	119911553	120100779	ENSG00000134250
	BRCA1	DNA repair associate	17	43044295	43170245	ENSG00000012048
	TP53	tumor protein p53	17	7565097	7590856	ENSG00000141510

This relation is represented as follows in SDQL:

```
{ <name="NOTCH2",desc="notch receptor 2", contig=1, start=119911553, end=120100779, gid="ENSG00000134250">,
  <name="BRCA1",desc="DNA repair associate", contig=17, start=43044295, end=43170245, gid="ENSG00000012048">,
  <name="TP53",desc="tumor protein p53", contig=17, start=7565097, end=7590856, gid="ENSG00000141510"> }
```

Only a subset of the attributes in the `Genes` relation are commonly used in a biomedical analysis. This can be achieved using the following expression:

```
sum(<g,v> in Genes) {
    <gene=g.name,contig=g.contig,start=g.start,end=g.end> }
```

Inefficiency of Joins. The presented translation for the join operator is inefficient. This is because one has to consider all combinations of elements of the input relations. In the case of equality joins, this situation can be improved by leveraging data locality as will be shown in Section 5.3.1.

3.2 Nested Relational Calculus

Relational algebra does not allow nested relations; a relation in the first normal form (1NF) when none of the attributes is a set of elements (Codd, 1970). Nested relational calculus allows attributes to be relations as well. In order to make the case more interesting, we consider NRC$^+$ (Koch *et al.*, 2016), a variant of nested relational calculus with *bag semantics* and without difference operator.

Nested relations are represented as dictionaries mapping each row to their multiplicities. As the rows can contain other relations, the keys of the outer dictionary can also contain dictionaries. Figure 5 shows the translation from positive nested relational calculus (without

Name	Translation		
Let Binding	$[\![$`let X =` e_1 `in` $e_2]\!]$	=	`let X =` $[\![e_1]\!]$ `in` $[\![e_2]\!]$
Empty Bag	$[\![\emptyset_T]\!]$	=	`{ }`$_{T,\texttt{int}}$
Singleton Bag	$[\![$`sng(`e`)`$]\!]$	=	`{` $[\![e]\!]$ `-> 1 }`
Flattening	$[\![$`flatten(`e`)`$]\!]$	=	`sum(<k,v> in`$[\![e]\!]$`) v * k`
Monadic Bind	$[\![$`for x in` e_1 `union` $e_2]\!]$	=	`sum(<x,x_v> in`$[\![e_1]\!]$`) x_v *` $[\![e_2]\!]$
Union	$[\![e_1 \uplus e_2]\!]$	=	$[\![e_1]\!] + [\![e_2]\!]$

Cartesian Product	$[\![e_1 \times e_2]\!]$	=	`sum(<x,x_v> in` $[\![e_1]\!]$ `)`
			`sum(<y,y_v> in` $[\![e_2]\!]$ `)`
			`{ <fst=x,snd=y> -> x_v*y_v }`

Fig. 5: Translation from NRC$^+$ (with bag semantics Koch *et al.* (2016)) to SDQL.

difference) to SDQL. The explanation on the translation of its constructs can be found in the extended technical report Shaikhha *et al.* (2021*a*).

Example 4. Consider the `Variants` input, which contains top-level metadata for genomic variants and nested genotype information for every sample. Genotype calls denoting the number of alternate alleles in a sample. An example of the nested `Variants` input is as follows:

Variants	contig	start	reference	alternate	genotypes: sample	genotypes: call
	17	43093817	C	A	TCGA-AN-A046	0
					TCGA-BH-A0B6	1
	1	119967501	G	C	TCGA-AN-A046	1
					TCGA-BH-A0B6	2

This nested relation is represented as follows in SDQL:

```
{ <contig=17, start=43093817, reference="C", alternate="A", genotypes=
    { <sample="TCGA-AN-A046", call=0> -> 1, <sample="TCGA-BH-A0B6", call=1> -> 1 }
         > -> 1,
  <contig=1, start=119967501, reference="G", alternate="C", genotypes=
    { <sample="TCGA-AN-A046", call=1> -> 1, <sample="TCGA-BH-A0B6", call=2> -> 1 }
         > -> 1 }
```

Example 5. The gene burden analysis uses data from `Variants` to calculate the mutational burden for every gene within every sample. The program first iterates over the top-level of `Variants`, iterates over the top-level of `Genes`, then assigning a variant to a gene if the variant lies within the mapped position on the genome. The program then iterates into the nested **genotypes** information of `Variants` to return sample, gene, and burden information; here, the **call** attribute provides the count of mutated alleles in that sample. This expression is represented as follows in NRC$^+$:

```
for v in vcf union for g in genes union
    if (v.contig = g.contig && g.start <= v.start && g.end >= v.start)
    then for c in v.genotypes union
      {sample := c.sample, gene := g.name, burden := c.call}
```

This expression is equivalent to the following SDQL expression (after pushing the multiplication of multiplicities of `Variants` and `Genes` inside the inner singleton dictionary construction):

```
sum(<v,v_v> in Variants) sum(<g,g_v> in Genes)
   if(g.contig==v.contig && g.start<=v.start && g.end>=v.start)
    then sum(<c,c_v> in v.genotypes)
      { <sample = c.sample, gene = g.name, burden = c.call> -> v_v *
          g_v * c_v }
```

The type of this output is `{ <sample:string, gene:string, burden:real> -> int }`.

3.3 Aggregation

An essential operator used in query processing workloads is aggregation. Both relational algebra and nested relational calculus need to be extended in order to support this operator. The former is extended with the group-by aggregate operator $\Gamma_{g;f}$, where g specifies the set of keys that are partitioned by, and f specifies the aggregation function. NRC^{agg} is an extended version of the latter with support for two aggregation operators; $\texttt{sumBy}_g^f$ is similar to group-by aggregates in relational algebra, whereas $\texttt{groupBy}_g$ only performs partitioning without performing any aggregation.

Figure 6 shows the translation of aggregations in relational algebra and NRC^{agg} to SDQL. The explanation of these operators can be found in the extended technical report Shaikhha *et al.* (2021*a*).

Generalized Aggregates. Both scalar and group-by aggregate operators can be generalized to support other forms of aggregates such as minimum and maximum by supplying appropriate semi-ring structure (i.e., addition, multiplication, zero, and one). For example, in the case of maximum, the maximum function is supplied as the addition operator, and the numerical addition needs to be supplied as the multiplication operator (Mohri, 2002). An extended set of semi-rings for scalar values are presented in Table 1. To compute aggregates such as average, one has to compute both summation and count using two aggregates. The performance of this expression can be improved as discussed later in Section 5.1.2.

Inefficiency of Group-by. The translated group-by aggregates are inefficient. This is because relational algebra and NRC need an internal implementation utilizing dictionaries for the grouping phase (i.e., the creation of the variable `tmp` in the second, fourth, fifth rules of Figure 6). Nevertheless, as there is no first-class support for dictionaries, the grouped structure is thrown away when the final aggregate result is produced. This additional phase involves an additional iteration over the elements, as illustrated in the next example.

Example 6. As the final step for computing gene burden, one has to perform sum-aggregate of the genotype call (now denoted burden) for each sample corresponding to that gene. By naming the previous NRC expression as `gv`, the following NRC^{agg} expression specifies the full burden analysis:

```
let gmb = groupBy_sample(gv)
for x in gmb union
  {sample := x.key, burdens := sumBy_gene(x.val)}
```

Name	Translation
Relational Algebra:	
Scalar Agg.	$[\![\Gamma_{\emptyset;f}(e)]\!]$ = `sum(<x,v> in` $[\![e]\!]$`) v *` $[\![f]\!]$`(x)`
Group-by Aggregate	$[\![\Gamma_{g;f}(e)]\!]$ = `let tmp=sum(<x,v> in` $[\![e]\!]$`) {`$[\![g]\!]$`(x)->v*`$[\![f]\!]$`(x)}` `in sum(<x,v> in tmp) { <key=x, val=v> -> 1 }`
NRC^{agg}:	
Scalar Agg.	$[\![\texttt{sumBy}^{f}_{\emptyset}(e)]\!]$ = `sum(<x,v> in` $[\![e]\!]$`) v *` $[\![f]\!]$`(x)`
Group-by Aggregate	$[\![\texttt{sumBy}^{f}_{g}(e)]\!]$ = `let tmp=sum(<x,v> in` $[\![e]\!]$`) {` $[\![g]\!]$`(x) -> v*`$[\![f]\!]$`(x) }` `in sum(<x,v> in tmp) { <key=x, val=v> -> 1 }`
Nest	$[\![\texttt{groupBy}_{g}(e)]\!]$ = `let tmp=sum(<x,v> in` $[\![e]\!]$`) {` $[\![g]\!]$`(x) -> {x -> v} }` `in sum(<x,v> in tmp) { <key=x, val=v> -> 1 }`

Fig. 6: Translation of aggregate operators of RA and NRC^{agg} (Smith *et al.*, 2020) to SDQL.

This expression is translated as the following SDQL expression:

```
let tmp = sum(<x,x_v> in gv) { x.sample -> { x -> x_v } } in
let gmb = sum(<x,x_v> in tmp) { <key=x, val=x_v> -> 1 } in
sum(<x,x_v> in gmb) { <sample = x.key, burdens =
  let tmp1 = sum(<b,b_v> in x.val) { b.gene -> x_v * b_v * b.burden } in
  sum(<t,t_v> in tmp1) { <key=t, val=t_v> -> 1 } > -> 1 }
```

This expression is of type `{ <sample:string,burdens:{<key:string,val:real> -> int}> -> int }`.

4 Expressiveness for Linear Algebra

This section shows the power of SDQL for expressing linear algebra workloads. We first show the representation of vectors in SDQL, followed by the representation of matrices in SDQL. We also show the translation of linear algebra operators to SDQL expressions and their Einstein summation notation referred to as `einsum` in libraries such as numpy.

4.1 Vectors

SDQL represents vectors as dictionaries mapping indices to the element values; thus, vectors with elements of type `S` are SDQL expressions of type `{ int -> S }`. This representation is similar to functional pull arrays in array processing languages (Keller *et al.*, 2010). The key difference is that the size of the array is not stored separately.

Example 7. Consider two vectors defined as $V = \begin{bmatrix} a_0 & 0 & a_1 & a_2 \end{bmatrix}$ and $U = \begin{bmatrix} b_0 & b_1 & b_2 & 0 \end{bmatrix}$. These vectors are represented in SDQL as `{ 0 ->` a_0`, 2 ->` a_1`, 3 ->` a_2 `}` and `{ 0 ->` b_0`, 1 ->` b_1`, 2 ->` b_2 `}`. The expression $V \circ U$ is evaluated to `{ 0 ->` a_0`*`b_0`, 2 ->` a_1`*`b_2`, 3 ->` a_2`*0 }`. As the value associated with the key `3` is zero, this dictionary is equivalent to `{ 0 ->` a_0`*`b_0`, 2 ->` a_1`*`b_2 `}`. This value corresponds to the result of evaluating $V \circ U$, that is the vector $\begin{bmatrix} a_0 b_0 & 0 & a_1 b_2 & 0 \end{bmatrix}$.

Name	Translation	Einsum
Vector Operations:		
Addition	$[\![V_1 + V_2]\!] = [\![V_1]\!] + [\![V_2]\!]$	-
Scal-Vec. Mul.	$[\![a \cdot V]\!] = [\![a]\!]$ `*` $[\![V]\!]$	`,i->i`
Hadamard Prod.	$[\![V_1 \circ V_2]\!] =$ `sum(<i,a> in` $[\![V_1]\!]$ `) { i->a*`$[\![V_2]\!]$`(i) }`	`i,i->i`
Dot Prod.	$[\![V_1 \cdot V_2]\!] =$ `sum(<i,a> in` $[\![V_1]\!]$ `) a *` $[\![V_2]\!]$`(i)`	`i,i->`
Summation	$[\![\sum_{a \in V} a]\!] =$ `sum(<i,a> in` $[\![V]\!]$ `) a`	`i->`
Matrix Operations:		
Transpose	$[\![M^T]\!] =$ `sum(<ij, a> in` $[\![M]\!]$ `)` `{ <row=ij.col, col=ij.row> -> a }`	`ij->ji`
Addition	$[\![M_1 + M_2]\!] = [\![M_1]\!] + [\![M_2]\!]$	-
Scal-Mat. Mul.	$[\![a \cdot M]\!] = [\![a]\!]$ `*` $[\![M]\!]$	`,ij->ij`
Hadamard Prod.	$[\![M_1 \circ M_2]\!] =$ `sum(<ij,a> in` $[\![M_1]\!]$ `)` `{ ij -> a *` $[\![M_2]\!]$`(ij) }`	`ij,ij->ij`

Matrix-Matrix Multiplication	$[\![M_1 \times M_2]\!] =$ `sum(<ij,a> in` $[\![M_1]\!]$`)sum(<jk,b> in` $[\![M_2]\!]$`)`
	`if(ij.col == jk.row) then`
	`{ <row=ij.row,col=jk.col> -> a*b }`
		`ij,jk->ik`

Mat-Vec. Mul.	$[\![M \cdot V]\!] =$ `sum(<ij,a> in` $[\![M]\!]$`)` `{ij.row->a*`$[\![V]\!]$`(ij.col)}`	`ij,j->i`
Trace	$[\![Tr(M)]\!] =$ `sum(<ij,a> in` $[\![M]\!]$`)` `if(ij.row==ij.col) then a`	`ii->`

Fig. 7: Translation of linear algebra operations to SDQL.

4.2 Matrices

Matrices are considered as dictionaries mapping the row and column indices to the element value. This means that matrices with elements of type `S` are SDQL expressions with the type `{ <row: int, col: int> -> S }`. Figure 7 shows the translation of vector and matrix operations to SDQL. We give a detailed explanation of these operators can be found in the extended technical report Shaikhha *et al.* (2021*a*).

Example 8. Consider the following matrix M of size 2×4: $\begin{bmatrix} c_0 & 0 & 0 & c_1 \\ 0 & c_2 & 0 & 0 \end{bmatrix}$. This matrix is in SDQL as `{<row=0,col=0> ->` c_0`,<row=0,col=3> ->` c_1`,<row=1,col=1> ->` c_2`}`. The expression $M \cdot V$ is evaluated to the following dictionary after translating to SDQL: `{ 0 ->` c_0`*`a_0`+`c_1`*`a_2`, 1 ->` c_2`*0 }`. This expression is the dictionary representation of the following vector, which is the result of the matrix-vector multiplication: $\begin{bmatrix} c_0 a_0 + c_1 a_2 & 0 \end{bmatrix}$.

Example 9. Computing the covariance matrix is an essential technique in machine learning, and is useful for training various models (Abo Khamis *et al.*, 2018). The covariance matrix of a matrix A is computed as $A^T A$. In our biomedical example, computing the covariance matrix enables us to train different machine learning models such as linear regression on top of the `Variant` dataset.

Point-wise Operations. In many machine learning applications, it is necessary to support point-wise application of functions such as cos, sin, and tan on matrices. SDQL can easily

Vertical Loop Fusion:		
`let y=sum(<x,x_v> in e1){f1(x)->x_v}` `in sum(<x,x_v> in y){f2(x)->x_v}`	$\leadsto$	`sum(<x,x_v> in e1)` `  { f2(f1(x)) -> x_v }`
`let y=sum(<x,x_v> in e1){x->f1(x_v)}` `in sum(<x,x_v> in y){x->f2(x_v)}`	$\leadsto$	`sum(<x,x_v> in e1)` `  { x -> f2(f1(x_v)) }`
Horizontal Loop Fusion:		

`let y1=sum(x in e1) f1(x) in`
`let y2=sum(x in e1) f2(x) in`
`f3(y1, y2)`
	$\leadsto$	`let tmp = sum(x in e1)`
		`  <y1 = f1(x), y2 = f2(x) >`
		`in f3(tmp.y1, tmp.y2)`

Loop Factorization:		
`sum(x in e1) e2 * f(x)`	$\leadsto$	`e2 * sum(x in e1) f(x)`
`sum(x in e1) f(x) * e2`	$\leadsto$	`(sum(x in e1) f(x)) * e2`
Loop-Invariant Code Motion:		
`sum(x in e1)` `  let y = e2 in f(x, y)`	$\leadsto$	`let y = e2 in` `sum(x in e1) f(x, y)`
Loop Memoization:		

`sum(x in e1)`
`  if(p(x) == e2) then g(x, e3)`
`    g(x, e3)`
	$\leadsto$	`let tmp=sum(x in e1)`
		`    {p(x)->{x.key->x.val}}`
		`in sum(x in tmp(e2)) g(x, e3)`

`sum(x in e1)`
`  if(p(x) == e2) then`
`    f(x)`
	$\leadsto$	`let tmp=sum(x in e1)`
		`  {p(x)->f(x)}`
		`in tmp(e2)`

Fig. 8: Transformation rules for loop optimizations.

support these operators by adding the corresponding scalar functions and using **sum** to apply them at each point.

Inefficiency of Operators. Note that the presented operators are highly inefficient. For example, matrix-matrix multiplication requires iterating over every combination of elements, whereas with a more efficient representation, this can be significantly improved. This improved representation is shown later in Section 6.1.

5 Efficiency

In this section, we present loop optimizations of SDQL. Figure 8 summarizes the transformation rules required for such optimizations.

5.1 Loop Fusion

5.1.1 Vertical Loop Fusion

One of the essential optimizations for collection programs is deforestation (Wadler, 1988; Gill *et al.*, 1993; Svenningsson, 2002; Coutts *et al.*, 2007). This optimization can remove an unnecessary intermediate collection in a vertical pipeline of operators, and is thus named as vertical loop fusion. The benefits of this optimization are manifold. The memory usage

```
let R1 = sum(<r,r_v> in R) { f1(r) -> r_v }    ⇝   sum(<r,r_v> in R)
in sum(<r1,r1_v> in R1) { f2(r1) -> r1_v }           { f2(f1(r)) -> r_v }
```

(a) Vertical fusion of `maps` in functional collections.

```
let R1 = sum(<r,r_v> in R) if(p1(r)) then { r -> r_v }    ⇝
in sum(<r1,r1_v> in R1) if(p2(r1)) then { r1 -> r1_v }

let R1 = sum(<r,r_v> in R) {r -> p1(r)*r_v}    ⇝   sum(<r,r_v> in R)
in sum(<r1,r1_v> in R1) {r1 -> p2(r1)*r1_v}        {r -> p1(r)*p2(r)*r_v}
```

(b) Vertical fusion of `filters` in functional collections.

```
let Vt = sum(<i,a> in V1) { i -> a*V2(i) }    ⇝   sum(<i,a> in V1)
in sum(<i,x1> in Vt) { i -> x1*V3(i) }            { i -> a*V2(i)*V3(i) }
```

(c) Vertical fusion of Hadamard product of three vectors.

```
let Rsum =                                  let Rsc =
 sum(<r,r_v> in R) r.A*r_v in                 sum(<r,r_v> in R)
let Rcount =                          ⇝         < Rsum = r.A*r_v, Rcount = r_v >
 sum(<r,r_v> in R) r_v in                   in Rsc.Rsum / Rsc.Rcount
Rsum / Rcount
```

(d) Horizontal fusion for the average computation.

```
sum(<x,x_v> in NR)                  sum(<x,x_v> in NR)
  sum(<y,y_v> in x.C)     ⇝           x.A * x_v *
    x.A*x_v*y.D*y_v                     (sum(y in x.C) y.D * y_v)
```

(e) Loop factorization for scalar aggregates in nested relations.

```
sum(<x,x_v> in NR)                       sum(<x,x_v> in NR)
 sum(<y,y_v> in x.C)             ⇝        sum(<y,y_v> in x.C)              ⇝
  { x.B -> x.A*x_v*y.D*y_v }               { x.B -> 1 } *x.A*x_v*y.D*y_v

sum(<x,x_v> in NR)                          sum(<x,x_v> in NR)
 {x.B->1}*x.A*x_v*                   ⇝       {x.B -> x.A*x_v*
  (sum(<y,y_v> in x.C) y.D*y_v)                (sum(<y,y_v> in x.C) y.D*y_v)}
```

(f) Loop factorization for group-by aggregates in nested relations.

```
                                                             sum(<x,x_v> in NR)
sum(<x,x_v> in NR)               sum(<x,x_v> in NR)            let E = S(x.B) in
 sum(<y,y_v> in x.C)     ⇝        let E = S(x.B) in     ⇝      x.A*x_v*E*(
  let E = S(x.B) in               sum(<y,y_v> in x.C)            sum(<y,y_v> in x.C)
  x.A*x_v*E*y.D*y_v                x.A*x_v*E*y.D*y_v              y.D*y_v)
```

(g) Loop-invariant code motion for dictionary lookup in nested relations.

Fig. 9: Examples for loop fusion (vertical and horizontal) and loop hoisting in SDQL.

is improved thanks to the removal of intermediate memory, and the run time is improved because the removal of the corresponding loop. In query processing engines, pull and push-based *pipelining* (Neumann, 2011; Ramakrishnan & Gehrke, 2000) has the same role as vertical loop fusion (Shaikhha *et al.*, 2018*b*). Similarly, in functional array processing languages, pull arrays and push arrays (Anker & Svenningsson, 2013; Claessen *et al.*, 2012; Svensson & Svenningsson, 2014) are responsible for fusion of arrays. However, none of the existing approaches support fusion for dictionaries. Next, we show how vertical fusion in SDQL subsumes the existing techniques.

Fusion in Functional Collections. As a classic example in functional programming, a sequence of two `map` operators can be naïvely expressed as the left expression in Figure 9a. There is no need to materialize the results of the first `map` into `R1`. Instead, by applying the first vertical loop fusion rule from Figure 8 one can fuse these two operators and remove the intermediate collection as depicted in the right expression of Figure 9a. Another interesting example is the fusion of two `filter` operators. The pipeline of these operators is expressed as the first SDQL expression in Figure 9b. The conditional construct in both summations can be pushed to the value of dictionary resulting in the second expressions. Finally, by applying the second rule of vertical fusion, the last expression is derived, which uses a single iteration over the elements of `R`, and the result collection has a zero multiplicity for elements where `p1` or `p2` is `false`.

Fusion in Linear Algebra. Similarly, in linear algebra programs there are cases where the materialization of intermediate vectors can be avoided. As an example, consider the Hadamard product of three vectors, which is naïvely translated as the first SDQL expression in Figure 9c. Again, the intermediate vector `Vt` is not necessary. By applying the second vertical loop fusion rule from Figure 8, one can avoid the materialization of `Vt`, as shown in the right expression in Figure 9c. This expression performs a single iteration over the elements of the vector `V1`.

5.1.2 Horizontal Loop Fusion

Another form of loop fusion involves simultaneous iterations over the same collection, referred to as horizontal loop fusion. More specifically, in query processing workloads, there could be several aggregate computations over the same relation. In such cases, one can share the scan over the same relation and compute all the aggregates simultaneously. For example, in order to compute the average, one can use the following two aggregates over the same relation `R`, as shown in the left expression in Figure 9d. In such a case, one can iterate over the input relation only once, and compute both aggregates as a tuple. In this optimized expression (cf. right expression in Figure 9d), the average is computed by dividing the element of the tuple storing summation over the count. This optimization corresponds to *merging a batch of aggregates* over the same relation in databases.

5.2 Loop Hoisting

5.2.1 Loop Factorization

One of the most important algebraic properties of the semi-ring structure is the distributive law, which enables factoring out a common factor in addition of two expressions. This algebraic law can be generalized to the case of summation over a collection (cf. Figure 8).

Consider a nested relation `NR` with type `{<A:real,B:int,C:{<D:real> -> int}> -> int}` where we are interested in computing the multiplication of the attributes `A` and `D`. This can be represented as the left expression in Figure 9e. The subexpression `x.A*x_v` is independent of the inner loop, and can be factored out, resulting in the right expression in the same figure.

This optimization can also benefit expressions involving dictionary construction, such as group by expressions. As an example, consider the same aggregation as before grouped

```
sum(<r,r_v> in R)
 sum(<s,s_v> in S)
  if(r.jr==s.js) then
  {concat(r,s)->r_v*s_v}
```

⇝

```
sum(<r,r_v> in R)
 let Sp =
  sum(<s,s_v> in S)
  {s.js->{s->s_v}} in
 sum(<s,s_v> in
   Sp(r.jr))
 {concat(r,s)->r_v*s_v}
```

⇝

```
let Sp =
 sum(<s,s_v> in S)
 {s.js ->{s->s_v}} in
sum(<r,r_v> in R)
 sum(<s,s_v> in
   Sp(r.jr))
 {concat(r,s)->r_v*s_v}
```

(a) Synthesizing hash join operator from nested loop join.

```
sum(<r,r_v> in R)
 sum(<s,s_v> in S)
  if(r.jr==s.js) then
  { r.jr->f(r)*g(s) }
```

⇝

```
sum(<r,r_v> in R)
 { r.jr-> f(r)*
   (sum(<s,s_v> in S)
    if(r.jr==s.js) then
    g(s) ) }
```

⇝

```
let Sp =
 sum(<s,s_v> in S)
 { s.js -> g(s) } in
sum(<r,r_v> in R)
 { r.jr->f(r)*Sp(r.jr) }
```

(b) Synthesizing groupjoin operator from nested loop join and group-by aggregation.

Fig. 10: Synthesizing hash join and groupjoin operators by loop memoization.

by attribute `B`, represented in the first expression of Figure 9f. According to the semantics of SDQL (cf. Section 8), we can rewrite the dictionary construction resulting in the second expression. Again, we can factor out the terms independent of the inner loop (cf. the third expression). By using the semantics of dictionaries, this expression can be translated to the last expression in Figure 9f. In this expression the intermediate dictionaries corresponding to each group are only constructed for each element of the outer relation, instead of each element of the inner relation.

5.2.2 *Loop-Invariant Code Motion*

In addition to multiplication operands, one can hoist let-bindings invariant to the loop. Consider the following example, where one computes the aggregate `A * E * D` where `E` comes from looking up (using hash join) for another relation `S`, represented as the first expression in Figure 9g. In this case, the computation of `E` of is independent of the inner loop and thus can be hoisted outside following the last rule of Figure 8, resulting in the middle expression. Additionally, this optimization enables further loop factorization, which results in the last expression in Figure 9g.

5.3 *Loop Memoization*

In many cases, the body of loops cannot be easily hoisted. Such cases require further memoization-based transformations on the loop body to make them independent of the loop variable, referred to as loop memoization.

5.3.1 *Synthesizing Hash Join*

In general, we can produce a nested dictionary by memoizing the inner loop. Then, instead of iterating the entire range of inner loop, only iterate over its relevant partition. Consider again the case of equality join between two relations `R` and `S` (cf. Section 3.1) based on the join keys `r.jr` and `s.js`, represented as the first expression in Figure 10a. This expression is inefficient, due to iterating over every combination of the elements of the two input relations. The body of the conditional is however dependent on the outer loop and thus cannot be hoisted outside. Applying the first loop memoization rule results in the middle expression; in order to join the two relations, it is sufficient to iterate over relation `R` and find the corresponding partition from relation `S` by using `Sp(r.jr)`. In this expression, the dictionary `Sp` is no longer dependent on `r`. Thus, we can perform loop-invariant code motion, which results in the last expression.

In the specific case of implementing a dictionary using a hash-table, this join algorithm corresponds to a hash join operator; The first loop corresponds to the *build phase* and the second loop corresponds to the *probe phase* (Ramakrishnan & Gehrke, 2000). This expression is basically the same expression as the one for the hash join operator. This means that the first rewrite rule of loop memoization when combined with loop hoisting *synthesizes hash join operator*.

Example 5 (Cont.). Let us consider again the join between `Gene` and `Variants`. The previous expression used nested loops in order to handle join, which is inefficient. The following expression uses hash join instead:

```
let Vp = sum(<v,v_v> in Variants)
  { v.contig -> {<start=v.start,genotypes=v.genotypes> -> v_v} } in
sum(<g,g_v> in Genes) sum(<v,v_v> in Vp(g.contig)) sum(<m,m_v> in
     v.genotypes)
  if(g.start<=v.start&&g.end>=v.start) then
    { <sample=m.sample,gene=m.gene,burden=m.call> -> g_v*v_v*m_v }
```

5.3.2 Synthesizing Groupjoin

There are special cases, where the loop memoization can perform even better. This achieved by performing a portion of computation while partitioning the data. This situation arises when computing an aggregation over the result of join between two relations. As an example, consider the summation of `f(r) * g(s)` on the elements `r` and `s` that successfully join, grouped by the join key, represented as the last expression of Figure 10b. In this case, the inner **`sum`** contains the terms `f(r)` and `r.jr` which are dependent on `r` and thus makes it impossible to be hoisted. The terms `r.jr` and `f(r)` inside the conditional body can be factored outside using the loop factorization rule, resulting in the middle expression. Afterwards, by applying the second rule of loop memoization, the dictionary bound to variable `Sp` is constructed. As this dictionary is no longer dependent on `r`, we can apply loop-invariant code motion, resulting in the last expression.

In fact, the result expression corresponds to the implementation of a groupjoin operator (Moerkotte & Neumann, 2011). In essence, the loop memoization and loop hoisting optimizations have the effect of *pushing aggregations past joins* (Yan & Larson, 1994).

5.3.3 Memoization Beyond Databases

In the case of max-product semi-ring (cf. Figure 1), these optimizations *synthesize variable elimination* for maximum a priority (MAP) inference in Bayesian networks (Abo Khamis *et al.*, 2016; Aji & McEliece, 2000). Furthermore, *loop normalization* (Shaikhha *et al.*, 2019) can also be considered a special case of this rule.

Optimization \ Feature	Purely functional	Dict. lookup	Dict. summation	Semi-ring	Compositional
Vertical loop fusion	✓	✓	✓		
Horizontal loop fusion	✓		✓		
Memoization	✓	✓	✓		
Loop factorization	✓		✓	✓	
Code motion	✓		✓		
Data layouts					✓

Fig. 11: The features of SDQL leveraged by each transformation.

5.4 Putting all Together

In this section, we investigate the design decisions behind SDQL that enables the optimizations presented before. The features of SDQL can be categorized as follows:

- **Purely functional:** SDQL does not allow any mutation and global side effect.
- **Dictionary lookup:** the dictionaries support a constant-time look up operation.
- **Dictionary summation:** iteration over dictionaries allows for both scalar aggregates and dictionary construction in the style of monoid comprehensions (Fegaras & Maier, 2000).
- **Semi-ring:** SDQL has constructs with such structure including semi-ring dictionaries.
- **Compositional:** semi-ring dictionaries accept semi-ring dictionaries as both keys and values.

Figure 11 shows the features that are leveraged by each loop optimization. The compositionality feature is essential for expressing various data layout representations, which is presented next.

6 Data Layout Representations

In this section, we investigate various data representations supported by SDQL, and show their correspondence to existing data formats used in query engines and linear algebra frameworks.

6.1 Flat vs. Curried Representation

Currying a function of type `T1×T2 => T3` results in a function of type `T1 => (T2 => T3)`. Similarly, dictionaries with a pair key can be curried into a nested dictionary. More specifically, a dictionary of type `{ <a: T1, b: T2> -> T3 }` can be curried into a dictionary of type `{ T1 -> { T2 -> T3 } }`.

6.1.1 Factorized Relations

Relations can be curried following a specified order for their attributes. In the database community, this representation is referred to as *factorized representation* (Olteanu & Schleich, 2016) using a *variable order*. In practice, a trie data structure can be used for factorized representation, and has proved useful for computational complexity improvements for joins, resulting into a class of join algorithms referred to as worst-case optimal joins (Veldhuizen, 2014), presented in Section 7.1.

Consider a relation $R(a_1, ..., a_n)$ (with bag semantics), the representation of which is a dictionary of type $\texttt{\{ <}a_1\texttt{:}A_1\texttt{,...,}a_n\texttt{:}A_n\texttt{> -> }\textbf{int}\texttt{ \}}$ in SDQL. By using the variable order of $[a_1, ..., a_n]$, the factorized representation of this relation in SDQL is a nested dictionary of type $\texttt{\{}A_1\texttt{->\{...->\{}A_n\texttt{->}\textbf{int}\texttt{\}...\}\}}$.

6.1.2 Curried Matrices

Matrices can also be curried as a dictionary with row as key, and another dictionary as value. The inner dictionary has column as key, and the element as value. Thus, a curried matrix with elements of type `S` is an SDQL expression of type `{ int -> { int -> S } }`.

Example 8 (Cont.). Consider matrix M from Example 8. The curried representation of this matrix in SDQL is $\texttt{\{ 0 -> \{ 0 -> }c_0\texttt{, 3 -> }c_1\texttt{ \}, 1 -> \{ 1 -> }c_2\texttt{ \} \}}$.

The flat encoding of matrices presented in Section 4.2 results in inefficient implementation for various matrix operations, as explained before. Using a curried representation instead, one can provide more efficient implementations for matrix operations, presented in Section 7.2.

Correspondence to Tensor Formats. The flat representation corresponds to the COO format of sparse tensors, whereas the curried one corresponds to CSF using hash tables Chou *et al.* (2018).

6.2 Sparse vs. Dense Layouts

6.2.1 Sparse Layout

So far, all collections were encoded as dictionaries with hash table as their underlying implementations. This representation is appropriate for sparse structures, but it is suboptimal for dense ones; typically linear algebra frameworks use arrays to store dense tensors.

6.2.2 Dense Layout

SDQL can leverage **`dense_int`** type in order to use array for implementing collections. As explained in Section 2, arrays are the special case of dictionaries with **`dense_int`** keys. The runtime environment of SDQL uses native array implementations for such dictionaries instead of hash-table data-structures. Thus, by using **`dense_int`** as the index for tensors, SDQL can have a more efficient layout for dense vectors and matrices. In this way, a vector is encoded as an array of elements and a matrix as a nested array of elements.

Next, we see how dense layout and in particular arrays can be used to implement row and columnar layout for query engines.

6.3 Row vs. Columnar Layouts

6.3.1 Row Layout

In cases where input relations do not have duplicates, there is no need to keep the boolean multiplicity information in the corresponding dictionaries. Instead, relations can be stored

<table>
<tr><th>Dictionary</th><th>Factorized</th></tr>
<tr><td>
<table>
<tr><td><A=a_1, B=b_1></td><td>1</td></tr>
<tr><td><A=a_1, B=b_2></td><td>1</td></tr>
<tr><td><A=a_2, B=b_3></td><td>1</td></tr>
</table>
</td><td>
<table>
<tr><td rowspan="2">a_1</td><td>b_1</td><td>1</td></tr>
<tr><td>b_2</td><td>1</td></tr>
<tr><td>a_2</td><td>b_3</td><td>1</td></tr>
</table>
</td></tr>
<tr><th>Row</th><th>Columnar</th></tr>
<tr><td>
<table>
<tr><td>0</td><td><A=a_1, B=b_1></td></tr>
<tr><td>1</td><td><A=a_1, B=b_2></td></tr>
<tr><td>2</td><td><A=a_2, B=b_3></td></tr>
</table>
</td><td>
<A=
<table>
<tr><td>0</td><td>a_1</td></tr>
<tr><td>1</td><td>a_1</td></tr>
<tr><td>2</td><td>a_2</td></tr>
</table>
, B=
<table>
<tr><td>0</td><td>b_1</td></tr>
<tr><td>1</td><td>b_2</td></tr>
<tr><td>2</td><td>b_3</td></tr>
</table>
>
</td></tr>
</table>

Fig. 12: Different data layouts for relations.

as dictionaries where the key is an index, and the value is the corresponding row. This means that the relation $R(a_1, ..., a_n)$ can be represented as a dictionary of type `{ idx_type -> {`a_1`:` A_1`, ...,` a_n`:` A_n`} }`. The key (of type `idx_type`) can be an arbitrary *candidate key*, as it can uniquely specify a row. By using **`dense_int`** type as the key of this dictionary, the keys are consecutive integer values starting from zero; thus, we encode relations using an array representation. This means that the previously mentioned relation becomes an array of type `[|<`a_1`:` A_1`, ...,` a_n`:` A_n`>|]`.

6.3.2 Columnar Layout

Column store (Idreos *et al.*, 2012) databases represent relations using vertical fragmentation. Instead of storing all fields of a record together as in row layout, columnar layout representation stores the values of each field in separate collections.

In SDQL, columnar layout is encoded as a record where each field stores the array of its values. This representation corresponds to the array of struct representation that is used in many high performance computing applications. Generally, the columnar layout representation of the relation $R(a_1, ..., a_n)$ is encoded as a record of type `<`a_1`: [|`A_1`|], ...,` a_n`: [|`A_n`|]>` in SDQL.

7 Advanced Algorithms

In this section, we show how we can use the compositional data layouts to express advanced algorithms for join processing and tensor processing in SDQL.

7.1 Worst-case Optimal Join Algorithms

Worst-case optimal join (WCOJ) algorithms are a relatively recently developed class of algorithms for efficient join processing. For certain class of queries, especially the ones that involve cyclic joins, these algorithms are asymptotically faster than traditional join algorithms.

Generic Join is the simplest and most widely adopted algorithm for WCOJ. The basic idea is to perform a multi-way join among several relations at the same time. This is achieved by nested iteration over different attributes. At each iteration, Generic Join computes the intersection of all relations based on the corresponding attribute.

```
let TH = sum(<t, Tv> in T)
  {t.x -> {<t.x,t.z> -> Tv}} in
let RT = sum(<r, Rv> in R)
  sum(<t, Tv> in TH(r.x))
  {r.y -> {<t.x, r.y, t.z> -> Rv*Tv}}
in sum(<s, Sv> in S)
  sum(<rt, RTv> in RT(s.y))
  if(rt.z == s.z && rt.y == s.y) then
  {<rt.x, rt.y, rt.z> -> Sv*RTv}
```

(a) Traditional hash join.

```
sum(<x, Rx> in R_trie)
  let Tx = T_trie(x) in
  if(Tx != {}) then
    sum(<y, Rv> in Rx)
      let Sy = S_trie(y) in
      if(Sy != {}) then
        sum(<z, Sv> in Sy)
          let Tv = Tx(z) in
          {<x, y, z> -> Rv*Sv*Tv}
```

(b) Generic join.

Fig. 13: Traditional joins vs. Generic Join implementation of a cyclic query that joins three relations R(x,y), S(y,z), T(z,x).

In SDQL, by representing relations as a trie, each level of trie corresponds to one attribute. Then, iterations over different attributes and computing the intersections can be expressed by **sum** and dictionary lookups over different levels of tries.

Figure 13 shows the implementation of traditional hash join and Generic Join for a cyclic query between three relations with the following SQL query:

```
-- schema: R(x,y), S(y,z), T(z,x)
SELECT R.x, S.y, T.z FROM R, S, T
WHERE R.y = S.y AND S.z = T.z AND R.x = T.x
```

The traditional join implementation uses two binary hash joins. It involved first building a hash table based on the relation `T`. Then, it iterates over the relation `R` and probes the relevant elements from the built hash table. During this process, it builds the hash table required for joining with relation `S`. Finally, it iterates over the elements of relation `S`, and probes the relevant elements from the second hash table.

The Generic Join implementation builds the trie representation for all three relations based on the following variable order: `x, y, z`. Then, it starts with iterating over the variable `x` from relation `R` and intersects its elements with the ones from relation `T`. Afterwards, it performs a similar iteration by iterating over the variable `y` by the elements from the relation `R` and intersects the elements with the relation `S`. Finally, it performs a similar task for the variable `z`.

7.2 Sparse Tensor Algebra Algorithms

A similar idea to worst-case optimal joins can be employed for tensors. Figure 14 shows the translation of matrix operators, using curried representation. In this representation, one can remove unnecessary iterations over matrices. This is achieved by turning **sum**s followed by equality conditionals into dictionary lookups.

As an example, let us consider the curried matrix-matrix multiplication. Instead of iterating over every combination of elements of two matrices, the curried representation allows a direct lookup on the elements of a particular row of the second matrix. Assuming that the dimension of the first matrix is $m \times n$, and the second matrix is of dimension $n \times k$, this improvement reduces the complexity from $O(mn^2k)$ down to $O(mnk)$.

Name	Translation	Einsum
Transpose	$[\![M_1^T]\!]$ = `sum(<i,row> in` $[\![M_1]\!]$ `) sum(<j,a> in row) { j -> { i -> a } }`	`ij->ji`
Hadamard Product	$[\![M_1 \circ M_2]\!]$ = `sum(<i,row> in` $[\![M_1]\!]$`) { i -> sum(<j,a> in row) { j -> a*`$[\![M_2]\!]$`(i)(j) } }`	`ij,ij->ij`
Matrix-Matrix Multiplication	$[\![M_1 \times M_2]\!]$ = `sum(<i,row> in` $[\![M_1]\!]$ `) { i -> sum(<j,a> in row) sum(<k,b> in` $[\![M_2]\!]$`(j)) { k -> a*b } }`	`ij,jk->ik`
Mat-Vec. Mul.	$[\![M \cdot V]\!]$ = `sum(<i,row> in` $[\![M]\!]$ `) { i -> sum(<j,a> in row) a *` $[\![V]\!]$`(j) }`	`ij,j->i`
Trace	$[\![Trace(M)]\!]$ = `sum(<i,row> in` $[\![M]\!]$ `) row(i)`	`ii->`

Fig. 14: Translation of curried matrix operations to SDQL.

Example 9 (Cont.). The computation of the covariance by curried matrices can be optimized as:

```
let At = sum(row in A) sum(x in row.val) { x.key -> {row.key -> x.val }
    } in
sum(row in At){ row.key -> sum(x in row.val) sum(y in
    A(x.key)){y.key->x.val*y.val} }
```

Furthermore, performing vertical loop fusion results in the following optimized program:

```
sum(row in A) sum(x in row.val) { x.key -> sum(y in
    row.val){y.key->x.val*y.val} }
```

8 Semantics

SDQL is mainly a standard functional programming language, but we study its specificity in this section. First, we show its typing/kinding properties. We then introduce a denotational semantics for SDQL that sheds another light on the language and helps us prove the correctness of the transformation rules presented in Section 5. We give the type safety proofs. Finally, we present the operational semantics of SDQL, based on which we give alternative proofs for the correctness of optimizations.

8.1 Typing

SDQL satisfies the following essential typing properties.

Lemma 8.1. *Let* $\mathbf{T}$ *denote the set of all types of SDQL.* $\otimes$ *is a well-defined partial operation* $\mathbf{T} \times \mathbf{T} \to \mathbf{T}$.

Proposition 8.2. *Every type/term defined using the inference rules of Figure 2 has a unique kind/type.*

Proof [Sketch] By induction on the structure of types/terms and case analysis on each kinding/typing rule. It is straightforward for most rules using the induction hypothesis. For the typing rules of dictionaries there are two cases on whether the dictionary is empty or not, and the type annotation ensures the property for the empty dictionary. As for **`sum`** and **`let`** which have a bound variable, we use the induction hypothesis on `e1` first. ■

8.2 Denotational Semantics

The kind system acts as a type refinement machinery. Roughly, a type is to be considered by default of kind `Type`. Otherwise, the kind indicates that the type carries more structure, more precisely that of a semi-module. More formally, the interpretation of types is given by induction on the kinding rules, and is shown in Figure 15. A type of kind `Type` is interpreted as a set, while a type of kind `SM(S)` is interpreted as a S-semi-module. A scalar type `S` represents a semi-ring and is therefore canonically a S-semi-module. A product of S-semi-modules is a semi-module, and so is the tensor product $\otimes_S$ of two S-semi-modules. One way to describe $\otimes_S$ is as the bifunctor on the category of S-semi-modules and S-module homomorphisms that classifies S-bilinear maps. It is an analogue for semi-modules to the tensor product of vector spaces. For more details on tensor products see e.g. Conrad (2018). The interpretation for a dictionary type is analogous to a free vector space on $|T1|$, in which every element is a finite formal sum of elements of $[\![\texttt{T2}]\!]$. One can show by induction that all our types of kind SM(S) are free S-semi-modules. Hence $[\![\texttt{T2}]\!]$ is a free S-semi-module and this implies that the interpretation for a dictionary type can itself be seen as a free S-semi-module.

For the semantics of environments Γ =`x1:T1, ..., xn:Tn`, we use:

$$[\![\Gamma]\!] \quad = \quad [\![\texttt{T1}]\!] \times \ldots \times [\![\texttt{Tn}]\!]$$

A term $[\![\Gamma \vdash \texttt{e: T}]\!]$ is interpreted as a function from $[\![\Gamma]\!]$ to $[\![\texttt{T}]\!]$. When it is clear from the context, we use $[\![\texttt{e}]\!]$ instead of $[\![\Gamma \vdash \texttt{e: T}]\!]$. We use the notation `v•k` to mean the vector whose only non-zero component `v` is at position `k` in $\bigoplus_{a \in |T1|} [\![\texttt{T2}]\!]$. We denote by γ any assignment of the variables of a context Γ. The denotational semantics for terms is shown in Figure 15. $Prom_{\texttt{S1}\to\texttt{S2}}$ maps the elements of the scalar semi-ring `S1` to `S2`. Every scalar type `S` is a semi-ring and as such admits distinguished elements `0` and `1`. The action of `S` on a type `T`::`SM(S)` thus restricts to an action `*` of the booleans on `T`. This gives the presented description to the semantics of conditionals which we use in the next section. For the semantics for dictionaries, we use a formal infinite sum, but similarly to standard polynomials this sum actually has a finite support and thus behaves like a finite sum in all contexts. For the semantics of **`sum`**, we apply the semantics of `e2` component-wise to the formal sum that is the semantics of `e1`. The resulting real sum is thus over a finite support, and is therefore well-defined.

Proposition 8.3 (Substitution lemma)**.** *For all* $\Gamma \vdash \texttt{e1: T1}$ *and* $\Gamma, \texttt{x: T1} \vdash \texttt{e2: T2}$*, the following holds:* $[\![\texttt{e2}]\!][[\![\texttt{e1}]\!]/x] = [\![\texttt{e2[e1/x]}]\!]$.

$$
\begin{array}{llll}
[\![\mathtt{S}]\!] \triangleq (S, +, 0) & & [\![\mathtt{<a1:T1,\ ...,\ an:Tn>}]\!] \triangleq [\![\mathtt{T1}]\!] \times \ldots \times [\![\mathtt{Tn}]\!] & \\
[\![\mathtt{T1} \otimes_S \mathtt{T2}]\!] \triangleq [\![\mathtt{T1}]\!] \otimes_S [\![\mathtt{T2}]\!] & & [\![\mathtt{\{T1\ ->\ T2\}}]\!] \triangleq \bigoplus_{a \in |T1|} [\![\mathtt{T2}]\!] & \\
\hline
[\![\mathtt{x}]\!]_\gamma & \triangleq \gamma(\mathtt{x}) & [\![\mathtt{<a1=e1,...,an=en>}]\!]_\gamma & \triangleq \mathtt{<}[\![\mathtt{e1}]\!]_\gamma, \ldots, [\![\mathtt{en}]\!]_\gamma\mathtt{>} \\
[\![\mathtt{c}]\!]_\gamma & \triangleq \mathtt{c} & [\![\mathbf{let}\ \mathtt{x = e1}\ \mathbf{in}\ \mathtt{e2}]\!]_\gamma & \triangleq [\![\mathtt{e2}]\!]_{\gamma[[\![\mathtt{e1}]\!]_\gamma/\mathtt{x}]} \\
[\![\mathbf{true}]\!]_\gamma & \triangleq 1 & [\![\mathbf{promote}_{\mathtt{S1,S2}}\mathtt{(e)}]\!]_\gamma & \triangleq Prom_{\mathtt{S1}\to\mathtt{S2}}([\![\mathtt{e}]\!]_\gamma) \\
[\![\mathbf{false}]\!]_\gamma & \triangleq 0 & & \\
[\![\mathbf{not}\mathtt{(e)}]\!]_\gamma & \triangleq 1 - [\![\mathtt{e}]\!]_\gamma & [\![\mathbf{if}\ \mathtt{e1}\ \mathbf{then}\ \mathtt{e2}\ \mathbf{else}\ \mathtt{e3}]\!]_\gamma & \triangleq [\![\mathtt{e1}]\!]_\gamma * [\![\mathtt{e2}]\!]_\gamma + (1 - [\![\mathtt{e1}]\!]_\gamma) * [\![\mathtt{e3}]\!]_\gamma \\
[\![\mathtt{e.ai}]\!]_\gamma & \triangleq \pi_i([\![\mathtt{e}]\!]_\gamma) & [\![\mathtt{e1(e2)}]\!]_\gamma & \triangleq \pi_{[\![\mathtt{e2}]\!]_\gamma}([\![\mathtt{e1}]\!]_\gamma) \\
[\![\mathtt{op(e)}]\!]_\gamma & \triangleq \mathtt{op}([\![\mathtt{e}]\!]_\gamma) & [\![\mathtt{\{\}}_{\mathtt{T1,T2}}]\!]_\gamma & \triangleq 0_{\mathtt{\{T1->T2\}}} \\
[\![\mathtt{e1\ +\ e2}]\!]_\gamma & \triangleq [\![\mathtt{e1}]\!]_\gamma + [\![\mathtt{e2}]\!]_\gamma & [\![\mathtt{\{k1->v1,...,kn->vn\}}]\!]_\gamma & \triangleq \sum_{i \in [1..n]} [\![\mathtt{vi}]\!]_\gamma \bullet [\![\mathtt{ki}]\!]_\gamma \\
[\![\mathtt{e1\ *\ e2}]\!]_\gamma & \triangleq [\![\mathtt{e1}]\!]_\gamma * [\![\mathtt{e2}]\!]_\gamma & & \\
\end{array}
$$

$$[\![\mathbf{sum}\mathtt{(x}\ \mathbf{in}\ \mathtt{e1)\ e2}]\!]_\gamma \triangleq \sum_{k \in X} [\![\mathtt{e2}]\!]_{\gamma[\mathtt{<k,}a_k\mathtt{>/x}]} \qquad ([\![\mathtt{e1}]\!]_\gamma \triangleq \sum_{k \in X} a_k \bullet k)$$

Fig. 15: Denotational Semantics for types and terms of SDQL.

Proof The proof is by structural induction on `e1`. We write I.H. as short for induction hypothesis. We only show the non standard cases.

- Case of dictionary creation:
  $[\![\mathtt{\{ki\ ->\ vi\}}]\!]_\gamma\mathtt{[}[\![\mathtt{e}]\!]_\gamma\mathtt{/x]}$
  $= (\sum_i [\![\mathtt{ki}]\!]_\gamma \bullet [\![\mathtt{vi}]\!]_\gamma)\mathtt{[}[\![\mathtt{e}]\!]_\gamma\mathtt{/x]}$
  $= \sum_i [\![\mathtt{ki}]\!]_\gamma\mathtt{[}[\![\mathtt{e}]\!]_\gamma\mathtt{/x]} \bullet [\![\mathtt{vi}]\!]_\gamma[[\![\mathtt{e}]\!]_\gamma\mathtt{/x]}$
  $= \sum_i [\![\mathtt{ki[e/x]}]\!]_\gamma \bullet [\![\mathtt{vi[\ e/x]}]\!]_\gamma$ by I.H.
  $= [\![\mathtt{\{ki[e/x]\ ->\ vi[e/x]\}}]\!]_\gamma$
  $= [\![\mathtt{\{ki->\ vi\}[e/x]}]\!]_\gamma$
- Case of **sum** introduction:
  $[\![\mathbf{sum}\ \mathtt{(x}\ \mathbf{in}\ \mathtt{e1)e2}]\!]_\gamma\mathtt{[}[\![\mathtt{e}]\!]_\gamma\mathtt{/y]}$
  $= \sum_{x \in X} [\![\mathtt{e2}]\!]_{\gamma''}$ $(\gamma'' = \gamma[\text{<ak,k>/x}, [\![\mathtt{e}]\!]_\gamma/\text{y}], [\![\mathtt{e1}]\!]_\gamma = \sum_k \text{ak}\bullet \text{k})$
  $= \sum_{x \in X} [\![\mathtt{e2}]\!]_{\gamma'}[[\![\mathtt{e}]\!]_\gamma\mathtt{/y]}$ $(\gamma' = \gamma[\text{<ak,k>/x}], [\![\mathtt{e1}]\!]_\gamma = \sum_k \text{ak}\bullet \text{k})$
  $= \sum_{x \in X} [\![\mathtt{e2[e/y]}]\!]_{\gamma'}$ $(\gamma' = \gamma[\text{<ak,k>/x}], [\![\mathtt{e1[e/y]}]\!]_\gamma = \sum_k \text{ak}\bullet \text{k})$ by I.H.
  $= [\![\mathtt{(}\mathbf{sum}\ \mathtt{(x}\ \mathbf{in}\ \mathtt{e1[e/y])e2[e/y])}]\!]_\gamma$
  $= [\![\mathtt{(}\mathbf{sum}\ \mathtt{(x}\ \mathbf{in}\ \mathtt{e1)e2)[e/y]}]\!]_\gamma$

∎

Theorem 8.4 (Soundness)**.** *For all closed terms* $\vdash$ `e:` T *and* $\vdash$ `v:` T *where* `v` *is a value, if* `e` *reduces to* `v` *in the operational semantics, then* $[\![\mathtt{e}]\!] = [\![\mathtt{v}]\!]$.

Proof [Sketch] Most rules follow from the S-semi-module structure of types, or standard denotational semantics in sets and functions. The only non standard case is **sum**, but the result follows from associativity of addition, and 0 being the unit of addition. ∎

8.3 Correctness of Optimizations

The denotational semantics allows us to easily prove correctness of the optimizations of Figure 8. In particular, the formal $\sum$ notation in the semantics mechanically provides an efficient and sound calculus that is reminiscent of the algebra of polynomials. We make use of this calculus in the following proofs.

Proposition 8.5. *The vertical loop fusion rules of Figure 8 are sound.*

Proof We prove the first rule. The second rule is proved similarly.

$[\![\texttt{let y = sum(x in e1) \{f1(x.key)->x.val\} in sum(x in y)\{f2(x.key)->x.val\}}]\!]_\gamma$ $=$

$[\![\texttt{sum(x in y)\{f2(x.key)->x.val\}}]\!]_{\gamma'}$ $(\gamma' = \gamma[[\![\texttt{sum(x in e1) \{f1(x.key)->x.val\}}]\!]_\gamma / \texttt{y}])$ $=$

$[\![\texttt{sum(x in y)\{f2(x.key)->x.val\}}]\!]_{\gamma'}$ $(\gamma' = \gamma[\sum_{k\in X} a_k \bullet [\![\texttt{f1}]\!]_\gamma(k) / \texttt{y}],\ [\![\texttt{e1}]\!]_\gamma = \sum_{k\in X} a_k \bullet k)$ $=$

$\sum_{k\in X} a_k \bullet [\![\texttt{f2}]\!]_\gamma([\![\texttt{f1}]\!]_\gamma(k))$ $([\![\texttt{e1}]\!]_\gamma = \sum_{k\in X} a_k \bullet k)$ $=$ $\sum_{k\in X} a_k \bullet [\![\texttt{f2}\circ\texttt{f1}]\!]_\gamma(k))$ $([\![\texttt{e1}]\!]_\gamma = \sum_{k\in X} a_k \bullet k)$ $=$

$[\![\texttt{sum(x in e1)\{f2(f1(x.key))->x.val\}}]\!]_\gamma$ ■

Proposition 8.6. *The loop factorization rules of Figure 8 are sound.*

Proof We prove the first rule, and the second rule is proved similarly.

$[\![\texttt{sum(x in e1) e2 * f(x)}]\!]_\gamma$ $=$ $\sum_{k\in X} [\![\texttt{e2 * f(x)}]\!]_{\gamma'}$ $(\gamma' = \gamma[<k, a_k>/\ \texttt{x}],\ [\![\texttt{e1}]\!]_\gamma = \sum_{k\in X} a_k \bullet k)$ $=$

$\sum_{k\in X} [\![\texttt{e2}]\!]_\gamma * [\![\texttt{f}]\!]_\gamma <k, a_k>$ $([\![\texttt{e1}]\!]_\gamma = \sum_{k\in X} a_k \bullet k)$ $=$ (bilinearity)

$[\![\texttt{e2}]\!]_\gamma * \sum_{k\in X} [\![\texttt{f}]\!]_\gamma <k, a_k>$ $([\![\texttt{e1}]\!]_\gamma = \sum_{k\in X} a_k \bullet k)$ $=$

$[\![\texttt{e2}]\!]_\gamma * \sum_{k\in X} [\![\texttt{f(x)}]\!]_{\gamma'}$ $(\gamma' = \gamma[<k, a_k>/\ \texttt{x}],\ [\![\texttt{e1}]\!]_\gamma = \sum_{k\in X} a_k \bullet k)$ $=$

$[\![\texttt{e2}]\!]_\gamma * [\![\texttt{sum(x in e1) f(x)}]\!]_\gamma$ $=$ $[\![\texttt{e2 * sum(x in e1) f(x)}]\!]_\gamma$

■

Proposition 8.7. *The horizontal loop fusion rules of Figure 8 are sound.*

Proof [Proof] $[\![\texttt{let y1=sum(x in e1) f1(x) in let y2=sum(x in e1) f2(x) in f3(y1,y2)}]\!]_\gamma$

$=$ $[\![\texttt{let y2=sum(x in e1) f2(x) in f3(y1,y2)}]\!]_{\gamma'}$ $(\gamma' = \gamma[[\![\texttt{sum(x in e1) f1(x)}]\!]_\gamma / \texttt{y1}])$

$=$ $[\![\texttt{let y2=sum(x in e1) f2(x) in f3(y1,y2)}]\!]_{\gamma'}$ $(\gamma' = \gamma[\sum_{k\in X} [\![\texttt{f1}]\!]_\gamma <k, a_k>/\ \texttt{y1}],\ [\![\texttt{e1}]\!]_\gamma = \sum_{k\in X} a_k \bullet k)$

$=$ $[\![\texttt{f3(y1,y2)}]\!]_{\gamma'}$ $(\gamma' = \gamma[\sum_{k\in X} [\![\texttt{f1}]\!]_\gamma <k, a_k>/\ \texttt{y1}, [\![\texttt{sum(x in e1) f2(x)}]\!]_\gamma\ /\texttt{y2}],\ [\![\texttt{e1}]\!]_\gamma = \sum_{k\in X} a_k \bullet k)$

$=$ $[\![\texttt{f3(y1,y2)}]\!]_{\gamma'}$ $(\gamma' = \gamma[\sum_{k\in X} [\![\texttt{f1}]\!]_\gamma <k, a_k>/\ \texttt{y1}, \sum_{k\in X} [\![\texttt{f2}]\!]_\gamma <k, a_k>\ /\texttt{y2}],\ [\![\texttt{e1}]\!]_\gamma = \sum_{k\in X} a_k \bullet k)$

$= [\![\texttt{f3}]\!]_\gamma \big(\sum_{k\in X} [\![\texttt{f1}]\!]_\gamma\texttt{<}k, a_k\texttt{>},\ \sum_{k\in X} [\![\texttt{f2}]\!]_\gamma\texttt{<}k, a_k\texttt{>}\big) \quad ([\![\texttt{e1}]\!]_\gamma = \sum_{k\in X} a_k \bullet k)$
$= [\![\texttt{f3(tmp.y1, tmp.y2)}]\!]_{\gamma'} \quad (\gamma' = \gamma[\texttt{<} \sum_{k\in X} [\![\texttt{f1}]\!]_\gamma\texttt{<}k, a_k\texttt{>}, [\![\texttt{f2}]\!]_\gamma\texttt{<}k, a_k\texttt{>}\ \texttt{>}\ /\texttt{tmp}],$
$[\![\texttt{e1}]\!]_\gamma = \sum_{k\in X} a_k \bullet k)$
$= [\![\texttt{let tmp = sum(x in e1) <y1=f1(x), y2=f2(x)> in f3(tmp.y1, tmp.y2)}]\!]_\gamma$ ■

Proposition 8.8. *The rewrite rule for loop-invariant code motion in Figure 8 is sound.*

Proof $[\![\texttt{sum(x in e1) let y = e2 in f(x, y)}]\!]_\gamma$
$= \sum_{k\in X} [\![\texttt{let y = e2 in f(x, y)}]\!]_{\gamma'} \quad (\gamma'=\gamma[\texttt{<}k, a_k\texttt{>}/\ \texttt{x}],\ [\![\texttt{e1}]\!]_\gamma = \sum_{k\in X} a_k \bullet k)$
$= \sum_{k\in X} [\![\texttt{f(x, y)}]\!]_{\gamma'} \quad (\gamma'=\gamma[\texttt{<}k, a_k\texttt{>}/\ \texttt{x},\ [\![\texttt{e2}]\!]_\gamma\ /\ \texttt{y}],\ [\![\texttt{e1}]\!]_\gamma = \sum_{k\in X} a_k \bullet k)$
$= [\![\texttt{sum(x in e1) f(x, y)}]\!]_{\gamma'} \quad (\gamma'=\gamma[[\![\texttt{e2}]\!]_\gamma\ /\ \texttt{y}])$
$= [\![\texttt{let y = e2 in sum(x in e1) f(x, y)}]\!]_\gamma$ ■

Proposition 8.9. *The rewrite rules for loop memoization in Figure 8 are sound.*

Proof We prove the second rule, and the first rule is proved similarly.
$[\![\texttt{sum(x in e1) if(p(x) == e2) then f(x)}]\!]_\gamma$
$= \sum_{k\in X} [\![\texttt{if(p(x) == e2) then f(x)}]\!]_{\gamma'} \quad (\gamma'=\gamma[\texttt{<}k, a_k\texttt{>}/\ \texttt{x}],\ [\![\texttt{e1}]\!]_\gamma = \sum_{k\in X} a_k \bullet k) \quad (\texttt{x} \notin \text{FVs}$
of $\texttt{f, p, e2})$
$= \sum_{k\in X} [\![\texttt{f}]\!]_\gamma(a_k, k) * ([\![\texttt{p}]\!]_\gamma(a_k, k) == [\![\texttt{e2}\ ]\!]_\gamma) \quad (\gamma'=\gamma[\texttt{<}k, a_k\texttt{>}/\ \texttt{x}],\ [\![\texttt{e1}]\!]_\gamma = \sum_{k\in X} a_k \bullet k)$
$= \sum_{k\in X} [\![\texttt{f}]\!]_\gamma(a_k, k) \quad (\gamma'=\gamma[\texttt{<}k, a_k\texttt{>}/\ \texttt{x}],\ [\![\texttt{e1}]\!]_\gamma = \sum_{k\in X} a_k \bullet k,\ [\![\texttt{p}]\!]_\gamma(a_k, k) == [\![\texttt{e2}\ ]\!]_\gamma)$
$= \pi_{[\![\texttt{e2}\ ]\!]_\gamma} \big(\sum_{k\in X} [\![\texttt{f}]\!]_\gamma(a_k, k) \bullet [\![\texttt{p}\ ]\!]_\gamma(a_k, k)\big) \quad (\gamma'=\gamma[\texttt{<}k, a_k\texttt{>}/\ \texttt{x}],\ [\![\texttt{e1}]\!]_\gamma = \sum_{k\in X} a_k \bullet k)$
$= \pi_{[\![\texttt{e2}\ ]\!]_\gamma} \big([\![\texttt{sum(x in e1) \{p(x)->f(x)\}}\ ]\!]_\gamma \big)$
$= [\![\texttt{let tmp=sum(x in e1) \{p(x)->f(x)\} in tmp(e2)}]\!]_\gamma$ ■

8.4 Operational Semantics

We now give a standard call-by-value small-step operational semantics to SDQL. The syntax for evaluation context and values as well as reduction rules are shown in Figure 16. All our types form a semi-ring with zero denoted by $\mathbf{0}_\texttt{T}$. $\mathbf{0}_\texttt{T}$ is a macro, defined by induction on `T` as follows. $\mathbf{0}_\texttt{S}$ is the constant 0 of the scalar type `S`. $\mathbf{0}_{\texttt{< a:T, ... >}} = \texttt{< a:}\mathbf{0}_\texttt{T}\texttt{, ... >}$. $\mathbf{0}_{\texttt{T1 -> T2}} = \texttt{\{ \}}_{\texttt{T1,T2}}$. For construction of records and dictionaries with multiple arguments, the evaluation order is from left to right. Next, we introduce some lemmas.

Lemma 8.10 (Confluence)**.** *Let* $\Gamma \vdash$ `e`: `T`*. If* `e` $\to$`e1` *and* `e` $\to$`e2`*, there exists* `e'` *such that* `e1` $\to^*$ `e'` *and* `e2`$\to^*$ `e'`*.*

Proof [Sketch] By inspection, the only non deterministic cases are dictionary addition and `sum` (that requires ranging over a dictionary). Formally, the denotational semantics of our dictionaries are unordered (they are sets). This allows + on semi-ring dictionaries to be commutative. ■

Evaluation contexts
E ::= `sum(x in` E`) e` | E`(e)` | `v(`E`)` | `let x =` E `in e` | `if(`E`) then e else e`
| `{ v -> v, ...,` E `-> e, ... }` | `{ v -> v, ..., v ->` E`, ... }`
| `< a_1 = v, ..., a_i =` E`, ... >` | E`.a` | E `* e` | `v *` E | E `+ e` | `v +` E
| $\texttt{promote}_{\texttt{S,S}}$`(`$E$`)` | []
Values
`v ::= { v -> v, ... }` | `< a = v, ... >` | `n` | `r` | `false` | `true` | $\mathbf{0}_{\mathtt{T}}$

$$\frac{}{\texttt{sum(x in\{k0->v0,...\})e2} \longrightarrow \texttt{e2[<key=k0,val=v0>/x]+sum(x<-\{k1->v1,...\})e2}}$$

$$\frac{\texttt{v1,v2:S}}{\texttt{(v1)+(v2)} \longrightarrow \texttt{(v1+v2)}} \qquad \frac{\texttt{v1,v2:S}}{\texttt{(v1)*(v2)} \longrightarrow \texttt{(v1*v2)}} \qquad \frac{}{\texttt{promote}_{\texttt{S1,S2}}\texttt{(v)} \longrightarrow \texttt{v}}$$

$$\frac{\texttt{e2: T}}{\texttt{sum(x in\{\})e2} \longrightarrow \mathbf{0}_{\mathtt{T}}} \qquad \frac{}{\texttt{let x=v in e2} \longrightarrow \texttt{e2[v/x]}} \qquad \frac{}{\texttt{<a\_0=e\_0,...>.a\_i} \longrightarrow \texttt{e\_i}}$$

$$\frac{}{\texttt{\{k\_0->v0\_0, ...\} + \{k\_0->v1\_0, ...\}} \longrightarrow \texttt{\{k\_0->v0\_0+v1\_0, ...\}}}$$

$$\frac{}{\texttt{<a\_0=v0\_0, ...> + <a\_0=v1\_0, ...>} \longrightarrow \texttt{<a\_0=v0\_0+v1\_0, ...>}}$$

$$\frac{\texttt{v1 : T3}}{\texttt{\{ \}}_{\texttt{T1,T2}}\texttt{ * v1} \longrightarrow \texttt{\{ \}}_{\texttt{T1,T2}\otimes\texttt{T3}}} \qquad \frac{}{\texttt{\{k\_0->v\_0, ...\} * v1} \longrightarrow \texttt{\{k\_0->(v\_0*v1), ...\}}}$$

$$\frac{}{\texttt{<a\_0=v\_0, ...> * v1} \longrightarrow \texttt{<a\_0=(v\_0*v1), ...>}}$$

$$\frac{\texttt{v1: S}}{\texttt{v1 * \{ \}}_{\texttt{T1,T2}} \longrightarrow \texttt{\{ \}}_{\texttt{T1,T2}}} \qquad \frac{\texttt{v1: S}}{\texttt{v1 * \{k\_0->v\_0, ...\}} \longrightarrow \texttt{\{k\_0->(v1*v\_0), ...\}}}$$

$$\frac{\texttt{v1: S}}{\texttt{v1 * <a\_0=v\_0, ...>} \longrightarrow \texttt{<a\_0=(v1*v\_0), ...>}}$$

$$\frac{\exists \texttt{j. k\_j = k1}}{\texttt{\{ k\_0 -> v\_0, ... \}(k1)} \longrightarrow \texttt{v\_j}} \qquad \frac{\nexists \texttt{j. k\_j = k1} \quad \forall \texttt{i. v\_i: T}}{\texttt{\{ k\_0 -> v\_0, ... \}(k1)} \longrightarrow \mathbf{0}_{\mathtt{T}}}$$

$$\frac{}{\texttt{if(true) then e1 else e2} \longrightarrow \texttt{e1}} \qquad \frac{}{\texttt{if(false) then e1 else e2} \longrightarrow \texttt{e2}}$$

Fig. 16: Reduction rules for SDQL.

Lemma 8.11 (Type Preservation). *If* $\Gamma \vdash$ `e: T` *and* `e` $\longrightarrow$ `e'` *then* $\Gamma \vdash$ `e': T`*.*

Proof [Sketch] By induction on the structure of `e` and case analysis on each reduction rule. ■

Lemma 8.12 (Fundamental lemma). *For every* `x1:T1,...,xn:Tn` $\vdash$ `e: T` *and every value* `v1:T1,...,vn:Tn`*,* `e[v1/x1,...,vn/xn]` *reduces to a value.*

Proof [Sketch] By induction on the structure of `e`, then case analysis on each typing rule. As usual, the quantification is for all `n` and not for fixed `n`. ■

Theorem 8.13. *Every closed and well-typed term* `e` *reduces to a unique value.*

Proof [Sketch] By choosing $\Gamma = \emptyset$ in Lemma 8.12. ∎

8.5 *Correctness of Optimizations using Operational Semantics*

We prove correct the optimizations of Figure 8. As is usual, we denote by $\rightarrow^*$ the transitive reflexive closure of $\rightarrow$. We say a rule `e` $\rightsquigarrow$ `e'` is sound (w.r.t. the evaluation semantics) if `e` and `e'` have the same operational semantics, i.e. `e` $\rightarrow^*$ `v` iff `e'` $\rightarrow^*$ `v`.

Proposition 8.14 (Correctness of Vertical Loop Fusion)**.** *The vertical loop fusion rules of Figure 8 are sound.*

Proof [Sketch] The correctness of the first rule can be proved by performing induction on the value of the dictionary `d={`$\mathtt{k}_1$`->`$\mathtt{v}_1$`, ..., `$\mathtt{k}_{n+1}$`->`$\mathtt{v}_{n+1}$`}` where `e1` $\rightarrow^*$ `d`. The correctness of the base case `d={}` is obvious. For the induction step, one has to consider different cases based on whether `f1(`$\mathtt{k}_{n+1}$`)` is equivalent to `f1(`$\mathtt{k}_i$`)` for $i \leq n$. If this is the case the proof is straightforward. If this is not the case, there will be two further cases. Assuming `f1(`$\mathtt{k}_{n+1}$`)`$\rightarrow^*$`k'`$_{p+1}$, either `f2(k'`$_{p+1}$`)` is equivalent to `f2(k'`$_j$`)` for some $j \leq p$.In each case, both LHS and RHS are evaluated to the same value.

The correctness of the second rule can be proved by simply computing the result of the evaluation of both the LHS and RHS for an arbitrary dictionary value for `e1`. ∎

Proposition 8.15 (Correctness of Horizontal Loop Fusion)**.** *The horizontal loop fusion rules of Figure 8 are sound.*

Proof [Sketch] Straightforward by induction on the value of dictionary `d` which is the result of evaluating `e1`. ∎

Proposition 8.16 (Correctness of Loop Factorization)**.** *The loop factorization rules of Figure 8 are sound.*

Proof [Sketch] By induction on the values of the dictionary `d` which is the result of evaluating `e1`. For the inductive step, we use the distributive law of the semi-ring structure. ∎

Proposition 8.17 (Correctness of Loop-Invariant Code Motion)**.** *The rewrite rule for loop-invariant code motion in Figure 8 is sound.*

Proof [Sketch] `e1` reduces to a value `{ k1 -> v1, ..., kn -> vn }`. The LHS reduces to $\sum_i$ `(let y = e2 in 1)*f(x, y)[ki,vi/x]`, where $\sum_i g_i$ is a shorthand for $g_1 + \ldots + g_n$. Assuming `e2` reduces to a value `v`, the first element of the summation reduces to `f(x, y)[k1,v1/x, v/y]`. This term then reduces to a value `f1`. Similarly, for each `i`, `f(x, y)[ki,vi/x, v/y]` reduces to `fi`. Hence, the LHS eventually reduces to $\sum_i$ `fi`. In the RHS, `e2` reduces first to the value `v`. Then the RHS reduces to `sum(x in e1) f(x, y)[v/y]`. We then conclude as before. `e1` reduces to a value `{ k1 -> v1, ..., kn -> vn }` and the RHS reduces to $\sum_i$ `fi`. In summary, what makes this optimization correct is that substituting `x` then `y` is the same as substituting `y` then `x`. ∎

Proposition 8.18 (Correctness of Loop Memoization). *The rewrite rule for loop memoization in Figure 8 is sound.*

Proof [Sketch] By induction on the dictionary `d` which is the result of evaluating `e1`. ∎

9 Implementation

SDQL is implemented as an external domain-specific language. The entire compiler toolchain is written in Scala. The order of rewrite rules are applied as follows until a fix-point is reached: 1) loop fusion, 2) loop-invariant code motion, 3) loop factorization, and 4) loop memoization. After each optimization, generic optimization such as DCE, CSE, and partial evaluation are also applied. Note that we currently expect the loop order to be specified correctly by the user. Finally, the optimized program is translated into C++.

9.1 C++ Code Generation

The code generation for SDQL is mostly straightforward, thanks to the first-order nature of most of its constructs. Thus, we do not face the technical challenges of compiling polymorphic higher-order functional languages (e.g., all objects are stack-allocated. Hence, GC is unnecessary). The key challenging construct is **`sum`** which is translated into `for`-loops. For the case of summations that produce dictionaries, the generated loop performs destructive updates to improve the performance (Henriksen *et al.*, 2017; Shaikhha, 2022).

9.2 C++ Runtime

The C++ runtime employs an efficient hash table implementation based on closed hashing for dictionaries.[3] For dictionaries with **`dense_int`** keys, the runtime either uses `std::array` or `std::vector` depending on whether the size is statically known during compilation time. In certain cases, input dictionaries are stored in ordered dictionaries based on arrays.[4] Finally, for implementing records, SDQL uses `std::tuple`.

9.3 Semi-Ring Extensions

Scalar Semi-Rings. Throughout the article, we only focused on three important scalar semi-rings, and the corresponding record and dictionary semi-rings. FAQ (Abo Khamis *et al.*, 2016) introduced several semi-ring structures with applications on graphical models, coding theory, and logic. Also, semi-rings were used for language recognition, reachability, and shortest path problems (Dolan, 2013; Shaikhha & Parreaux, 2019). SDQL can support such applications by including additional scalar semi-rings, a subset of which are presented

[3] https://github.com/greg7mdp/parallel-hashmap
[4] https://beta.boost.org/doc/libs/1_68_0/doc/html/boost/container/flat_map.html

SDQL[ring]		
`-(-e)`	$\rightsquigarrow$	`e`
`e + (-e)`	$\rightsquigarrow$	`0`

SDQL[closure]		
`1 + e * closure(e)`	$\rightsquigarrow$	`closure(e)`
`1 + closure(e) * e`	$\rightsquigarrow$	`closure(e)`

SDQL[prod]		
`(prod(x in e1) f1(x)) *` `(prod(x in e1) f2(x))`	$\rightsquigarrow$	`prod(x in e1)` `f1(x) * f2(x)`

SDQL[rec]		
`rec(x => let y=e1 in f(x,y))(e2)`	$\rightsquigarrow$	`let y=e1 in rec(x => f(x,y))(e2)`

Fig. 17: Additional transformation rules for language extensions of SDQL.

in Table 1. The `promote` construct can be used to annotate numeric values with the type of the appropriate types in such cases.

Non-scalar Semi-Rings. The support for semi-ring extensions in SDQL is beyond scalar types. As an example, SDQL supports the (semi-)ring of the covariance matrix (Nikolic & Olteanu, 2018). For each $n \in \mathbb{Z}$, the domain $\mathbb{D}$ of this semi-ring is a triple $< \mathbb{R}, \mathbb{R}^n, \mathbb{R}^{n \times n} >$. The additive and multiplicative identities are defined as $0^{\mathbb{D}} \triangleq < 0, 0^n, 0^{n \times n} >$ and $1^{\mathbb{D}} \triangleq < 1, 0^n, 0^{n \times n} >$. For each $a \triangleq < s_a, v_a, m_a >$ and $b \triangleq < s_b, v_b, m_b >$, the addition and multiplication are defined as:

$$
\begin{aligned}
a +^{\mathbb{D}} b &\triangleq < s_a + s_b, v_a + v_b, m_a + m_b > \\
a \times^{\mathbb{D}} b &\triangleq < s_a * s_b, s_a * v_b + v_a * s_b, s_b * m_a + s_a * m_b + v_a * v_b + v_b * v_a >
\end{aligned}
$$

We use this semi-ring to compute covariance matrix as aggregates over relations (cf. Section 10.4).

9.4 Language Extensions

In this section, we define possible language extensions over SDQL. Apart from an additional expressive power, each extension enables further optimizations, which are demonstrated in Figure 17. We use SDQL[`X`] to denote SDQL extended with `X`.

SDQL[`ring`]: SDQL + Ring Dictionaries. We have consistently talked about semi-ring structures, and how semi-ring dictionaries can be formed using value elements with such structures. There is another important structure, referred to as *ring*, for the cases that the addition operator admits an inverse. The transformation rules enabled by the ring structure are shown in Figure 17. As it can be observed in Table 1, real and integer sum-products form ring structures. Similarly to semi-ring dictionaries, one can obtain ring dictionaries by using values that form a ring. In this case, the additive inverse of a particular ring dictionary is a ring dictionary with the same keys but with inverse value elements.

SDQL[`closure`]: SDQL + Closed Semi-Rings. Orthogonally, one can extend the semi-ring structure with a closure operator (Dolan, 2013). In this way, transitive closure algorithms can also be expressed by generalizing semi-rings to closed semi-rings (Lehmann, 1977). In many cases, the semi-ring structures involve an additional idempotence axiom (`a + a = a`) resulting in dioids. The closure operator for dioids is called a Kleene star and the extended structure is referred to as Kleena algebra, which is useful for expressing path problems in graphs among other use-cases (Gondran & Minoux, 2008). This structure can be reflected in our kind-system; the product of dioids/Kleene algebras forms a dioid/Kleene algebra. In future work, we would like to investigate how to express the standard algorithm that computes `closure`(A) for a matrix A over a Kleene algebra in terms of a program involving semi-ring dictionaries over a Kleene algebra.

SDQL[`prod`]: SDQL + Product. We have only considered the summation over semi-ring dictionaries. One can use `prod` instead of `sum`. This would allow to elegantly express universal quantification over the possible assignments of that variable, like in FAQ (Abo Khamis *et al.*, 2016) to express quantified Boolean queries. As an example, checking if the predicate `p` is satisfied by all elements of relation `R` is phrased as: `prod(r <- R) p(r)`. The commutative monoid structure of multiplication allows for optimizations with a similar impact as horizontal loop fusion (cf. Figure 17).

SDQL[`rec`]: SDQL + Recursion. Apart from supporting the closure and product constructs, it is possible to support more general forms of recursion. As shown for matrix query languages (Geerts *et al.*, 2021), an additional for-loop-style construct can express summation, product, transitive closure, as well as matrix inversion. This general form of recursion also allows for iterations, similarly to the `while` construct in IFAQ (Shaikhha *et al.*, 2020) that enables iterative computations required for optimization producures such as batch gradient decent (BGD). The additional expressive power of this construct comes with limited optimization opportunies; loop fusion and factorization are no longer applicable to them, however, code motion can still be leveraged (cf. Figure 17).

10 Experimental Results

10.1 Experimental Setup

For the first experiment in Section 10.2, we use a MacBook Pro running macOS 14.5 with Apple M1 Max CPU and 64 GB of LPDDR5 RAM. For the rest of the experiments in Section 10.2, we use a server running Ubuntu 22.04 LTS equipped with a 2.2 GHz Intel Xeon Silver 4210 CPU and 220 GB of main memory. We run our Section 10.4 experiments on an iMac equipped with an Intel Core i5 CPU running at 2.7GHz, 32GB of DDR3 RAM with OS X 10.13.6. We use CLang 10. on the first two machines and CLang 1000.10.44.4 on the last machine for compiling the generated C++ code on the last machine using the *-O3 -march=native -mtune=native -ftree-vectorize* flags. Our competitor systems use Scala 2.12.2, Spark 3.0.1, DuckDB 1.0.0, Python 3.7.4 (Python 2.7.12 for MorpheusPy), NumPy 1.16.2, and SciPy 1.2.1. All experiments are run on one CPU core.[5] We measure the average run time execution of five runs excluding the loading time.

[5] SDQL.py Shahrokhi & Shaikhha (2023*a*); Shahrokhi *et al.* (2023) has extended SDQL with parallelism for query processing and SDQLite Schleich *et al.* (2023); Shaikhha *et al.* (2024) with advanced sparse storage formats for tensor processing. However, they are beyond the scope of this article.

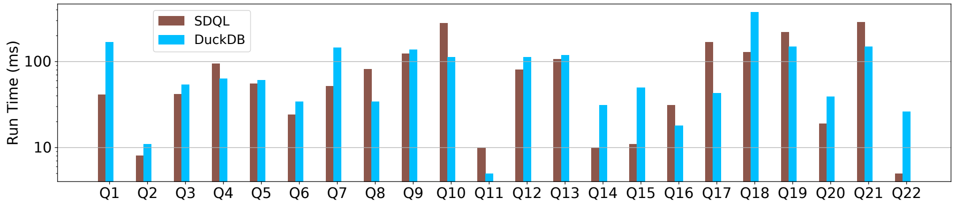


Fig. 18: Run time results for TPCH queries comparing SDQL and DuckDB.

10.2 Database Workloads

In this section, we investigate the performance of SDQL for online analytical processing (OLAP) workloads used in the DBs. For this purpose, we consider two types of benchmarks.

First, we use TPCH, a decision support system benchmark used for traditional analytical database management systems. We initially consider all 22 TPCH queries and compare the performance of SDQL with DuckDB (Raasveldt & Mühleisen, 2019), a state-of-the-art analytical query processing engine. Then, for a representative subset of TPCH queries, we compare the performance of generated optimized code for the dictionary layout, row layout, and columnar layout of SDQL.

Second, we use the IMDB dataset for the worst-case optimal join experiments. We benchmark the Generic Join implementation in SDQL and consider the Generic Join implementation from (Wang *et al.*, 2023) and the DuckDB implementation as competitors.

Figure 18 shows the performance of all TPCH queries. The SDQL compiler uses the columnar layout for all inputs in all queries. Overall, we observe that SDQL is in most cases faster than DuckDB. This is despite the fact that DuckDB specializes in handling in-memory database analytics tasks. However, SDQL can handle other types of workloads, as we will see in the next section.

Figure 19 shows that the row layout for input relations leads to a 4.2× speedup over the standard dictionary layout. The columnar layout further improves the performance by 1.5×. This is due to improved cache locality, as unused columns are not read into cache in case of the columnar layout.

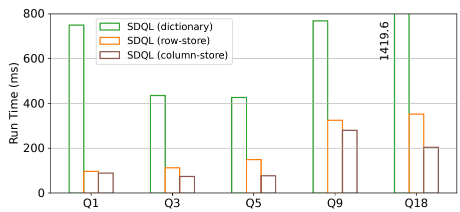


Fig. 19: Run time results for TPCH queries comparing different data layouts in SDQL.

Figure 20 shows that the performance for WCOJ. Similarly to FreeJoin (Wang *et al.*, 2023), we start with the physical plans produced by DuckDB and modify them to use WCOJ operators instead of binary joins. By computing the geometric mean, the SDQL's implementation of Generic Join is 1.85× faster than the implementation of the similar implementation in FreeJoin (Wang *et al.*, 2023). Also, we observe a 0.77× speedup compared to the traditional joins implemented by DuckDB. This behaviour is similar to the observations made by FreeJoin (Wang *et al.*, 2023).

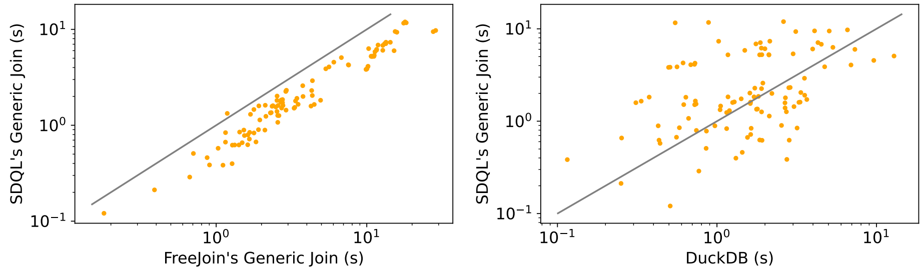


Fig. 20: Run time results for IMDB dataset comparing the worst-case optimal join algorithms in SDQL, GJ from FreeJoin (Wang *et al.*, 2023), and traditional joins in DuckDB (Raasveldt & Mühleisen, 2019).

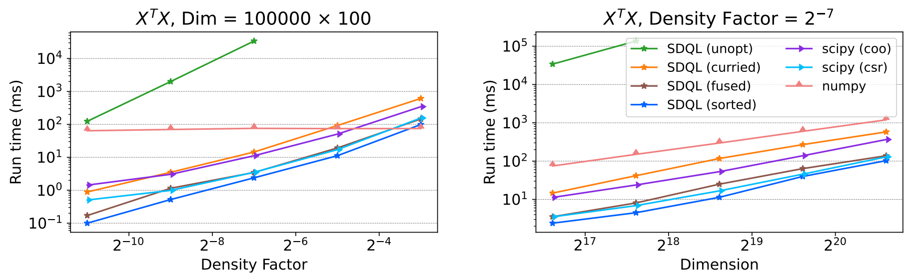


Fig. 21: Run time results for computing the covariance matrix comparing different optimizations and representations in SDQL, SciPy, and NumPy. The dimension for the input matrix of the left figure is 100000×100, and the dimension of the input matrix of the right figure is $N \times 100$ with the density of 2^{-7}.

10.3 Linear Algebra Workloads

In this section, we investigate the performance of SDQL for linear algebra workloads. We consider both matrix and higher-order tensor workloads. For the matrix processing workload, we use NumPy and SciPy as competitors, which use dense and sparse representations for matrices. This workload involves matrix transpose, which is not supported by systems such as taco (Kjolstad *et al.*, 2017). For the tensor processing workloads, we use taco (Kjolstad *et al.*, 2017) as the only competitor, both their artifacts and the latest version available. SciPy does not support higher-order tensors, and it was shown before (Kjolstad *et al.*, 2017; Chou *et al.*, 2018) that on these workloads, taco is faster than systems such as SPLATT (Smith *et al.*, 2015), Tensor Toolbox (Bader & Kolda, 2008), and TensorFlow (Abadi *et al.*, 2016). For a fair comparison, we have included the time for assembling the output tensor in taco.

Sparse Matrix Processing. First, we consider the task of computing the covariance matrix $X^T X$ (cf. Section 4), where X is a synthetically generated input data matrix of varying dimensions and density. We consider the following different versions of the generated code from SDQL: 1) unoptimized, which is the uncurried representation of matrices, 2) curried, which uses the curried representation 3) fused, which additionally fuses the transpose and multiplication operators, and 4) sorted, which additionally uses sorted dictionaries for the sparse inputs.

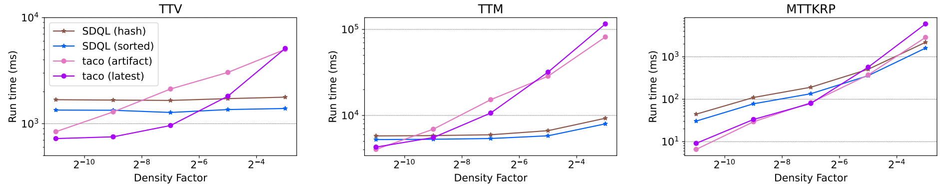


Fig. 22: Run time results of SDQL and taco for TTV, TTM, and MTTKRP on Nell-2 dataset by varying the sparsity of the second and third operands. Both systems use a sparse representation for all tensor modes.

As Figure 21 shows, using curried representation can provide asymptotic improvements over the naïve representation, thanks to the improved matrix multiplication operator (cf. Section 6.1). Furthermore, performing fusion can provide 4.4× speedup on average. On top of that, using both sorted and unsorted dictionaries for sparse data can provide 1.7× speedup on average.

The usage of dense representation (by NumPy) can provide better implementations as the matrix becomes more dense; however, for smaller densities, sparse representations (by SciPy and SDQL) can be up to 640× faster. Finally, the most optimized version of the generated code by SDQL is on average 5.5× and 1.8× faster than the COO and CSR representations of SciPy, respectively, thanks to fusion and the efficient low-level code generated by SDQL.

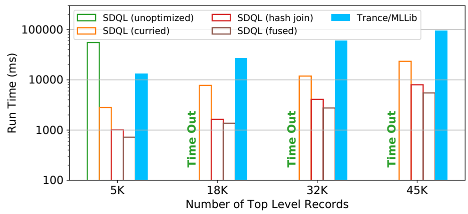
Sparse Tensor Processing. Next, we consider three higher-order tensor workloads on NELL-2, a real world dataset coming from the Never Ending Language-Learning project (Carlson *et al.*, 2010). Figure 22 shows the performance comparison for these workloads. We observe that especially for a medium range of sparsity SDQL is faster than taco. For sparser scenarios, taco shows better performance, thanks to the DCSR format and its merge-based multiplications. A similar observation on hash/CSR formats has been made in (Chou *et al.*, 2018). Also, SDQL achieves up to 1.5× speedup by supporting sorted dictionaries.

	TTV	**TTM**	**MTTKRP**
LA Formulation	$A_{ij} = \Sigma_k B_{ijk} c_k$	$A_{ijk} = \Sigma_k B_{ijl} C_{kl}$	$A_{ij} = \Sigma_{k,l} B_{ikl} C_{kj} D_{lj}$
Einsum Notation	`ijk,k->ij`	`ijl,kl->ijk`	`ijk,kj,lj->ij`

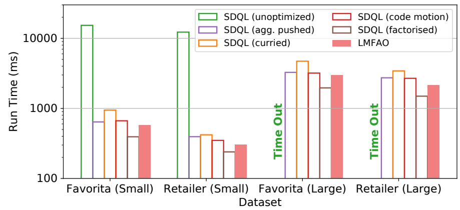
10.4 Hybrid LA/DB Workload

As the final set of experiments, we consider hybrid workloads that involve linear algebra and query processing. Figure 23 shows the experimental results for computing the covariance matrix. We consider experiments that use 1) nested, 2) relational, and 3) normalized matrix input datasets.

Nested Data. For nested data, we use our motivating biomedical example as the workload and variant data from 1000 genomes dataset as input (Sudmant *et al.*, 2015). The experiment involves computing the covariance matrix of the join of `Genes` and `Variants` relations, by increasing the number of the elements of the former relation; this is synonymous to increasing the number of features in the covariant matrix by approximately 15, 30, 55, and 70. We consider the following four versions of the generated code from SDQL:



(a) Biomedical query with different optimizations in SDQL and Trance/MLLib.



(b) Retail forecasting using different optimizations in SDQL and LMFAO.

Fig. 23: Run time results for computing covariance matrix over nested and relational data.

1) unoptimized code that uses uncurried representation for matrices, 2) curried version that uses curried representation for intermediate matrices, 3) a version that uses hash join for joining `Genes` and `Variants`, and 4) a version obtained by fusing intermediate dictionaries resulting from grouping and matrix transpose. As our competitor, we only consider Trance Smith *et al.* (2020) for the query processing part, which implements an extension of NRC^{+} with aggregation called NRC^{agg} and uses Spark MLLib Meng *et al.* (2016) for the linear algebra processing. This is because in-database machine learning frameworks such as IFAQ Shaikhha *et al.* (2020), LMFAO Schleich *et al.* (2019), and Morpheus Chen *et al.* (2017); Li *et al.* (2019) do not support nested relations.

As Figure 23a shows, we observe that using curried representation gives asymptotic improvements, and allows SDQL to scale to larger inputs. Furthermore, using hash join, gives an additional 3× speedup. This speedup can be larger for larger `Genes` relations. Performing fusion results in an additional 50% speedup thanks to the removal of intermediate dictionaries and less loop traversals. Finally, we observe around one order of magnitude performance improvement over Trance/MLLib thanks to the lack of need for unnesting, which is enabled by nested dictionaries provided by SDQL.

Relational Data. Next, we compute the covariance matrix over the result of join of relational input. To do so, we use the semi-ring of the covariance matrix (cf. Section 9.3). We use two real-world relational datasets: 1) *Favorita* Favorita (2017), a publicly available Kaggle dataset, and 2) *Retailer*, a US retailer dataset Schleich *et al.* (2016). Both datasets are used in retail forecasting scenarios and consist of 6 and 5 relations, respectively. We only use five continuous attributes of these datasets. We consider the following five versions of the generated code, where optimizations are applied accumulatively: 1) unoptimized code that involves materializing the result of join before computing the aggregates, 2) a version where all the aggregates are push down before the join computation, 3) a curried version that uses a trie representation for input relations and intermediate results, 4) a version that applies loop-invariant code motion, and 5) the most optimized version that performs loop factorization after all the previous optimizations. As our competitor, we use LMFAO (Schleich *et al.*, 2019), an in-DB ML framework that was shown to be up to two orders of magnitude faster than Tensorflow (Abadi *et al.*, 2016) and MADLib (Hellerstein *et al.*, 2012) for these two datasets.

Figure 23b shows that first, pushing aggregates before join results in around one order of magnitude performance improvement, thanks to the removal of the intermediate large join. Second, using a curried representation degrades the performance, due to the fact that iterations over hash tables is more costly. Third, code motion can leverage the trie-based

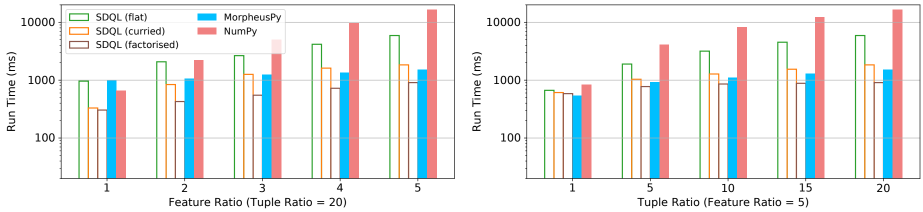


Fig. 24: Run time of SDQL, MorpheusPy, and NumPy for computing the covariance matrix over normalized matrix. For both plots, S has two features ($d_S = 2$) and R contains one million tuples ($n_R = 1M$). In the left figure, $n_S = 20M$ and $d_R \in \{2, 4, 6, 8, 10\}$. In the right figure, $d_R = 10$ and $n_S \in \{1M, 5M, 10M, 15M, 20M\}$.

iteration, and hoist invariant computations outside the loop to bring 30% speed up in comparison with the curried version. Finally, loop factorization leverages the distributivity rule for the semi-ring of covariance matrix, and factorizes the costly multiplications outside the inner loops. On average, this optimization brings 60% speed up in comparison with the previous version, and 40% speed up over LMFAO.

Normalized Matrix Data. Finally, we compute the covariance matrix over the join of relations represented as normalized matrices. We use the same semi-ring as the one for relational data. As the competitor, we consider NumPy and MorpheusPy (Side Li, 2019*a*), a Python-based implementation of Morpheus (Chen *et al.*, 2017). The publicly available version of Morpheus only supports one primary-key foreign-key join of two relations (Side Li, 2019*b*), i.e., $R \bowtie S$. Figure 24 shows the performance of Morpheus and SDQL for computing the covariance matrix over such a join. As in the original Morpheus paper (Chen *et al.*, 2017), the join computation time for NumPy is not included. Also, the values for the primary key is the dense integer values between one and one million; thus all competitors use a dense representation for them. The number of tuples for R is one million ($n_R = 1M$), and for S varies between millions ($n_S \in \{1M, 5M, 10M, 15M, 20M\}$). The number of the features for S is two ($d_S = 2$), and for R varies between two and ten ($d_R \in \{2, 4, 6, 8, 10\}$).

Figure 24 shows that the NumPy-based implementation over the materialized join can have a better performance for relations with the same number of features. The factorized computation starts showing its benefits for larger feature ratios. MorpheusPy is always better than the flat representation of SDQL. This is thanks to vectorization, which shows its impacts further as the feature ratio increases. Finally, we observe a superior performance for SDQL once the curried representation and loop factorization are used. As the tuple ratio increases, the speed up of SDQL over MorpheusPy climbs up to 1.7×; this is because of loop factorization enabled by the curried representation for relation S. MorpheusPy expresses aggregations and joins in terms of linear algebra operations using NumPy, which do not benefit from such optimizations.

11 Related Work

In this section, we review the literature. Table 2 summarizes the differences between different data analytics approaches and SDQL.

Relational Query Engines. Just-in-time compilation of queries has been heavily investigated in the DB community (Krikellas *et al.*, 2010; Neumann, 2011; Koch *et al.*, 2014;

Shaikhha *et al.*, 2018*a*, 2016; Viglas *et al.*, 2014; Crotty *et al.*, 2015; Nagel *et al.*, 2014; Karpathiotakis *et al.*, 2015; Armbrust *et al.*, 2015; Palkar *et al.*, 2017; Tahboub *et al.*, 2018). As an alternative, vectorized query engines process blocks of data to remove interpretation overhead (Zukowski *et al.*, 2005). None of these efforts have focused on handling hybrid DB/LA workloads as opposed to SDQL. We leave extending SDQL with vectorization features (Shahrokhi, 2024; Shahrokhi & Shaikhha, 2023*b*) for future.

Nested Data Models. Nested relational model (Roth *et al.*, 1988) and monad calculus (Breazu-Tannen *et al.*, 1992; Breazu-Tannen & Subrahmanyam, 1991; Wadler, 1990; Grust & Scholl, 1999; Trinder, 1992; Buneman *et al.*, 1995) support complex data models but do not support aggregations and efficient equi-joins (Gibbons *et al.*, 2018). Monoid comprehensions solve the former issue (Fegaras & Maier, 2000), however, require an intermediate algebra to support equi-joins efficiently. Kleisli (Wong, 2000), BQL (Libkin & Wong, 1997), and Trance (Smith *et al.*, 2020) extend monad calculus with aggregations and bag semantics. Representing flat relations as bags has been investigated in AGCA (Koch *et al.*, 2014), FAQ (Abo Khamis *et al.*, 2016), and HoTTSQL (Chu *et al.*, 2017). SDQL extends all these approaches by allowing nested dictionaries and representing relations and intermediate group-by aggregates as dictionaries. Although monadic and monoid collection structures were observed, SDQL is the first work that introduces semi-ring dictionaries.

Language-Integrated Queries. LINQ (Meijer *et al.*, 2006) and Links (Cooper *et al.*, 2007) mainly aim to generate SQL or host language's code from nested functional queries. One of the main challenges for them is to resolve avalanche of queries during this translation, for which techniques such as query shredding has proved useful (Grust *et al.*, 2010; Cheney *et al.*, 2014). Comprehensive Comprehensions (CompComp) (Jones & Wadler, 2007) extend Haskell's list comprehensions with group-by and order-by. Rather than only serving as a frontend language and relying on the target language to perform optimizations, SDQL takes an approach similar to Kleisli (Wong, 2000); it directly translates nested collections to low-level code, and enables more aggressive optimizations.

Loop Fusion. Functional languages use deforestation (Wadler, 1988; Gill *et al.*, 1993; Svenningsson, 2002; Coutts *et al.*, 2007; Takano & Meijer, 1995; Emoto *et al.*, 2012) to remove unnecessary intermediate collections. This optimization is implemented by rewrite rule facilities of GHC (Jones *et al.*, 2001) in Haskell (Gill *et al.*, 1993), and also by using multi-stage programming in Scala (Jonnalagedda & Stucki, 2015; Kiselyov *et al.*, 2017; Shaikhha *et al.*, 2018*b*). Generalized stream fusion (Mainland *et al.*, 2013) combines deforestation with vectorization for Haskell. Functional array processing languages such as APL (Iverson, 1962), SAC (Grelck & Scholz, 2006), Futhark (Henriksen *et al.*, 2017), and $\widetilde{\text{F}}$ (Shaikhha *et al.*, 2019) also need to support loop fusion. Such languages mainly use pull and push arrays (Anker & Svenningsson, 2013; Claessen *et al.*, 2012; Svensson & Svenningsson, 2014; Axelsson *et al.*, 2011; Kiselyov, 2018; Shaikhha *et al.*, 2017) to remove unnecessary intermediate arrays. Even though these work support fusion for lists of key-value pairs, they do not support dictionaries. Thus, they do not have efficient support for operators such as grouping and hash join.

Linear Algebra Languages. DSLs such as Lift (Steuwer *et al.*, 2015), Halide (Ragan-Kelley *et al.*, 2013), Diderot (Chiw *et al.*, 2012), and OptiML (Sujeeth *et al.*, 2011) can generate parallel code from their high-level programs, while DSLs such as Spiral (Puschel *et al.*, 2005), LGen (Spampinato & Püschel, 2016; Spampinato *et al.*, 2018) exploit the

Table 2: Comparison of different data analytics approaches. ● means that the property is supported, ○ means that it is absent in the work, and ◐ means that the property is partially supported. For the corresponding sets of operators supported by (nested) relational and linear algebra refer to Figures 4-7.

	Expressiveness					Data Repr.					Specialization				
	Relational Algebra	Nested Rel. Calc.	Group-by Aggregates	Efficient Equi-Joins	Linear Algebra	Set & Bag	Dense Array	Sparse Tensor	Dictionary	Semi-rings	Loop Fusion	Loop Hoisting	Loop Memoization	Code Generation	Vectorization
SDQL (This Article)	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○
Query Compilers (HyPer)	●	○	●	●	○	●	●	○	●	○	◐	◐	○	●	○
Vectorized Query Engines (DuckDB)	●	○	●	●	○	●	●	○	●	○	◐	◐	○	○	●
Monad Calculus, NRC$^+$	●	●	○	○	○	●	○	○	○	○	◐	◐	○	○	○
Monoid Comprehension	●	●	●	○	○	●	○	○	○	○	◐	◐	○	○	○
Monad Calc. + Agg. (Kleisli, Trance)	●	●	●	○	◐	●	○	○	○	○	◐	◐	○	●	○
Lang. Integ. Queries (LINQ)	●	●	●	○	●	●	○	○	○	○	◐	◐	○	○	○
Functional Lists (Stream Fusion)	●	●	●	○	●	●	◐	○	○	○	●	◐	○	●	●
Functional APL (Futhark, SAC)	◐	◐	◐	○	●	◐	●	○	○	○	●	◐	◐	●	●
Dense LA Library (NumPy)	○	○	○	○	●	○	●	○	○	○	○	○	○	○	●
Dense LA DSL (Lift,Halide,LGen)	○	○	○	○	●	○	●	○	○	○	●	◐	○	●	●
Sparse LA Library (SPLATT, SciPy)	○	○	○	○	●	○	●	◐	○	○	○	○	○	○	◐
Sparse LA DSL (TACO)	○	○	○	○	●	○	●	●	○	○	◐	◐	○	●	○
Sparse LA + Semi-rings (GraphBLAS)	○	○	○	○	●	○	●	◐	○	●	○	○	○	○	◐
DB/LA by casting to LA (Morpheus)	◐	○	●	●	●	●	●	◐	○	○	○	○	○	○	●
DB/LA by casting to DB (LMFAO)	●	○	●	●	◐	●	●	◐	○	●	◐	◐	○	●	○
DB/LA by unified IR (IFAQ)	●	○	●	●	●	●	○	●	●	●	◐	●	◐	●	○
DB/LA by combined IR (Raven)	●	○	●	●	●	●	●	◐	○	○	◐	◐	◐	●	●

memory hierarchy and make careful decisions on tiling and scheduling decisions. These DSLs exploit the memory hierarchy by relying on searching algorithms for making tiling and scheduling decisions. The generated output is a C function that includes intrinsics to enable SIMD vector extensions. SPL (Xiong *et al.*, 2001) is a language that expresses recursion and mathematical formulas. TACO (Kjolstad *et al.*, 2017) generates efficient low-level code for compound linear algebra operations on dense and sparse matrices, with extensions to support additional semirings (Henry *et al.*, 2021). Finch (Ahrens *et al.*, 2024) and StructTensor (Ghorbani *et al.*, 2023) provide support for tensors with structured sparsity and redundancy. All these languages are limited to linear algebra workloads and do not support database workloads.

Semi-Ring Languages. The use of semi-rings for expressing graph problems as linear algebra is well-known (Kepner & Gilbert, 2011). This connection has been used for expressing path problems by solving matrix equations (Tarjan, 1981; Backhouse & Carré, 1975; Valiant, 1975). SDQL requires extensions in order to express such problems (cf. Section 9.4). GraphBLAS (Kepner *et al.*, 2016) is a framework for expressing graph problems in terms of sparse linear algebra. The functional languages has shown before an appropriate implementation choice for linear algebra languages with various semi-ring instances (Shaikhha & Parreaux, 2019; Dolan, 2013). In the DB world, K-relations (Green *et al.*, 2007) use semi-rings (Karvounarakis & Green, 2012) and semi-modules (Amsterdamer *et al.*, 2011) for encoding provenance information for relational algebra with aggregations. The pvc-tables (Fink *et al.*, 2012) are a representation system that use this idea to encode aggregations in databases with uncertainties. The closest work to ours is FAQ (Abo Khamis *et al.*, 2016), which provides a unified declarative interface for LA and DB. However, none of the existing work support nested data models.

DB/LA Query Languages. There has been a recent interest in the study on the expressive power of query languages for hybrid DB/LA tasks. Matrix query languages (Geerts *et al.*, 2021) such as MATLANG (Brijder *et al.*, 2019) and its extensions have shown to be connected to different fragments of relational algebra with aggregates. LARA (Hutchison *et al.*, 2017) is a query language over associative tables (flat dictionaries), with more expressive power than MATLANG (Brijder *et al.*, 2019). Associative algebra (Jananthan *et al.*, 2017) defines a query language over associative arrays (flat dictionaries, and without the ability to map between dictionaries of different value types) expressive enough for both database and linear algebra workloads. All these query languages are declarative and can only serve as frontend query languages; they need to rely on the techniques offered by other formalisms (e.g., FAQ (Abo Khamis *et al.*, 2016)) for optimizations. Indexed Streams (Kovach *et al.*, 2023) provides a formally verified compiler and can express both database and tensor algebra workloads. Furthermore, none of these languages support nested data like SDQL.

DB/LA Frameworks. Hybrid database and linear algebra workloads, such as training machine learning models over databases are increasingly gaining attention. Traditionally, these workloads are processed in two isolated environments: 1) the training data set is constructed using a database system or libraries such as Python Pandas, and then 2) the model is trained over the materialized dataset using frameworks such as scikit-learn (Pedregosa *et al.*, 2011), TensorFlow (Abadi *et al.*, 2016), PyTorch (Paszke *et al.*, 2017), etc. There has been some efforts on avoiding the separation of the environments by defining ML tasks as user-defined functions inside the database system such as MADlib (Hellerstein *et al.*, 2012), Bismarck (Feng *et al.*, 2012), and GLADE PF-OLA (Qin & Rusu, 2015); however, the training process is still executed after the training dataset is materialized.

Alternative approaches avoid the materialization of the training dataset. The current solutions are currently divided into four categories. First, systems such as Morpheus (Chen *et al.*, 2017; Li *et al.*, 2019) cast the in-DB ML task as a linear algebra problem on top of R (Chen *et al.*, 2017) and NumPy (Li *et al.*, 2019). An advantage of this system is that it benefits from efficient linear algebra frameworks (cf. Section 10.4). However, one requires to encode database knowledge in terms of linear algebra rewrite rules and implement query evaluation techniques for them (e.g., trie-based evaluation as observed in Section 10.4). The second category are systems such as F (Olteanu & Schleich, 2016; Schleich *et al.*, 2016), AC/DC (Khamis *et al.*, 2018), and LMFAO (Schleich *et al.*, 2019) that cast the in-DB ML task as a batch of aggregate queries. The third approach involves defining an intermediate representation (IR) that *combines* linear and relational algebra constructs together. Raven (Karanasos *et al.*, 2020) and MatRel (Yu *et al.*, 2021) are frameworks that provide such an IR. Implementing cross-domain optimizations requires developing new transformation rules for different combinations of linear and relational algebra constructs, which can be tedious and error-prone. The fourth category resolves this issue by defining a unified intermediate language that can express both workloads. Lara (Kunft *et al.*, 2019) provides a two-level IR. The first level combines linear and relational algebra constructs. The second level is based on monad-calculus and can perform cross-domain optimizations such as vertical loop fusion and selection push down. IFAQ (Shaikhha *et al.*, 2020, 2021b) introduces a single dictionary-based DSL for expressing the entire data science pipelines. SDQL also falls into the fourth category, and additionally supports nested data, dense representations, and more loop optimizations (cf. Table 2). Furthermore, to the best of our knowledge, SDQL is the only hybrid DB/LA framework for which type safety and the correctness of the optimizations are proved using denotational and operational semantics.

Acknowledgments

The first author thanks Huawei for their support of the distributed data management and processing laboratory at the University of Edinburgh. This work was funded in part by a gift from RelationalAI.

References

GLib: Library package for low-level data structures in C. https://developer.gnome.org/glib/2.38/.

Abadi, Martín, Barham, Paul, Chen, Jianmin, Chen, Zhifeng, Davis, Andy, Dean, Jeffrey, Devin, Matthieu, Ghemawat, Sanjay, Irving, Geoffrey, Isard, Michael, *et al.* . (2016). Tensorflow: A system for large-scale machine learning. *Pages 265–283 of: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. USA: USENIX Association.

Abo Khamis, Mahmoud, Ngo, Hung Q., & Rudra, Atri. (2016). FAQ: Questions Asked Frequently. *Pages 13–28 of: Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS '16. New York, NY, USA: Association for Computing Machinery.

Abo Khamis, Mahmoud, Ngo, Hung Q., Nguyen, XuanLong, Olteanu, Dan, & Schleich, Maximilian. (2018). In-database learning with sparse tensors. *Page 325–340 of: Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. SIGMOD/PODS '18. New York, NY, USA: Association for Computing Machinery.

Ahrens, Willow, Collin, Teodoro Fields, Patel, Radha, Deeds, Kyle, Hong, Changwan, & Amarasinghe, Saman P. (2024). Finch: Sparse and structured array programming with control flow. *Corr*, **abs/2404.16730**.

Aji, Srinivas M, & McEliece, Robert J. (2000). The generalized distributive law. *Ieee transactions on information theory*, **46**(2), 325–343.

Amsterdamer, Yael, Deutch, Daniel, & Tannen, Val. (2011). Provenance for aggregate queries. *Pages 153–164 of: Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*.

Anker, Johan, & Svenningsson, Josef. (2013). An EDSL approach to high performance haskell programming. *Pages 1–12 of: ACM Haskell Symposium*. New York, NY, USA: ACM.

Armbrust, Michael, Xin, Reynold S., Lian, Cheng, Huai, Yin, Liu, Davies, Bradley, Joseph K., Meng, Xiangrui, Kaftan, Tomer, Franklin, Michael J., Ghodsi, Ali, & Zaharia, Matei. (2015). Spark SQL: Relational Data Processing in Spark. *Pages 1383–1394 of: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. New York, NY, USA: ACM.

Axelsson, Emil, Claessen, Koen, Sheeran, Mary, Svenningsson, Josef, Engdal, David, & Persson, Anders. (2011). The Design and Implementation of Feldspar an Embedded Language for Digital Signal Processing. *Pages 121–136 of: Proceedings of the 22Nd International Conference on Implementation and Application of Functional Languages*. IFL'10. Berlin, Heidelberg: Springer-Verlag.

Backhouse, R. C., & Carré, B. A. (1975). Regular Algebra Applied to Path-finding Problems. *Ima journal of applied mathematics*, **15**(2), 161–186.

Bader, Brett W, & Kolda, Tamara G. (2008). Efficient matlab computations with sparse and factored tensors. *Siam journal on scientific computing*, **30**(1), 205–231.

Breazu-Tannen, Val, & Subrahmanyam, Ramesh. (1991). *Logical and computational aspects of programming with sets/bags/lists*. Springer.

Breazu-Tannen, Val, Buneman, Peter, & Wong, Limsoon. (1992). *Naturally embedded query languages*. Springer.

Brijder, Robert, Geerts, Floris, Bussche, Jan Van Den, & Weerwag, Timmy. (2019). On the expressive power of query languages for matrices. *Acm trans. database syst.*, **44**(4).

Buneman, Peter, Naqvi, Shamim, Tannen, Val, & Wong, Limsoon. (1995). Principles of programming with complex objects and collection types. *Theor. comput. sci.*, **149**(1), 3–48.

Carlson, Andrew, Betteridge, Justin, Kisiel, Bryan, Settles, Burr, Hruschka, Estevam, & Mitchell, Tom. (2010). Toward an architecture for never-ending language learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 24.

Chalmers, Zachary R., Connelly, Caitlin F., Fabrizio, David, Gay, Laurie, Ali, Siraj M., Ennis, Riley, Schrock, Alexa, Campbell, Brittany, Shlien, Adam, Chmielecki, Juliann, Huang, Franklin, He, Yuting, Sun, James, Tabori, Uri, Kennedy, Mark, Lieber, Daniel S., Roels, Steven, White, Jared, Otto, Geoffrey A., Ross, Jeffrey S., Garraway, Levi, Miller, Vincent A., Stephens, Phillip J., & Frampton, Garrett M. (2017). Analysis of 100,000 human cancer genomes reveals the landscape of tumor mutational burden. *Genome medicine*, **9**(1), 34.

Chen, Lingjiao, Kumar, Arun, Naughton, Jeffrey, & Patel, Jignesh M. (2017). Towards linear algebra over normalized data. *Proceedings of the vldb endowment*, **10**(11), 1214–1225.

Cheney, James, Lindley, Sam, & Wadler, Philip. (2014). Query shredding: efficient relational evaluation of queries over nested multisets. *Pages 1027–1038 of: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*.

Chiw, Charisee, Kindlmann, Gordon, Reppy, John, Samuels, Lamont, & Seltzer, Nick. (2012). Diderot: A Parallel DSL for Image Analysis and Visualization. *Pages 111–120 of: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI'12. ACM.

Chou, Stephen, Kjolstad, Fredrik, & Amarasinghe, Saman. (2018). Format abstraction for sparse tensor algebra compilers. *Proc. acm program. lang.*, **2**(OOPSLA).

Chu, Shumo, Weitz, Konstantin, Cheung, Alvin, & Suciu, Dan. (2017). Hottsql: Proving query rewrites with univalent sql semantics. *Acm sigplan notices*, **52**(6), 510–524.

Claessen, Koen, Sheeran, Mary, & Svensson, Bo Joel. (2012). Expressive array constructs in an embedded gpu kernel programming language. *Pages 21–30 of: Proceedings of the 7th Workshop on Declarative Aspects and Applications of Multicore Programming*. DAMP '12. NY, USA: ACM.

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

Committee, National Research Council (US). (2005). On the nature of biological data. *Chap. 3 of:* Wooley, John C., & Lin, Herbert S. (eds), *Catalyzing Inquiry at the Interface of Computing and Biology*. National Academies Press (US).

Conrad, Keith. (2018). Tensor products. *Notes of course, available on-line*.

Cooper, Ezra, Lindley, Sam, Wadler, Philip, & Yallop, Jeremy. (2007). Links: Web programming without tiers. *Pages 266–296 of: Proceedings of the 5th International Conference on Formal Methods for Components and Objects*. FMCO'06. Berlin, Heidelberg: Springer-Verlag.

Cormen, Thomas H, Leiserson, Charles E, Rivest, Ronald L, & Stein, Clifford. (2009). *Introduction to algorithms*. MIT press.

Coutts, Duncan, Leshchinskiy, Roman, & Stewart, Don. (2007). Stream fusion. from lists to streams to nothing at all. *ICFP '07*.

Crotty, Andrew, Galakatos, Alex, Dursun, Kayhan, Kraska, Tim, Çetintemel, Ugur, & Zdonik, Stanley B. (2015). Tupleware:" big" data, big analytics, small clusters. *CIDR*.

Dolan, Stephen. (2013). Fun with semirings: A functional pearl on the abuse of linear algebra. *Pages 101–110 of: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. New York, NY, USA: Association for Computing Machinery.

Emoto, Kento, Fischer, Sebastian, & Hu, Zhenjiang. (2012). Filter-embedding semiring fusion for programming with mapreduce. *Formal aspects of computing*, **24**(4), 623–645.

Fancello, Laura, Gandini, Sara, Pelicci, Pier Giuseppe, & Mazzarella, Luca. (2019). Tumor mutational burden quantification from targeted gene panels: major advancements and challenges. *Journal for immunotherapy of cancer*, **7**(1), 183.

Favorita, Corporacion. 2017 (October). *Corp. Favorita Grocery Sales Forecasting: Can you accurately predict sales for a large grocery chain?*

Fegaras, Leonidas, & Maier, David. (2000). Optimizing object queries using an effective calculus. *Acm trans. database syst.*, **25**(4), 457–516.

Feng, Xixuan, Kumar, Arun, Recht, Benjamin, & Ré, Christopher. (2012). Towards a unified architecture for in-rdbms analytics. *Pages 325–336 of: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. New York, NY, USA: ACM.

Fink, Robert, Han, Larisa, & Olteanu, Dan. (2012). Aggregation in probabilistic databases via knowledge compilation. *Proc. VLDB endow.*, **5**(5), 490–501.

Geerts, Floris, Muñoz, Thomas, Riveros, Cristian, Van den Bussche, Jan, & Vrgoč, Domagoj. (2021). Matrix query languages. *Acm sigmod record*, **50**(3), 6–19.

Ghorbani, Mahdi, Huot, Mathieu, Hashemian, Shideh, & Shaikhha, Amir. (2023). Compiling structured tensor algebra. *Proc. ACM program. lang.*, **7**(OOPSLA2), 204–233.

Gibbons, Jeremy, Henglein, Fritz, Hinze, Ralf, & Wu, Nicolas. (2018). Relational algebra by way of adjunctions. *Proc. acm program. lang.*, **2**(ICFP).

Gill, Andrew, Launchbury, John, & Peyton Jones, Simon L. (1993). A short cut to deforestation. *Pages 223–232 of: Proceedings of the conference on Functional programming languages and computer architecture*. FPCA. ACM.

Gondran, Michel, & Minoux, Michel. (2008). *Graphs, dioids and semirings: new models and algorithms*. Vol. 41. Springer Science & Business Media.

Green, Todd J, Karvounarakis, Grigoris, & Tannen, Val. (2007). Provenance semirings. *Pages 31–40 of: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*.

Grelck, Clemens, & Scholz, Sven-Bodo. (2006). SAC—A functional array language for efficient multi-threaded execution. *Int. journal of parallel programming*, **34**(4), 383–427.

Grust, Torsten, & Scholl, MarcH. (1999). How to comprehend queries functionally. *Journal of intelligent information systems*, **12**(2-3), 191–218.

Grust, Torsten, Rittinger, Jan, & Schreiber, Tom. (2010). Avalanche-safe LINQ compilation. *Pvldb*, **3**(1-2), 162–172.

Hellerstein, Joseph M, Ré, Christoper, Schoppmann, Florian, Wang, Daisy Zhe, Fratkin, Eugene, Gorajek, Aleksander, Ng, Kee Siong, Welton, Caleb, Feng, Xixuan, Li, Kun, *et al.* . (2012). The madlib analytics library: or mad skills, the sql. *Proceedings of the vldb endowment*, **5**(12), 1700–1711.

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

Henry, Rawn, Hsu, Olivia, Yadav, Rohan, Chou, Stephen, Olukotun, Kunle, Amarasinghe, Saman P., & Kjolstad, Fredrik. (2021). Compilation of sparse array programming models. *Proc. ACM program. lang.*, **5**(OOPSLA), 1–29.

Hutchison, Dylan, Howe, Bill, & Suciu, Dan. (2017). Laradb: A minimalist kernel for linear and relational algebra computation. *Pages 1–10 of: Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*.

Idreos, S, Groffen, F, Nes, N, Manegold, S, Mullender, S, & Kersten, M. (2012). Monetdb: Two decades of research in column-oriented database. *Ieee data engineering bulletin*.

Iverson, Kenneth E. (1962). A Programming Language. *Pages 345–351 of: Proceedings of the May 1-3, 1962, spring joint computer conference*. ACM.

Jananthan, Hayden, Zhou, Ziqi, Gadepally, Vijay, Hutchison, Dylan, Kim, Suna, & Kepner, Jeremy. (2017). Polystore mathematics of relational algebra. *Pages 3180–3189 of: 2017 IEEE International Conference on Big Data (Big Data)*. IEEE.

Jones, Simon Peyton, & Wadler, Philip. (2007). Comprehensive comprehensions. *Pages 61–72 of: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*.

Jones, Simon Peyton, Tolmach, Andrew, & Hoare, Tony. (2001). Playing by the rules: rewriting as a practical optimisation technique in ghc. *Pages 203–233 of: Haskell workshop*, vol. 1.

Jonnalagedda, Manohar, & Stucki, Sandro. (2015). Fold-based fusion as a library: A generative programming pearl. *Pages 41–50 of: Proceedings of the 6th ACM SIGPLAN Symposium on Scala*. ACM.

Karanasos, Konstantinos, Interlandi, Matteo, Xin, Doris, Psallidas, Fotis, Sen, Rathijit, Park, Kwanghyun, Popivanov, Ivan, Nakandal, Supun, Krishnan, Subru, Weimer, Markus, *et al.* . (2020). Extending relational query processing with ml inference. *CIDR*.

Karpathiotakis, Manos, Alagiannis, Ioannis, Heinis, Thomas, Branco, Miguel, & Ailamaki, Anastasia. (2015). Just-in-time data virtualization: Lightweight data management with vida. *CIDR*.

Karvounarakis, Grigoris, & Green, Todd J. (2012). Semiring-annotated data: queries and provenance? *Acm sigmod record*, **41**(3), 5–14.

Keller, Gabriele, Chakravarty, Manuel MT, Leshchinskiy, Roman, Peyton Jones, Simon, & Lippmeier, Ben. (2010). Regular, shape-polymorphic, parallel arrays in haskell. *Acm sigplan notices*, **45**(9), 261–272.

Kepner, Jeremy, & Gilbert, John. (2011). *Graph algorithms in the language of linear algebra*. Vol. 22. SIAM.

Kepner, Jeremy, Aaltonen, Peter, Bader, David, Buluç, Aydin, Franchetti, Franz, Gilbert, John, Hutchison, Dylan, Kumar, Manoj, Lumsdaine, Andrew, Meyerhenke, Henning, *et al.* . (2016). Mathematical foundations of the graphblas. *Pages 1–9 of: 2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE.

Khamis, Mahmoud Abo, Ngo, Hung Q., Nguyen, XuanLong, Olteanu, Dan, & Schleich, Maximilian. (2018). Ac/dc: In-database learning thunderstruck. *Pages 8:1–8:10 of: Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*. DEEM'18. New York, NY, USA: ACM.

Kiselyov, Oleg. (2018). Reconciling abstraction with high performance: A metaocaml approach. *Foundations and trends in programming languages*, **5**(1), 1–101.

Kiselyov, Oleg, Biboudis, Aggelos, Palladinos, Nick, & Smaragdakis, Yannis. (2017). Stream fusion, to completeness. *Pages 285–299 of: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. New York, NY, USA: ACM.

Kjolstad, Fredrik, Kamil, Shoaib, Chou, Stephen, Lugato, David, & Amarasinghe, Saman. (2017). The tensor algebra compiler. *Proc. acm program. lang.*, **1**(OOPSLA), 77:1–77:29.

Koch, Christoph, Ahmad, Yanif, Kennedy, Oliver, Nikolic, Milos, Nötzli, Andres, Lupei, Daniel, & Shaikhha, Amir. (2014). DBToaster: higher-order delta processing for dynamic, frequently fresh views. *Vldbj*, **23**(2), 253–278.

Koch, Christoph, Lupei, Daniel, & Tannen, Val. (2016). Incremental view maintenance for collection programming. *Page 75–90 of: Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS '16. New York, NY, USA: Association for Computing Machinery.

Kovach, Scott, Kolichala, Praneeth, Gu, Tiancheng, & Kjolstad, Fredrik. (2023). Indexed streams: A formal intermediate representation for fused contraction programs. *Proc. ACM program. lang.*,

7(PLDI), 1169–1193.

Krikellas, Konstantinos, Viglas, Stratis, & Cintra, Marcelo. (2010). Generating code for holistic query evaluation. *Pages 613–624 of: ICDE*.

Kunft, Andreas, Katsifodimos, Asterios, Schelter, Sebastian, Breß, Sebastian, Rabl, Tilmann, & Markl, Volker. (2019). An intermediate representation for optimizing machine learning pipelines. *Proceedings of the vldb endowment*, **12**(11), 1553–1567.

Lehmann, Daniel J. (1977). Algebraic structures for transitive closure. *Theoretical computer science*, **4**(1), 59–76.

Li, Side, Chen, Lingjiao, & Kumar, Arun. (2019). Enabling and optimizing non-linear feature interactions in factorized linear algebra. *Pages 1571–1588 of: Proceedings of the 2019 International Conference on Management of Data*. ACM.

Libkin, Leonid, & Wong, Limsoon. (1997). Query languages for bags and aggregate functions. *Journal of computer and system sciences*, **55**(2), 241–272.

Mainland, Geoffrey, Leshchinskiy, Roman, & Peyton Jones, Simon. (2013). Exploiting Vector Instructions with Generalized Stream Fusion. *Pages 37–48 of: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP'13. New York, NY, USA: ACM.

Masseroli, Marco, Pinoli, Pietro, Venco, Francesco, Kaitoua, Abdulrahman, Jalili, Vahid, Palluzzi, Fernando, Muller, Heiko, & Ceri, Stefano. (2015). GenoMetric Query Language: A Novel Approach to Large-scale Genomic Data Management. *Bioinformatics*, **31**(12), 1881–1888.

Meijer, Erik, Beckman, Brian, & Bierman, Gavin. (2006). LINQ: Reconciling Object, Relations and XML in the .NET Framework. *Pages 706–706 of: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. ACM.

Meng, Xiangrui, Bradley, Joseph, Yavuz, Burak, Sparks, Evan, Venkataraman, Shivaram, Liu, Davies, Freeman, Jeremy, Tsai, DB, Amde, Manish, Owen, Sean, Xin, Doris, Xin, Reynold, Franklin, Michael J., Zadeh, Reza, Zaharia, Matei, & Talwalkar, Ameet. (2016). Mllib: Machine learning in apache spark. *The journal of machine learning research*, **17**(1), 1235–1241.

Moerkotte, Guido, & Neumann, Thomas. (2011). Accelerating queries with group-by and join by groupjoin. *Proceedings of the vldb endowment*, **4**(11).

Mohri, Mehryar. (2002). Semiring frameworks and algorithms for shortest-distance problems. *Journal of automata, languages and combinatorics*, **7**(3), 321–350.

Nagel, Fabian, Bierman, Gavin, & Viglas, Stratis D. (2014). Code generation for efficient query processing in managed runtimes. *Pvldb*, **7**(12), 1095–1106.

Neumann, Thomas. (2011). Efficiently Compiling Efficient Query Plans for Modern Hardware. *Pvldb*, **4**(9), 539–550.

Nikolic, Milos, & Olteanu, Dan. (2018). Incremental view maintenance with triple lock factorization benefits. *Pages 365–380 of: Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. New York, NY, USA: ACM.

Olteanu, Dan, & Schleich, Maximilian. (2016). Factorized databases. *Sigmod rec.*, **45**(2), 5–16.

Palkar, Shoumik, Thomas, James J, Shanbhag, Anil, Narayanan, Deepak, Pirk, Holger, Schwarzkopf, Malte, Amarasinghe, Saman, Zaharia, Matei, & InfoLab, Stanford. (2017). Weld: A common runtime for high performance data analytics. *Conference on Innovative Data Systems Research (CIDR)*.

Paszke, Adam, Gross, Sam, Chintala, Soumith, Chanan, Gregory, Yang, Edward, DeVito, Zachary, Lin, Zeming, Desmaison, Alban, Antiga, Luca, & Lerer, Adam. (2017). Automatic Differentiation in PyTorch. *NIPS 2017 Autodiff Workshop: The Future of Gradient-based Machine Learning Software and Techniques*.

Pedregosa, Fabian, Varoquaux, Gaël, Gramfort, Alexandre, Michel, Vincent, Thirion, Bertrand, Grisel, Olivier, Blondel, Mathieu, Prettenhofer, Peter, Weiss, Ron, Dubourg, Vincent, *et al.* . (2011). Scikit-learn: Machine learning in python. *Journal of machine learning research*, **12**(Oct), 2825–2830.

Puschel, Markus, Moura, José MF, Johnson, Jeremy R, Padua, David, Veloso, Manuela M, Singer, Bryan W, Xiong, Jianxin, Franchetti, Franz, Gacic, Aca, Voronenko, Yevgen, *et al.* . (2005).

SPIRAL: code generation for DSP transforms. *Proceedings of the ieee*, **93**(2), 232–275.

Qin, Chengjie, & Rusu, Florin. (2015). Speculative approximations for terascale distributed gradient descent optimization. *Page 1 of: Proceedings of the Fourth Workshop on Data analytics in the Cloud*. ACM.

Raasveldt, Mark, & Mühleisen, Hannes. (2019). Duckdb: an embeddable analytical database. *Pages 1981–1984 of:* Boncz, Peter A., Manegold, Stefan, Ailamaki, Anastasia, Deshpande, Amol, & Kraska, Tim (eds), *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM.

Ragan-Kelley, Jonathan, Barnes, Connelly, Adams, Andrew, Paris, Sylvain, Durand, Frédo, & Amarasinghe, Saman. (2013). Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Pages 519–530 of: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI'13. New York, NY, USA: ACM.

Ramakrishnan, Raghu, & Gehrke, Johannes. (2000). *Database management systems*. 2nd edn. Osborne/McGraw-Hill.

Roth, Mark A, Korth, Herry F, & Silberschatz, Abraham. (1988). Extended algebra and calculus for nested relational databases. *Acm transactions on database systems (tods)*, **13**(4), 389–417.

Schleich, Maximilian, Olteanu, Dan, & Ciucanu, Radu. (2016). Learning linear regression models over factorized joins. *Pages 3–18 of: Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. New York, NY, USA: ACM.

Schleich, Maximilian, Olteanu, Dan, Abo Khamis, Mahmoud, Ngo, Hung Q., & Nguyen, XuanLong. (2019). A layered aggregate engine for analytics workloads. *Pages 1642–1659 of: Proceedings of the 2019 International Conference on Management of Data*. SIGMOD '19. New York, NY, USA: ACM.

Schleich, Maximilian, Shaikhha, Amir, & Suciu, Dan. (2023). Optimizing tensor programs on flexible storage. *Proc. ACM manag. data*, **1**(1), 37:1–37:27.

Shahrokhi, Hesam, & Shaikhha, Amir. (2023a). Building a compiled query engine in python. *Pages 180–190 of:* Verbrugge, Clark, Lhoták, Ondrej, & Shen, Xipeng (eds), *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction, CC 2023, Montréal, QC, Canada, February 25-26, 2023*. ACM.

Shahrokhi, Hesam, & Shaikhha, Amir. (2023b). An efficient vectorized hash table for batch computations. *Pages 27:1–27:27 of:* Ali, Karim, & Salvaneschi, Guido (eds), *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*. LIPIcs, vol. 263. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Shahrokhi, Hesam, Groeger, Callum, Yang, Yizhuo, & Shaikhha, Amir. (2023). Efficient query processing in python using compilation. *Pages 199–202 of:* Das, Sudipto, Pandis, Ippokratis, Candan, K. Selçuk, & Amer-Yahia, Sihem (eds), *Companion of the 2023 International Conference on Management of Data, SIGMOD/PODS 2023, Seattle, WA, USA, June 18-23, 2023*. ACM.

Shahrokhi, Seyed Hesamoddin. (2024). *Compilation and code generation for efficient data science*. Ph.D. thesis, School of Informatics.

Shaikhha, Amir. (2022). Deep fusion for efficient nested recursive computations. *Pages 33–44 of:* Scholz, Bernhard, & Kameyama, Yukiyoshi (eds), *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2022, Auckland, New Zealand, December 6-7, 2022*. ACM.

Shaikhha, Amir, & Parreaux, Lionel. (2019). Finally, a Polymorphic Linear Algebra Language. *Proceedings of the 33rd European Conference on Object-Oriented Programming*. ECOOP'19.

Shaikhha, Amir, Klonatos, Yannis, Parreaux, Lionel, Brown, Lewis, Dashti, Mohammad, & Koch, Christoph. (2016). How to architect a query compiler. *Pages 1907–1922 of: Proceedings of the 2016 International Conference on Management of Data*. SIGMOD'16. New York, NY, USA: ACM.

Shaikhha, Amir, Fitzgibbon, Andrew, Peyton Jones, Simon, & Vytiniotis, Dimitrios. (2017). Destination-passing Style for Efficient Memory Management. *Pages 12–23 of: Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*.

FHPC 2017. New York, NY, USA: ACM.

Shaikhha, Amir, Klonatos, Yannis, & Koch, Christoph. (2018a). Building efficient query engines in a high-level language. *Acm transactions on database systems*, **43**(1), 4:1–4:45.

Shaikhha, Amir, Dashti, Mohammad, & Koch, Christoph. (2018b). Push versus Pull-Based Loop Fusion in Query Engines. *Journal of functional programming*, **28**, e10.

Shaikhha, Amir, Fitzgibbon, Andrew, Vytiniotis, Dimitrios, & Peyton Jones, Simon. (2019). Efficient differentiable programming in a functional array-processing language. *Proceedings of the acm on programming languages*, **3**(ICFP), 97.

Shaikhha, Amir, Schleich, Maximilian, Ghita, Alexandru, & Olteanu, Dan. (2020). Multi-layer optimizations for end-to-end data analytics. *Page 145–157 of: CGO*.

Shaikhha, Amir, Huot, Mathieu, Smith, Jaclyn, & Olteanu, Dan. (2021a). Functional collection programming with semi-ring dictionaries. *Corr*, **abs/2103.06376**.

Shaikhha, Amir, Schleich, Maximilian, & Olteanu, Dan. (2021b). An intermediate representation for hybrid database and machine learning workloads. *Proc. VLDB endow.*, **14**(12), 2831–2834.

Shaikhha, Amir, Huot, Mathieu, Smith, Jaclyn, & Olteanu, Dan. (2022). Functional collection programming with semi-ring dictionaries. *Proc. ACM program. lang.*, **6**(OOPSLA1), 1–33.

Shaikhha, Amir, Huot, Mathieu, & Hashemian, Shideh. (2024). A tensor algebra compiler for sparse differentiation. *Pages 1–12 of:* Grosser, Tobias, Dubach, Christophe, Steuwer, Michel, Xue, Jingling, Ottoni, Guilherme, & ernando Magno Quintão Pereira (eds), *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2024, Edinburgh, United Kingdom, March 2-6, 2024*. IEEE.

Side Li, Arun Kumar. (2019a). *Morpheuspy*. https://github.com/ADALabUCSD/MorpheusPy.

Side Li, Arun Kumar. (2019b). *Morpheuspy – issue #3*. https://github.com/ADALabUCSD/MorpheusPy/issues/3.

Smith, Jaclyn, Benedikt, Michael, Nikolic, Milos, & Shaikhha, Amir. (2020). Scalable querying of nested data. *Proceedings of the vldb endowment*, **14**(3), 445–457.

Smith, Shaden, Ravindran, Niranjay, Sidiropoulos, Nicholas D, & Karypis, George. (2015). Splatt: Efficient and parallel sparse tensor-matrix multiplication. *Pages 61–70 of: 2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE.

Spampinato, Daniele G., & Püschel, Markus. (2016). A basic linear algebra compiler for structured matrices. *Pages 117–127 of: Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM.

Spampinato, Daniele G., Fabregat-Traver, Diego, Bientinesi, Paolo, & Püschel, Markus. (2018). Program generation for small-scale linear algebra applications. *Pages 327–339 of: Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. New York, NY, USA: ACM.

Steuwer, Michel, Fensch, Christian, Lindley, Sam, & Dubach, Christophe. (2015). Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code. *Pages 205–217 of: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015. New York, NY, USA: ACM.

Sudmant, Peter H., Rausch, Tobias, Gardner, Eugene J., Handsaker, Robert E., Abyzov, Alexej, Huddleston, John, Zhang, Yan, Ye, Kai, Jun, Goo, Hsi-Yang Fritz, Markus, *et al.* . (2015). An integrated map of structural variation in 2,504 human genomes. *Nature*, **526**(7571), 75–81.

Sujeeth, Arvind, Lee, HyoukJoong, Brown, Kevin, Rompf, Tiark, Chafi, Hassan, Wu, Michael, Atreya, Anand, Odersky, Martin, & Olukotun, Kunle. (2011). OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. *Pages 609–616 of: Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. ICML '11.

Svenningsson, Josef. (2002). Shortcut fusion for accumulating parameters & zip-like functions. *Pages 124–132 of: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*. ICFP '02. ACM.

Svensson, Bo Joel, & Svenningsson, Josef. (2014). Defunctionalizing push arrays. *Pages 43–52 of: Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*. FHPC '14. NY, USA: ACM.

Tahboub, Ruby Y, Essertel, Grégory M, & Rompf, Tiark. (2018). How to architect a query compiler, revisited. *Pages 307–322 of: Proceedings of the 2018 International Conference on Management of Data*.

Takano, Akihiko, & Meijer, Erik. (1995). Shortcut deforestation in calculational form. *Page 306–313 of: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. FPCA '95. New York, NY, USA: Association for Computing Machinery.

Tarjan, Robert Endre. (1981). A unified approach to path problems. *J. acm*, **28**(3), 577–593.

Team, Hail. (2020). *Hail 0.2*. https://github.com/hail-is/hail.

Trinder, Phil. (1992). Comprehensions, a Query Notation for DBPLs. *Pages 55–68 of: Proc. of the 3rd DBPL workshop*. DBPL3. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Valiant, Leslie G. (1975). General context-free recognition in less than cubic time. *Journal of computer and system sciences*, **10**(2), 308–315.

Vasilache, Nicolas, Zinenko, Oleksandr, Theodoridis, Theodoros, Goyal, Priya, DeVito, Zachary, Moses, William S, Verdoolaege, Sven, Adams, Andrew, & Cohen, Albert. (2018). Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arxiv preprint arxiv:1802.04730*.

Veldhuizen, Todd L. (2014). Leapfrog Triejoin: A Simple, Worst-Case Optimal Join Algorithm. *Pages 96–106 of: Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*.

Viglas, Stratis, Bierman, Gavin M., & Nagel, Fabian. (2014). Processing Declarative Queries Through Generating Imperative Code in Managed Runtimes. *IEEE data eng. bull.*, **37**(1), 12–21.

Voss, Kate, Gentry, Jeff, & Auwera, Geraldine Van Der. (2017). Full-stack genomics pipelining with GATK4+ WDL+ Cromwell [version 1; not peer reviewed]. *F1000research*, 4.

Wadler, Philip. (1988). Deforestation: Transforming programs to eliminate trees. *Pages 344–358 of: ESOP'88*. Springer.

Wadler, Philip. (1990). Comprehending monads. *Pages 61–78 of: Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. LFP '90. New York, NY, USA: ACM.

Wang, Yisu Remy, Willsey, Max, & Suciu, Dan. (2023). Free join: Unifying worst-case optimal and traditional joins. *Proc. ACM manag. data*, **1**(2), 150:1–150:23.

Wong, Limsoon. (2000). Kleisli, a functional query system. *Journal of functional programming*, **10**(1), 19–56.

Xiong, Jianxin, Johnson, Jeremy, Johnson, Robert, & Padua, David. (2001). SPL: A Language and Compiler for DSP Algorithms. *Pages 298–308 of: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. PLDI'01. New York, NY, USA: ACM.

Yan, Weipeng P., & Larson, Per-Åke. (1994). Performing group-by before join. *Page 89–100 of: Proceedings of the Tenth International Conference on Data Engineering*. USA: IEEE Computer Society.

Yu, Yongyang, Tang, Mingjie, & Aref, Walid G. (2021). Scalable relational query processing on big matrix data. *arxiv preprint arxiv:2110.01767*.

Zukowski, Marcin, Boncz, Peter A, Nes, Niels, & Héman, Sándor. (2005). MonetDB/X100 - A DBMS In The CPU Cache. *Ieee data eng. bull.*, **28**(2), 17–22.