Functional Collection Programming with Semi-ring Dictionaries

AMIR SHAIKHHA, University of Edinburgh, United Kingdom
MATHIEU HUOT, University of Oxford, United Kingdom
JACLYN SMITH, University of Oxford, United Kingdom
DAN OLTEANU, University of Zurich, Switzerland

This paper 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, 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 behind 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 be competitive with or outperforms a host of systems that are state of the art in their own domain: in-memory DB systems Typer and Tectorwise for (flat, not nested) relational data; SciPy for LA workloads; sparse tensor compiler taco; the Trance nested relational engine; and the in-database machine learning engines LMFAO and Morpheus for hybrid DB/LA workloads over relational data.

CCS Concepts: • **Software and its engineering** → *Domain specific languages*; • **Computing methodologies** → *Linear algebra algorithms*; • **Information systems** → *Query languages*.

Additional Key Words and Phrases: Semi-Ring Dictionary, Sparse Linear Algebra, Nested Relational Algebra.

ACM Reference Format:
Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. 2022. Functional Collection Programming with Semi-ring Dictionaries. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 89 (April 2022), 33 pages. https://doi.org/10.1145/3527333

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 [Kjolstad et al. 2017; Vasilache et al. 2018].

The main contribution of this paper is SDQL, a purely functional language that is simple, canonical, efficient, and expressive enough for hybrid database (DB) and linear algebra (LA) workloads.

Authors' addresses: Amir Shaikhha, University of Edinburgh, United Kingdom; Mathieu Huot, University of Oxford, United Kingdom; Jaclyn Smith, University of Oxford, United Kingdom; Dan Olteanu, University of Zurich, Switzerland.

2475-1421/2022/4-ART89
https://doi.org/10.1145/3527333

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 and Larson 1994] and *matrix chain ordering* [Cormen et al. 2009], respectively.

Contributions. This paper 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/type system of SDQL keeps 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-database 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 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).
- We give denotational semantics using 0-preserving functions between K-semi-modules, and prove the correctness of SDQL optimizations (Section 7).
- We implemented a prototype compiler and runtime for SDQL (Section 8). We show experimentally (Section 9) 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 systems Typer and Tectorwise. 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.

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

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

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

Motivating Example. The following setting is used throughout the paper 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 [Chalmers et al. 2017; Fancello et al. 2019]. 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 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 paper 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`:

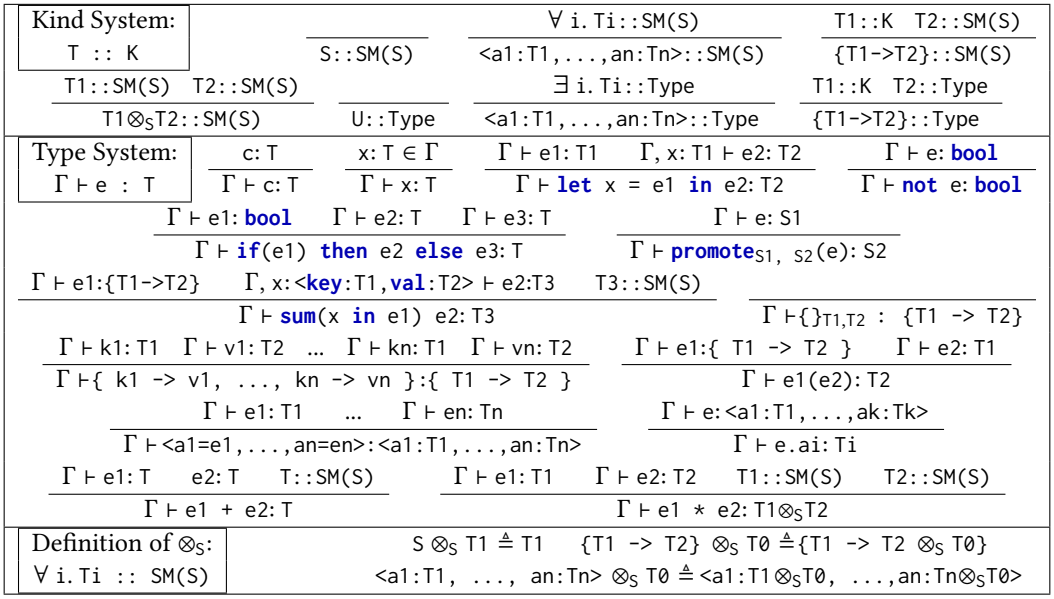


Fig. 2. Kind System and Type System of SDQL.

```
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 and 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⊗ST2` is another `S`-semi-module. It comes equipped with a canonical map which we also denote using `*`: `T1×T2⟶ T1⊗ST2` 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

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

Name	Type	Domain	Addition	Multiplication	Zero	One	Ring
Real Sum-Product	`real`	$\mathbb{R}$	$+$	$\times$	`0`	1	✓
Integer Sum-Product	`int`	$\mathbb{Z}$	$+$	$\times$	`0`	1	✓
Natural 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`	✗

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.

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 `promote`$_{\texttt{S1,S2}}$`(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 $\{\}_{\texttt{K,V}}$ 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 is point-wise, that is the values of elements with the same key are added. The elements of an SD with $\mathbf{0}_{\texttt{V}}$ 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 $\mathbf{0}_{\texttt{V}}$-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 }`.

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

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

Fig. 3. Extended constructs of SDQL.

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 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).

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.

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 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 supplementary materials.

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 difference) to SDQL. The explanation on the translation of its constructs can be found in the supplementary material.

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:

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

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

Variants	contig	start	reference	alternate	genotypes	
					sample	**call**
	17	43093817	C	A	TCGA-AN-A046	0
					TCGA-BH-A0B6	1
					sample	**call**
	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.

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

Fig. 6. Translation of aggregate operators of relational algebra and NRC^{agg} [Smith et al. 2020] to SDQL.

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 supplementary materials.

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 to have 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)}
```

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

In this section, we show 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

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(x in [[`V_1`]] ) { x.key->x.val*[[`V_2`]](x.key) }`	i,i->i
Dot Prod.	$[\![V_1 \cdot V_2]\!]$ = `sum(x in [[`V_1`]] ) x.val * [[`V_2`]](x.key)`	i,i->
Summation	$[\![\sum_{a \in V} a]\!]$ = `sum(x in [[`V`]] ) x.val`	i->
Matrix Operations:		
Transpose	$[\![M^T]\!]$ = `sum(x in [[`M`]] ) { <row=x.key.col, col=x.key.row> -> x.val }`	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(x in [[`M_1`]] ) { x.key -> x.val * [[`M_2`]](x.key) }`	ij,ij->ij
Matrix-Matrix Multiplication	$[\![M_1 \times M_2]\!]$ = `sum(x in [[`M_1`]] ) sum(y in [[`M_2`]] ) if(x.key.col == y.key.row) then { <row=x.key.row,col=y.key.col> -> x.val*y.val }`	ij,jk->ik
Mat-Vec. Mul.	$[\![M \cdot V]\!]$ = `sum(x in [[`M`]]) {x.key.row->x.val*[[`V`]](x.key.col)}`	ij,j->i
Trace	$[\![Tr(M)]\!]$ = `sum(<k,v> in [[`M`]]) if(k.row==k.col) then v`	ii->

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

show the translation of linear algebra operators to SDQL expressions together with 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 $\begin{bmatrix} a_0b_0 & 0 & a_1b_2 & 0 \end{bmatrix}$.

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 in the supplementary material.

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: $\begin{bmatrix} c_0a_0 + c_1a_2 & 0 \end{bmatrix}$.

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

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}
```
⇝
```
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)}
```
⇝
```
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)
```
⇝
```
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)
```
⇝
```
e2 * sum(x in e1) f(x)
```

```
sum(x in e1) f(x) * e2
```
⇝
```
(sum(x in e1) f(x)) * e2
```

Loop-Invariant Code Motion:

```
sum(x in e1) let y = e2 in f(x, y)
```
⇝
```
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)
```
⇝
```
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)
```
⇝
```
let tmp=sum(x in e1) {p(x)->f(x)}
in tmp(e2)
```

Fig. 8. Transformation rules for loop optimizations.

is computed as A^TA. 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 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 [Coutts et al. 2007; Gill et al. 1993; Svenningsson 2002; Wadler 1988]. 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 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 and Gehrke 2000] has the same role as vertical loop fusion [Shaikhha et al. 2018a]. Similarly, in functional array processing languages, pull arrays and push arrays [Anker and Svenningsson 2013; Claessen et al. 2012; Svensson and 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.

```
let R1 = sum(<r,r_v> in R) { f1(r) -> r_v }
in sum(<r1,r1_v> in R1) { f2(r1) -> r1_v }
```
$\leadsto$
```
sum(<r,r_v> in R)
  { 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 }
```
$\leadsto$
```
let R1 = sum(<r,r_v> in R) { r -> p1(r)*r_v }
in sum(<r1,r1_v> in R1) { r1 -> p2(r1)*r1_v }
```
$\leadsto$
```
sum(<r,r_v> in R)
  { r -> p1(r)*p2(r)*r_v }
```

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

```
let Vt = sum(<row,x> in V1) { row -> x * V2(row) }
in sum(<row,x1> in Vt) { row -> x1 * V3(row) }
```
$\leadsto$
```
sum(<row,x> in V1) { row ->
  x * V2(row) * V3(row) }
```

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

```
let Rsum = sum(<r,r_v> in R) r.A * r_v in
let Rcount = sum(<r,r_v> in R) r_v in
Rsum / Rcount
```
$\leadsto$
```
let RsumRcount = sum(<r,r_v> in R)
  < Rsum = r.A * r_v, Rcount = r_v >
in RsumRcount.Rsum / RsumRcount.Rcount
```

(d) Horizontal fusion for the average computation.

```
sum(<x,x_v> in NR)
  sum(<y,y_v> in x.C) x.A*x_v*y.D*y_v
```
$\leadsto$
```
sum(<x,x_v> in NR)
  x.A * x_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(<y,y_v> in x.C)
  { x.B -> x.A * x_v * y.D * y_v }
```
$\leadsto$
```
sum(<x,x_v> in NR) sum(<y,y_v> in x.C)
  { x.B -> 1 } * x.A * x_v * y.D * y_v
```
$\leadsto$
```
sum(<x,x_v> in NR) {x.B->1}*x.A*x_v*
  (sum(<y,y_v> in x.C) y.D * y_v)
```
$\leadsto$
```
sum(<x,x_v> in NR) {x.B -> x.A*x_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(<y,y_v> in x.C)
    let E = S(x.B) in
    x.A*x_v*E*y.D*y_v
```
$\leadsto$
```
sum(<x,x_v> in NR)
  let E = S(x.B) in
  sum(<y,y_v> in x.C)
    x.A*x_v*E*y.D*y_v
```
$\leadsto$
```
sum(<x,x_v> in NR)
  let E = S(x.B) in
  x.A*x_v*E*(
  sum(<y,y_v> in x.C) 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.

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 by attribute `B`, represented in the first expression of Figure 9f. According to the semantics of SDQL (cf. Section 7), 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 `jkR(r)` and `jkS(s)`, represented as the first expression in Figure 10a. This

```
sum(<r,r_v> in R)
 sum(<s,s_v> in S)
  if(jkR(r)==jkS(s)) then
  { concat(r,s)->r_v*s_v }
```
⇝
```
sum(<r,r_v> in R)
 let Sp = sum(<s,s_v> in S)
  { jkS(s) -> {s->s_v} } in
 sum(<s,s_v> in Sp(jkR(r)))
  { concat(r,s)->r_v*s_v }
```
⇝
```
let Sp = sum(<s,s_v> in S)
 { jkS(s) -> {s->s_v} } in
sum(<r,r_v> in R)
 sum(<s,s_v> in Sp(jkR(r)))
  { 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(jkR(r)==jkS(s)) then
  { jkR(r)->f(r)*g(s) }
```
⇝
```
sum(<r,r_v> in R) { jkR(r)->
 f(r)*(sum(<s,s_v> in S)
  if(jkR(r)==jkS(s)) then
   g(s) ) }
```
⇝
```
let Sp = sum(<s,s_v> in S)
 { jkS(s) -> g(s) } in
sum(<r,r_v> in R)
 { jkR(r)->f(r)*Sp(jkR(r)) }
```

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

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

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(jkR(r))`. 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 and 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 `jkR(r)` which are dependent on `r` and thus makes it impossible to be hoisted. The terms `jkR(r)` 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 and Neumann 2011]. In essence, the loop memoization and loop hoisting optimizations have the effect of *pushing aggregations past joins* [Yan and Larson 1994].

Table 2. The features of SDQL leveraged by each transformation.

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

5.3.3 Memoization Beyond Databases. In the case of using max-product semi-ring (cf. Figure 1) these optimization can *synthesize variable elimination* for maximum a priority (MAP) inference in Bayesian networks [Abo Khamis et al. 2016; Aji and McEliece 2000]. Furthermore, *loop normalization* [Shaikhha et al. 2019] can also be thought of as a special case of this rule.

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 and 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.

Table 2 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 and 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].

Consider a relation $R(a_1, ..., a_n)$ (with bag semantics), the representation of which is a dictionary of type `{ <`a_1`:`A_1`, ..., `a_n`:`A_n`> -> int }` 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 $\{A_1\texttt{->}\{...\texttt{->}\{A_n\texttt{->int}\}...\}\}$.

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 } }`.

$[\![M_1 \times M_2]\!]$ = `sum(row in` $[\![M_1]\!]$ `) { row.key ->`
`sum(x in row.val) sum(y in` $[\![M_2]\!]$`(x.key)) { y.key -> x.val * y.val } }`

Fig. 11. Translation of matrix-matrix multiplication for curried matrices to SDQL.

Example 8 (Cont.). Consider matrix M from Example 8. The curried representation of this matrix in SDQL is `{ 0 -> { 0 ->` c_0`, 3 ->` c_1 `}, 1 -> { 1 ->` c_2 `} }`.
The flat encoding of matrices presented in Section 4.2 results in inefficient implementation for various matrix operations, as explained before. By using a curried representation instead, one can provide more efficient implementations for matrix operations.

As an example, Figure 11 shows the translation of 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 $O(mnk)$.

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} }
```

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 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`>|]`.

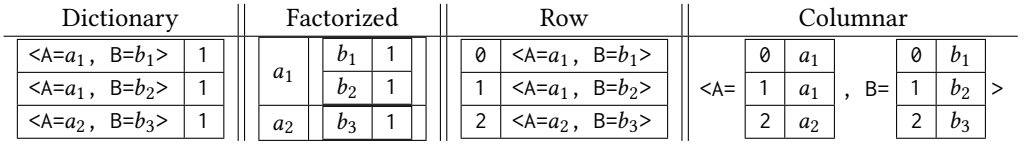


Fig. 12. Different data layouts for relations.

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 $\langle a_1 : \texttt{[|}A_1\texttt{|]}, \ \ldots, \ a_n : \texttt{[|}A_n\texttt{|]} \rangle$ in SDQL.

7 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. The operational semantics and type safety proofs can be found in the supplementary materials.

7.1 Typing

SDQL satisfies the following essential typing properties.

Lemma 7.1. *Let* $\mathbf{T}$ *denote the set of all types of SDQL.* $\otimes$ *is a well-defined partial operation* $\mathbf{T} \times \mathbf{T} \rightarrow \mathbf{T}$.

Proposition 7.2. *Every type/term defined using the 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. □

7.2 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 13. 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.

$$\llbracket \texttt{S} \rrbracket \triangleq (S, +, 0) \qquad \llbracket \texttt{<a1:T1, ..., an:Tn>} \rrbracket \triangleq \llbracket \texttt{T1} \rrbracket \times \ldots \times \llbracket \texttt{Tn} \rrbracket$$

$$\llbracket \texttt{T1} \otimes_S \texttt{T2} \rrbracket \triangleq \llbracket \texttt{T1} \rrbracket \otimes_S \llbracket \texttt{T2} \rrbracket \qquad \llbracket \texttt{\{T1 -> T2\}} \rrbracket \triangleq \bigoplus_{a \in |T1|} \llbracket \texttt{T2} \rrbracket$$

$$\llbracket \texttt{x} \rrbracket_\gamma \triangleq \gamma(\texttt{x}) \qquad \llbracket \texttt{<a1=e1,...,an=en>} \rrbracket_\gamma \triangleq \texttt{<}\llbracket \texttt{e1} \rrbracket_\gamma, \ldots, \llbracket \texttt{en} \rrbracket_\gamma\texttt{>}$$

$$\llbracket \texttt{c} \rrbracket_\gamma \triangleq \texttt{c} \qquad \llbracket \texttt{let x = e1 in e2} \rrbracket_\gamma \triangleq \llbracket \texttt{e2} \rrbracket_{\gamma[\llbracket \texttt{e1} \rrbracket_\gamma/\texttt{x}]}$$

$$\llbracket \texttt{true} \rrbracket_\gamma \triangleq 1 \qquad \llbracket \texttt{promote}_{\texttt{S1,S2}}\texttt{(e)} \rrbracket_\gamma \triangleq \mathit{Prom}_{\texttt{S1}\to\texttt{S2}}(\llbracket \texttt{e} \rrbracket_\gamma)$$

$$\llbracket \texttt{false} \rrbracket_\gamma \triangleq 0 \qquad \llbracket \texttt{if e1 then e2 else e3} \rrbracket_\gamma \triangleq \llbracket \texttt{e1} \rrbracket_\gamma * \llbracket \texttt{e2} \rrbracket_\gamma + (1 - \llbracket \texttt{e1} \rrbracket_\gamma) * \llbracket \texttt{e3} \rrbracket_\gamma$$

$$\llbracket \texttt{not(e)} \rrbracket_\gamma \triangleq 1 - \llbracket \texttt{e} \rrbracket_\gamma \qquad \llbracket \texttt{e1(e2)} \rrbracket_\gamma \triangleq \pi_{\llbracket \texttt{e2} \rrbracket_\gamma}(\llbracket \texttt{e1} \rrbracket_\gamma)$$

$$\llbracket \texttt{e.ai} \rrbracket_\gamma \triangleq \pi_i(\llbracket \texttt{e} \rrbracket_\gamma) \qquad \llbracket \texttt{\{\}}_{\texttt{T1,T2}} \rrbracket_\gamma \triangleq 0_{\texttt{\{T1->T2\}}}$$

$$\llbracket \texttt{op(e)} \rrbracket_\gamma \triangleq \texttt{op}(\llbracket \texttt{e} \rrbracket_\gamma) \qquad \llbracket \texttt{\{k1->v1,...,kn->vn\}} \rrbracket_\gamma \triangleq \sum_{i \in [1..n]} \llbracket \texttt{vi} \rrbracket_\gamma \bullet \llbracket \texttt{ki} \rrbracket_\gamma$$

$$\llbracket \texttt{e1 + e2} \rrbracket_\gamma \triangleq \llbracket \texttt{e1} \rrbracket_\gamma + \llbracket \texttt{e2} \rrbracket_\gamma$$

$$\llbracket \texttt{e1 * e2} \rrbracket_\gamma \triangleq \llbracket \texttt{e1} \rrbracket_\gamma * \llbracket \texttt{e2} \rrbracket_\gamma$$

$$\llbracket \texttt{sum(x in e1) e2} \rrbracket_\gamma \triangleq \sum_{k \in X} \llbracket \texttt{e2} \rrbracket_{\gamma[\texttt{<k,}a_k\texttt{>/x}]} \qquad (\llbracket \texttt{e1} \rrbracket_\gamma \triangleq \sum_{k \in X} a_k \bullet k)$$

Fig. 13. Denotational Semantics for types and terms of SDQL.

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

$$\llbracket \Gamma \rrbracket = \llbracket \texttt{T1} \rrbracket \times \ldots \times \llbracket \texttt{Tn} \rrbracket$$

A term $\llbracket \Gamma \vdash \texttt{e: T} \rrbracket$ is interpreted as a function from $\llbracket \Gamma \rrbracket$ to $\llbracket \texttt{T} \rrbracket$. When it is clear from the context, we use $\llbracket \texttt{e} \rrbracket$ instead of $\llbracket \Gamma \vdash \texttt{e: T} \rrbracket$. We use the notation v•k to mean the vector whose only non-zero component v is at position k in $\bigoplus_{a \in |T1|} \llbracket \texttt{T2} \rrbracket$. We denote by γ any assignment of the variables of a context Γ. The denotational semantics for terms is shown in Figure 13. $\mathit{Prom}_{\texttt{S1}\to\texttt{S2}}$ 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 7.3 (Substitution lemma). *For all* $\Gamma \vdash$ e1: T1 *and* Γ, x: T1 $\vdash$ e2: T2, *the following holds:* $\llbracket \texttt{e2} \rrbracket[\llbracket \texttt{e1} \rrbracket/\texttt{x}] = \llbracket \texttt{e2[e1/x]} \rrbracket$.

Theorem 7.4 (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* $\llbracket \texttt{e} \rrbracket = \llbracket \texttt{v} \rrbracket$.

Proof sketch. For both Proposition 7.3 and Theorem 7.4, the proof is by induction on the structure of terms and case analysis on the structure of terms in the first case, and on the last rule used of the operational semantics in the other case. The only non-standard cases are the ones involving a dictionary or sum. More details can be found in the supplementary materials. □

7.3 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 7.5. *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 7.6. *The loop factorization rules of Figure 8 are sound.*

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

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

□

The correctness proofs of the remaining optimizations, horizontal fusion, loop-invariant code motion, and loop memoization, based on both operational and denotational arguments can be found in the supplementary materials.

8 IMPLEMENTATION

SDQL is implemented as an external domain-specific language. The entire compiler tool-chain 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++.

8.1 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 there is no need for GC). The key challenging construct is **sum** which is translated into `for`-loops. Furthermore, for the case of summations that produce dictionaries, the generated loop performs destructive updates to the collection, to improve the performance [Henriksen et al. 2017].

8.2 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. Finally, for implementing records, SDQL uses `std::tuple`.

[3]https://github.com/greg7mdp/parallel-hashmap

SDQL[ring]		
`-(-e)`	$\leadsto$	`e`
`e + (-e)`	$\leadsto$	`0`
SDQL[closure]		
`1 + e * closure(e)`	$\leadsto$	`closure(e)`
`1 + closure(e) * e`	$\leadsto$	`closure(e)`
SDQL[prod]		
`(prod(x in e1) f1(x)) * (prod(x in e1) f2(x))`	$\leadsto$	`prod(x in e1) f1(x) * f2(x)`
SDQL[rec]		
`rec(x => let y=e1 in f(x,y))(e2)`	$\leadsto$	`let y=e1 in rec(x => f(x,y))(e2)`

Fig. 14. Additional transformation rules for language extensions of SDQL.

8.3 Semi-ring Extensions

Scalar Semi-rings. Throughout the paper, 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 and Parreaux 2019]. SDQL can support such applications by including additional scalar semi-rings, a subset of which are presented 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 and 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 9.4).

8.4 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 14. 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 14. 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

and 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 14).

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 14).

9 EXPERIMENTAL RESULTS

9.1 Experimental Setup

We run our experiments on a 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 1000.10.44.4 for compiling the generated C++ code using the O3 flag. Our competitor systems use Scala 2.12.2, Spark 3.0.1, 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.[4] We measure the average run time execution of five runs excluding the loading time.

9.2 Database Workloads

In this section, we investigate the performance of SDQL for online analytical processing (OLAP) workloads used in the databases. For this purpose, we compare the performance of generated optimized code for the dictionary layout, row layout, and columnar layout of SDQL with the open source implementation[5] [Kersten et al. 2018] of two state-of-the-art analytical query processing engines: 1) Typer for HyPer [Neumann 2011], and 2) Tectorwise for Vectorwise [Zukowski et al. 2005].

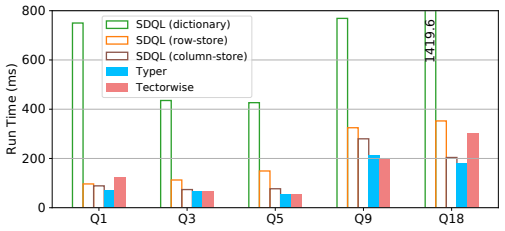


Fig. 15. Run time results for TPCH queries comparing different data layouts in SDQL, Typer, and Tectorwise.

For these experiments, we use TPCH, the main benchmark for such workloads in databases. Instead of running all 22 TPCH queries, we only use a representative subset of them for the following reasons. First, previous research [Boncz et al. 2014; Kersten et al. 2018] identified that this subset has the "choke points" of all TPCH queries. Second, the open source implementations of Typer and

[4]Prior work on parallelism for database query engines [Graefe 1994], nested data processing (flattening and shredding [Smith et al. 2020]), and sparse linear algebra [Kjolstad et al. 2017] can be transferred to SDQL, which we leave as future work.
[5]https://github.com/TimoKersten/db-engine-paradigms

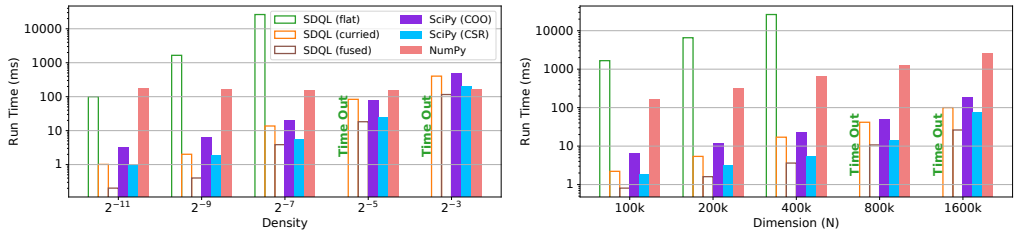


Fig. 16. Run time results for computing the covariance matrix comparing different optimizations and representations in SDQL, SciPy, and NumPy. The dimension for the input matrix of the left figure is 100000×100, and the dimension of the input matrix of the right figure is $N \times 100$ with the density of 2^{-7}.

Tectorwise only support this subset. We further restricted this subset to the queries that construct intermediate dictionaries; we excluded Q6 as it does not have any joins or group-by aggregates.

Figure 15 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. The columnar layout leads to performance on par with Tectorwise, but SDQL remains about 20% slower than Typer. The performance can be further improved by better memory management and string processing techniques, as used in Typer and Tectorwise.

9.3 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. SciPy does not support higher-order tensors, and it was shown before [Chou et al. 2018; Kjolstad et al. 2017] that on these workloads, taco is faster than systems such as SPLATT [Smith et al. 2015], Tensor Toolbox [Bader and Kolda 2008], and TensorFlow [Abadi et al. 2016]. For a fair comparison, we have included the time for assembling the output tensor in taco.

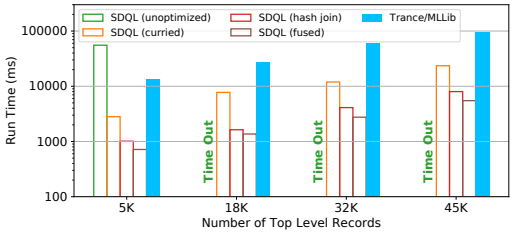
Sparse Matrix Processing. First, we consider the task of computing the covariance matrix $X^T X$ (cf. Section 4), where X is a synthetically generated input data matrix of varying dimensions and density. We consider the following different versions of the generated code from SDQL: 1) unoptimized, which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

As Figure 16 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 2× 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 two orders of magnitude faster. Finally, the most optimized version of the generated code by SDQL is in average 3× and 2× faster than the COO and CSR represenations of SciPy, respectively, thanks to fusion and the efficient low-level code generated by SDQL.

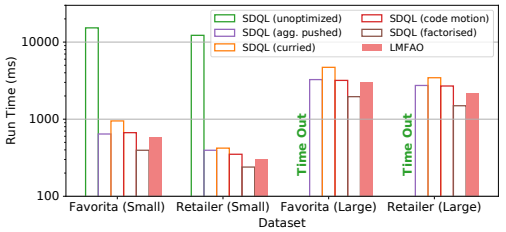
Sparse Tensor Processing. Next, we consider three higher-order tensor workloads on NELL-2, a real world dataset coming from the Never Ending Language-Learning project [Carlson et al. 2010]. Table 3 shows the performance comparison for these workloads. We observe that especially for a medium range of sparsity SDQL is faster than taco (from 1.4× to 23×). For sparser scenarios,

Table 3. 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.

	Sparsity	2^{-11}		2^{-9}		2^{-7}		2^{-5}		2^{-3}	
Kernel	LA Formulation	SDQL	taco	SDQL	taco	SDQL	taco	SDQL	taco	SDQL	taco
TTV	$A_{ij} = \Sigma_k B_{ijk} c_k$	621.8	466.3	621.8	544.9	632.0	866.2	661.8	2088.1	729.4	6742.7
TTM	$A_{ijk} = \Sigma_k B_{ijl} C_{kl}$	4534.2	5936.2	4679.6	7851.6	4764.2	15563.9	5189.2	46153.7	7146.6	169865.5
MTTKRP	$A_{ij} = \Sigma_{k,l} B_{ikl} C_{kj} D_{lj}$	5.6	4.3	18.4	17.3	32.2	60.4	103.2	388.1	723.8	4371.1



(a) Biomedical query with different optimizations in SDQL and Trance [Smith et al. 2020]/MLLib.



(b) Retail forecasting using different optimizations in SDQL and LMFAO [Schleich et al. 2019].

Fig. 17. Run time results for computing covariance matrix over nested and relational data.

taco shows better performance (up to 1.3×), 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].

9.4 Hybrid LA/DB Workload

As the final set of experiments, we consider hybrid workloads that involve linear algebra and query processing. Figure 17 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: 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 17a 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 8.3). We use two real-world

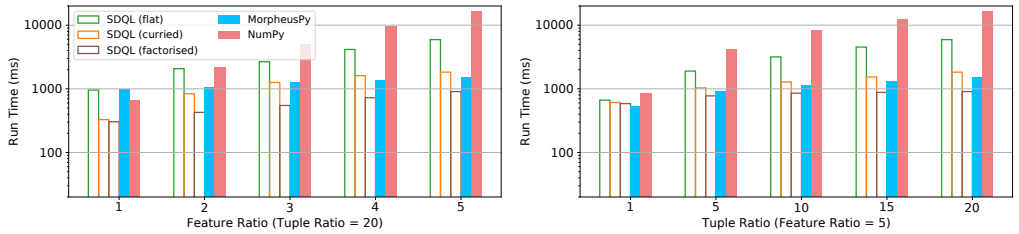


Fig. 18. Run time of SDQL, MorpheusPy, and NumPy for computing the covariance matrix over normalized matrix. For both plots, S has two features ($d_S = 2$) and R contains one million tuples ($n_R = 1M$). In the left figure, $n_S = 20M$ and $d_R \in \{2, 4, 6, 8, 10\}$. In the right figure, $d_R = 10$ and $n_S \in \{1M, 5M, 10M, 15M, 20M\}$.

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 17b 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 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 2019a], 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 2019b], i.e., $R \bowtie S$. Figure 18 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 18 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, thanks to the vectorization offered by NumPy. Finally, we observe a superior performance for SDQL once the curried representation is used. As the tuple ratio increases, the speed up of SDQL over MorpheusPy climbs up to 1.7×, thanks to the loop factorization enabled by the curried representation for relation S, which is not available for MorpheusPy/NumPy.

10 RELATED WORK

In this section, we review the literature. Table 4 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 [Armbrust et al. 2015; Crotty et al. 2015; Karpathiotakis et al. 2015; Koch et al. 2014; Krikellas et al. 2010; Nagel et al. 2014; Neumann 2011; Palkar et al. 2017; Shaikhha et al. 2018b, 2016; Tahboub et al. 2018; Viglas et al. 2014]. 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.

Nested Data Models. Nested relational model [Roth et al. 1988] and monad calculus [Breazu-Tannen et al. 1992; Breazu-Tannen and Subrahmanyam 1991; Buneman et al. 1995; Grust and Scholl 1999; Trinder 1992; Wadler 1990] support complex data models but do not support aggregations and efficient equi-joins [Gibbons et al. 2018]. Monoid comprehensions solve the former issue [Fegaras and Maier 2000], however, require an intermediate algebra to support equi-joins efficiently. Kleisli [Wong 2000], BQL [Libkin and 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 [Cheney et al. 2014; Grust et al. 2010]. Comprehensive Comprehensions (CompComp) [Jones and 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 [Coutts et al. 2007; Emoto et al. 2012; Gill et al. 1993; Svenningsson 2002; Takano and Meijer 1995; Wadler 1988] 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 and Stucki 2015; Kiselyov et al. 2017; Shaikhha et al. 2018a]. Generalized stream fusion [Mainland et al. 2013] combines deforestation with vectorization for Haskell. Functional array processing languages such as APL [Iverson 1962], SAC [Grelck and Scholz 2006], Futhark [Henriksen et al. 2017], and $\widetilde{\mathrm{F}}$ [Shaikhha et al. 2019] also need to support loop fusion. Such languages mainly use pull and push arrays [Anker and Svenningsson 2013; Axelsson et al. 2011; Claessen et al. 2012; Kiselyov 2018; Shaikhha et al. 2017; Svensson and Svenningsson 2014] 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 et al. 2018; Spampinato and Püschel 2016] exploit the memory hierarchy and make careful 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

Table 4. 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 Representation					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 Paper)	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○
Query Compilers (HyPer)	●	○	●	●	○	●	●	○	●	○	◐	◐	○	●	○
Vectorized Query Engines (Vectorwise)	●	○	●	●	○	●	●	○	●	○	◐	◐	○	○	●
Monad Calculus, NRC$^+$	●	●	○	○	○	●	○	○	○	○	◐	◐	○	○	○
Monoid Comprehension	●	●	●	○	○	●	○	○	○	○	◐	◐	○	○	○
Monad Calc. + Agg. (Kleisli, Trance)	●	●	●	○	◐	●	○	○	○	○	◐	◐	○	●	○
Lang. Integrated Queries (LINQ, CompComp)	●	●	●	○	●	●	○	○	○	○	◐	◐	○	○	○
Functional Lists (Generalized 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)	●	○	●	●	●	●	●	◐	○	○	◐	◐	◐	●	●

operations on dense and sparse matrices. 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 and Gilbert 2011]. This connection has been used for expressing path problems by solving matrix equations [Backhouse and Carré 1975; Tarjan 1981; Valiant 1975]. SDQL requires extensions in order to express such problems (cf. Section 8.4). 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 [Dolan 2013; Shaikhha and Parreaux 2019]. In the DB world, K-relations [Green et al. 2007] use semi-rings [Karvounarakis and 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. 2019a] 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. 2019b]. 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. 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 and 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 9.4). 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 9.4). The second category are systems such as F [Olteanu and 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. For implementing cross-domain optimizations, this approach 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, 2021] 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 4). 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.

11 CONCLUSION

In this paper, we introduce a statically typed and functional language based on semi-ring dictionaries. SDQL is expressive enough for different data science use-cases with a better or competitive performance relative to specialized systems. For example, the performance of SDQL is competitive with the state-of-the-art in-memory DB systems that are especially built for DB workloads, and thus cannot efficiently handle other use-cases including sparse LA, and in-DB ML over different formats of data: nested, relational, and normalized matrix. This makes SDQL a suitable intermediate language for data science pipelines typically expressed in several languages and executed using different systems. For future, we plan to add the support for vectorization and parallelization.

ACKNOWLEDGEMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 682588. The authors also acknowledge the EPSRC grant EP/T022124/1 (QUINTON) and Huawei for their support of the distributed data management and processing laboratory at the University of Edinburgh.

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. In-Database Learning with Sparse Tensors. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Houston, TX, USA) *(SIGMOD/PODS '18)*. Association for Computing Machinery, New York, NY, USA, 325–340.

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

Srinivas M Aji and Robert J McEliece. 2000. The generalized distributive law. *IEEE transactions on Information Theory* 46, 2 (2000), 325–343.

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

Johan Anker and Josef Svenningsson. 2013. An EDSL approach to high performance Haskell programming. In *ACM Haskell Symposium*. ACM, New York, NY, USA, 1–12.

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

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

R. C. Backhouse and B. A. Carré. 1975. Regular Algebra Applied to Path-finding Problems. *IMA Journal of Applied Mathematics* 15, 2 (04 1975), 161–186.

Brett W Bader and Tamara G Kolda. 2008. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing* 30, 1 (2008), 205–231.

Peter Boncz, Thomas Neumann, and Orri Erling. 2014. *TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark*. Springer International Publishing, Cham, 61–76.

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

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

Robert Brijder, Floris Geerts, Jan Van Den Bussche, and Timmy Weerwag. 2019a. On the Expressive Power of Query Languages for Matrices. *ACM Trans. Database Syst.* 44, 4, Article 15 (oct 2019), 31 pages.

Robert Brijder, Floris Geerts, Jan Van Den Bussche, and Timmy Weerwag. 2019b. On the Expressive Power of Query Languages for Matrices. *ACM Trans. Database Syst.* 44, 4, Article 15 (oct 2019), 31 pages.

Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. 1995. Principles of Programming with Complex Objects and Collection Types. *Theor. Comput. Sci.* 149, 1 (Sept. 1995), 3–48.

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

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

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

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

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

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

Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: Proving query rewrites with univalent SQL semantics. *ACM SIGPLAN Notices* 52, 6 (2017), 510–524.

Koen Claessen, Mary Sheeran, and Bo Joel Svensson. 2012. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In *Proceedings of the 7th Workshop on Declarative Aspects and Applications of Multicore Programming (DAMP '12)*. ACM, NY, USA, 21–30.

E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (June 1970), 377–387.

National Research Council (US) Committee. 2005. On the Nature of Biological Data. In *Catalyzing Inquiry at the Interface of Computing and Biology*, John C. Wooley and Herbert S. Lin (Eds.). National Academies Press (US), Chapter 3.

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

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web Programming Without Tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects* (Amsterdam, The Netherlands) *(FMCO'06)*. Springer-Verlag, Berlin, Heidelberg, 266–296.

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

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion. From Lists to Streams to Nothing at All. In *ICFP '07*.

Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B Zdonik. 2015. Tupleware:" Big" Data, Big Analytics, Small Clusters.. In *CIDR*.

Stephen Dolan. 2013. Fun with Semirings: A Functional Pearl on the Abuse of Linear Algebra. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 101–110.

Kento Emoto, Sebastian Fischer, and Zhenjiang Hu. 2012. Filter-embedding semiring fusion for programming with MapReduce. *Formal Aspects of Computing* 24, 4 (2012), 623–645.

Laura Fancello, Sara Gandini, Pier Giuseppe Pelicci, and Luca Mazzarella. 2019. Tumor mutational burden quantification from targeted gene panels: major advancements and challenges. *Journal for ImmunoTherapy of Cancer* 7, 1 (2019), 183. https://doi.org/10.1186/s40425-019-0647-4

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

Leonidas Fegaras and David Maier. 2000. Optimizing Object Queries Using an Effective Calculus. *ACM Trans. Database Syst.* 25, 4 (Dec. 2000), 457–516.

Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. 2012. Towards a Unified Architecture for in-RDBMS Analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. ACM, New York, NY, USA, 325–336.

Robert Fink, Larisa Han, and Dan Olteanu. 2012. Aggregation in Probabilistic Databases via Knowledge Compilation. 5, 5 (jan 2012), 490–501.

Floris Geerts, Thomas Muñoz, Cristian Riveros, Jan Van den Bussche, and Domagoj Vrgoč. 2021. Matrix Query Languages. *ACM SIGMOD Record* 50, 3 (2021), 6–19.

Jeremy Gibbons, Fritz Henglein, Ralf Hinze, and Nicolas Wu. 2018. Relational Algebra by Way of Adjunctions. *Proc. ACM Program. Lang.* 2, ICFP, Article 86 (July 2018), 28 pages. https://doi.org/10.1145/3236781

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

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

G. Graefe. 1994. Volcano-an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering* 6, 1 (1994), 120–135.

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

Clemens Grelck and Sven-Bodo Scholz. 2006. SAC—A Functional Array Language for Efficient Multi-threaded Execution. *Int. Journal of Parallel Programming* 34, 4 (2006), 383–427.

Torsten Grust, Jan Rittinger, and Tom Schreiber. 2010. Avalanche-safe LINQ Compilation. *PVLDB* 3, 1-2 (Sept. 2010), 162–172.

Torsten Grust and MarcH. Scholl. 1999. How to Comprehend Queries Functionally. *Journal of Intelligent Information Systems* 12, 2-3 (1999), 191–218.

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

Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference*

on Programming Language Design and Implementation. ACM, 556–571.

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

S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. 2012. Monetdb: Two decades of research in column-oriented database. *IEEE Data Engineering Bulletin* (2012).

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

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

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell workshop*, Vol. 1. 203–233.

Simon Peyton Jones and Philip Wadler. 2007. Comprehensive comprehensions. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*. 61–72.

Manohar Jonnalagedda and Sandro Stucki. 2015. Fold-based Fusion As a Library: A Generative Programming Pearl. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala* (Portland, OR, USA). ACM, 41–50.

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

Manos Karpathiotakis, Ioannis Alagiannis, Thomas Heinis, Miguel Branco, and Anastasia Ailamaki. 2015. Just-in-time data virtualization: Lightweight data management with ViDa. In *CIDR*.

Grigoris Karvounarakis and Todd J Green. 2012. Semiring-annotated data: queries and provenance? *ACM SIGMOD Record* 41, 3 (2012), 5–14.

Gabriele Keller, Manuel MT Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. 2010. Regular, shape-polymorphic, parallel arrays in Haskell. *ACM Sigplan Notices* 45, 9 (2010), 261–272.

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

Jeremy Kepner and John Gilbert. 2011. *Graph algorithms in the language of linear algebra*. Vol. 22. SIAM.

Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proceedings of the VLDB Endowment* 11, 13 (2018), 2209–2222.

Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. AC/DC: In-Database Learning Thunderstruck. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning* (Houston, TX, USA) *(DEEM'18)*. ACM, New York, NY, USA, Article 8, 10 pages.

Oleg Kiselyov. 2018. Reconciling Abstraction with High Performance: A MetaOCaml approach. *Foundations and Trends in Programming Languages* 5, 1 (2018), 1–101.

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream Fusion, to Completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. ACM, New York, NY, USA, 285–299.

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages.

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

Christoph Koch, Daniel Lupei, and Val Tannen. 2016. Incremental View Maintenance For Collection Programming. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (San Francisco, California, USA) *(PODS '16)*. Association for Computing Machinery, New York, NY, USA, 75–90.

Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *ICDE*. 613–624.

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

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

Side Li, Lingjiao Chen, and Arun Kumar. 2019. Enabling and Optimizing Non-linear Feature Interactions in Factorized Linear Algebra. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, 1571–1588.

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

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

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

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

Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.

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

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

Fabian Nagel, Gavin Bierman, and Stratis D. Viglas. 2014. Code Generation for Efficient Query Processing in Managed Runtimes. *PVLDB* 7, 12 (Aug. 2014), 1095–1106.

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

Milos Nikolic and Dan Olteanu. 2018. Incremental View Maintenance with Triple Lock Factorization Benefits. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. ACM, New York, NY, USA, 365–380.

Dan Olteanu and Maximilian Schleich. 2016. Factorized Databases. *SIGMOD Rec.* 45, 2 (Sept. 2016), 5–16.

Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. 2017. Weld: A Common Runtime for High Performance Data Analytics. In *Conference on Innovative Data Systems Research (CIDR)*.

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

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

Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* 93, 2 (2005), 232–275.

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

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI'13)*. ACM, New York, NY, USA, 519–530.

Raghu Ramakrishnan and Johannes Gehrke. 2000. *Database Management Systems* (2nd ed.). Osborne/McGraw-Hill.

Mark A Roth, Herry F Korth, and Abraham Silberschatz. 1988. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems (TODS)* 13, 4 (1988), 389–417.

Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. 2019. A Layered Aggregate Engine for Analytics Workloads. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. ACM, New York, NY, USA, 1642–1659.

Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning Linear Regression Models over Factorized Joins. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 3–18.

Amir Shaikhha, Mohammad Dashti, and Christoph Koch. 2018a. Push versus Pull-Based Loop Fusion in Query Engines. *Journal of Functional Programming* 28 (2018), e10.

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

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

Amir Shaikhha, Yannis Klonatos, and Christoph Koch. 2018b. Building Efficient Query Engines in a High-Level Language. *ACM Transactions on Database Systems* 43, 1, Article 4 (April 2018), 45 pages.

Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to Architect a Query Compiler. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD'16)*. ACM, New York, NY, USA, 1907–1922.

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

Amir Shaikhha, Maximilian Schleich, Alexandru Ghita, and Dan Olteanu. 2020. Multi-Layer Optimizations for End-to-End Data Analytics. In *CGO*. 145–157.

Amir Shaikhha, Maximilian Schleich, and Dan Olteanu. 2021. An Intermediate Representation for Hybrid Database and Machine Learning Workloads. *Proc. VLDB Endow.* 14, 12 (2021), 2831–2834.

Arun Kumar Side Li. 2019a. MorpheusPy. https://github.com/ADALabUCSD/MorpheusPy.

Arun Kumar Side Li. 2019b. MorpheusPy – Issue #3. https://github.com/ADALabUCSD/MorpheusPy/issues/3.

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

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

Daniele G. Spampinato, Diego Fabregat-Traver, Paolo Bientinesi, and Markus Püschel. 2018. Program Generation for Small-scale Linear Algebra Applications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. ACM, New York, NY, USA, 327–339.

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

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

Peter H. Sudmant, Tobias Rausch, Eugene J. Gardner, Robert E. Handsaker, Alexej Abyzov, John Huddleston, Yan Zhang, Kai Ye, Goo Jun, Markus Hsi-Yang Fritz, et al. 2015. An integrated map of structural variation in 2,504 human genomes. *Nature* 526, 7571 (2015), 75–81. https://doi.org/10.1038/nature15394

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

Josef Svenningsson. 2002. Shortcut Fusion for Accumulating Parameters & Zip-like Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming* (Pittsburgh, PA, USA) *(ICFP '02)*. ACM, 124–132.

Bo Joel Svensson and Josef Svenningsson. 2014. Defunctionalizing Push Arrays. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing* (Gothenburg, Sweden) *(FHPC '14)*. ACM, NY, USA, 43–52.

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

Akihiko Takano and Erik Meijer. 1995. Shortcut Deforestation in Calculational Form. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture* (La Jolla, California, USA) *(FPCA '95)*. Association for Computing Machinery, New York, NY, USA, 306–313.

Robert Endre Tarjan. 1981. A Unified Approach to Path Problems. *J. ACM* 28, 3 (jul 1981), 577–593.

Hail Team. 2020. Hail 0.2. https://github.com/hail-is/hail.

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

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

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

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

Stratis Viglas, Gavin M. Bierman, and Fabian Nagel. 2014. Processing Declarative Queries Through Generating Imperative Code in Managed Runtimes. *IEEE Data Eng. Bull.* 37, 1 (2014), 12–21.

Kate Voss, Jeff Gentry, and Geraldine Van Der Auwera. 2017. Full-stack genomics pipelining with GATK4+ WDL+ Cromwell [version 1; not peer reviewed]. *F1000Research* (2017), 4. https://doi.org/10.7490/f1000research.1114631.1

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

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

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

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

Weipeng P. Yan and Per-Åke Larson. 1994. Performing Group-By before Join. In *Proceedings of the Tenth International Conference on Data Engineering*. IEEE Computer Society, USA, 89–100.

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

Marcin Zukowski, Peter A Boncz, Niels Nes, and Sándor Héman. 2005. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Eng. Bull.* 28, 2 (2005), 17–22.