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.

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. 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:

\[ Q(a, d) = \Gamma^a_{a,d} R_1(a, b) \Join R_2(b, c) \Join R_3(c, d) \]
\[ N(i, l) = \Sigma_{j,k} M_1(i, j) \cdot M_2(j, k) \cdot M_3(k, l) \]

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:

\[ Q'(a, c) = \Gamma^a_{a,c} R_1(a, b) \Join R_2(b, c) \]
\[ N'(i, k) = \Sigma_{j,k} M_1(i, j) \cdot M_2(j, k) \cdot M_3(k, l) \]

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.
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.

1.1 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.

• 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 LMF AO and Morpheus.

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

---

1In this paper, by (nested) relational and linear algebra, we mean the corresponding sets of operators presented in Figures 4-7.
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; 0S is the identity element for + and 1S is for *. When clear from the context, we use 0 and 1 as identity elements. Furthermore, the following algebraic laws hold for all elements a, b, and c:

- a + (b + c) = (a + b) + c
- 0 + a = a + 0 = a
- 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:

$$a \ast (u + v) = a \ast u + a \ast v$$
$$a \ast (u + v) = a \ast u + a \ast v = a$$

(a + b) \ast u = a \ast u + b \ast u

\begin{itemize}
  \item \textbf{Tensor product.} For two types $T_1$ and $T_2$ that are S-semi-modules, the tensor product $T_1 \otimes T_2$ is another S-semi-module. It comes equipped with a canonical map which we also denote using $\ast$: $T_1 \times T_2 \rightarrow T_1 \otimes T_2$ with the following laws for all elements $u_1,u_2:T_1$ and $v_1,v_2:T_2$:

$$u_1 \ast (v_1 + v_2) = u_1 \ast v_1 + u_1 \ast v_2$$

$$u_1 \ast u_2 \ast v_1 = u_1 \ast v_1 + u_2 \ast v_1$$

$$1 \ast u_1 = u_1$$
\end{itemize}

\section{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 $\text{SM}(S)$; semi-ring dictionaries with S-semi-module value types are also S-semi-modules (i.e., they have the kind $\text{SM}(S)$). However, dictionaries with value types of the ordinary kind $\text{Type}$ are of kind $\text{Type}$. Similar patterns apply to records.

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

The addition of two expressions requires both operands to have the same type of kind $\text{SM}(S)$. This means that the body of summation also needs to have a type of kind $\text{SM}(S)$. The type system rules for the multiplication operator are defined inductively. Multiplying a scalar with a dictionary results in a dictionary with the same keys, but with the values multiplied with the scalar value. Multiplying a dictionary with another term also results in a dictionary with the same keys, and values multiplied with that term. Note that the multiplication operator is not commutative in general.\footnote{To be more precise, the scalar $\ast$ is commutative, but the tensor product $\ast$ is commutative up to reordering.}

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

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{kind-system-type-system-sdql.png}
\caption{Kind System and Type System of SDQL.}
\end{figure}
### 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  | $\mathbb{N}$ | $+$      | $\times$      | $0$  | $1$ | X    |
| Min-Product         | mnpr | (0, ∞)  | min      | $\times$      | $\infty$ | $1$ | X    |
| Max-Product         | mxpr | (0, ∞)  | max      | $\times$      | $0$  | $1$ | X    |
| Min-Sum             | mns  | (−∞, ∞) | min      | +              | $\infty$ | $0$ | X    |
| Max-Sum             | mxs  | (−∞, ∞) | max      | +              | $−\infty$ | $0$ | X    |
| Max-Min             | mxm  | (−∞, ∞) | max      | min            | $−\infty$ | $+\infty$ | X |
| Boolean             | bool | $\{T, F\}$ | $\lor$   | $\land$       | $false$ | $true$ | X    |

#### 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 $S_1$ to type $S_2$ is achieved by $promote_{S_1,S_2}(s)$.

**Dictionaries.** A dictionary with keys of type $K$, and values of type $V$ is represented by the data type $\{K \rightarrow V\}$. The expression $\{k_1 \rightarrow v_1, \ldots, k_n \rightarrow v_n\}$, constructs a dictionary of $n$ elements with keys $k_1, \ldots, k_n$ and values $v_1, \ldots, v_n$. The expression $\{k, v\}$ constructs an empty dictionary of type $\{ K \rightarrow V \}$, 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 $0_V$ 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 $0_V$-valued $k_j$.

The multiplication $dict \times s$, 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 $v_i \times s$ as the values. For the expression $s \times dict$, 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 $s \times v_i$ as values. Note that the multiplication operator is not commutative by default.

**Example 2.** Consider the following two SDs: $\{"a"\rightarrow2, "b"\rightarrow3\}$ named as dict1 and $\{"a"\rightarrow4, "c"\rightarrow5\}$ named as dict2. The result of $dict1 \times dict2$ is $\{"a"\rightarrow6, "b"\rightarrow3, "c"\rightarrow5\}$. This is because $dict1$ is equivalent to $\{"a"\rightarrow2, "b"\rightarrow3, "c"\rightarrow0\}$ and $dict2$ is equivalent to $\{"a"\rightarrow4, "b"\rightarrow0, "c"\rightarrow5\}$, and element-wise addition of them results in $\{"a"\rightarrow2+4, "b"\rightarrow3+0, "c"\rightarrow0+5\}$.

The result of $dict1 \times dict2$ is $\{"a"\rightarrow2+4 \times dict2, "b"\rightarrow3+2 \times dict2\}$. The expression $2 \times dict2$ is evaluated to $\{"a"\rightarrow2\times4, "c"\rightarrow2\times5\}$. By performing similar computations, dict1 $\times$ dict2 is evaluated to $\{"a"\rightarrow(\text{a}+\text{b}+\text{c}), "b"\rightarrow\text{a}+\text{b}+\text{c}\}$. On the other hand, dict2 $\times$ dict1 is $\{"a"\rightarrow4 \times dict1, "c"\rightarrow5 \times dict1\}$. After performing similar computations, the expression is evaluated to $\{"a"\rightarrow(\text{a}+\text{b}+\text{c}), "b"\rightarrow\text{a}+\text{b}+\text{c}\}$.

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

**Example 1 (Cont.).** Assume the dictionary $d$ with the value $\{"a"\rightarrow2, "b"\rightarrow3\}$, and the record $r$ with the value $<c=4>$. The expression $d \times r$ is evaluated as $\{\text{"a" } \rightarrow \text{<c=8>}, \text{ "b" } \rightarrow \text{<c=12>}\}$.
### Extension

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

### 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 $x.\text{key}$ specifying the key and $x.\text{val}$ specifying the value. One can alternatively use the syntactic sugar $\sum(<k,v> \in d) e$ that binds $k$ to $x.\text{key}$ and $v$ to $x.\text{val}$ (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 $\sum(x \in d) x.\text{val}$ where $d$ is a dictionary with value of $\{ "a" \rightarrow 2, "b" \rightarrow 3 \}$. This expression is evaluated to 5, which is the result of adding the values ($2 + 3$) in dictionary $d$. Let us consider the expression $\sum(<k,v> \in d) \{ k \rightarrow v * 2 \}$, with the same value as before for $d$. This expression is evaluated to $\{ "a" \rightarrow 4, "b" \rightarrow 6 \}$, which is the result of the addition of $\{ "a" \rightarrow 2*2 \}$ and $\{ "b" \rightarrow 3*2 \}$.

### 2.5 Set and Array

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

### 3 EXPRESSIVENESS FOR DATABASES

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

### 3.1 Relational Algebra

Relational algebra [Codd 1970] is the foundation of many query languages used in database management systems, including SQL. In general, a relation $R(a_1, \ldots, a_n)$ (with set semantics) is represented as a dictionary of type $\{ <a_1: A_1, \ldots, a_n: A_n> \rightarrow \text{bool} \}$ 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:

![Fig. 3. Extended constructs of SDQL.](image-url)
Fig. 4. Translation from relational algebra (with set semantics) to SDQL.

This relation is represented as follows in SDQL:

\[
\{ \langle \text{name} = \text{NOTCH2}, \text{desc} = \text{notch receptor 2}, \text{contig} = 1, \text{start} = 119911553, \text{end} = 120100779, \text{gid} = \text{ENSG00000134250} \rangle, \\
\langle \text{name} = \text{BRCA1}, \text{desc} = \text{DNA repair associate}, \text{contig} = 17, \text{start} = 43044295, \text{end} = 43170245, \text{gid} = \text{ENSG00000012048} \rangle, \\
\langle \text{name} = \text{TP53}, \text{desc} = \text{tumor protein p53}, \text{contig} = 17, \text{start} = 7565097, \text{end} = 7590856, \text{gid} = \text{ENSG00000141510} \rangle \}
\]

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:

\[
\sum(<\text{g},v> \text{ in Genes}) \{ \langle \text{gene} = \text{g.name}, \text{contig} = \text{g.contig}, \text{start} = \text{g.start}, \text{end} = \text{g.end} \rangle \}
\]

**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:

\[
\{ \langle \text{contig} = 17, \text{start} = 43093817, \text{reference} = \text{C}, \text{alternate} = \text{A}, \text{genotypes} = \{ \langle \text{sample} = \text{TCGA-AN-A046}, \text{call} = 0 \rangle \rightarrow 1, \langle \text{sample} = \text{TCGA-BH-A0B6}, \text{call} = 1 \rangle \rightarrow 1 \} \rightarrow 1, \\
\langle \text{contig} = 1, \text{start} = 119967501, \text{reference} = \text{G}, \text{alternate} = \text{C}, \text{genotypes} = \{ \langle \text{sample} = \text{TCGA-AN-A046}, \text{call} = 1 \rangle \rightarrow 1, \langle \text{sample} = \text{TCGA-BH-A0B6}, \text{call} = 2 \rangle \rightarrow 1 \} \rightarrow 1 \} \}
\]

This nested relation is represented as follows in SDQL:

\[
\{ \langle \text{contig} = 17, \text{start} = 43093817, \text{reference} = \text{C}, \text{alternate} = \text{A}, \text{genotypes} =\{ \langle \text{sample} = \text{TCGA-AN-A046}, \text{call} = 0 \rangle \rightarrow 1, \langle \text{sample} = \text{TCGA-BH-A0B6}, \text{call} = 1 \rangle \rightarrow 1 \} \rightarrow 1, \\
\langle \text{contig} = 1, \text{start} = 119967501, \text{reference} = \text{G}, \text{alternate} = \text{C}, \text{genotypes} = \{ \langle \text{sample} = \text{TCGA-AN-A046}, \text{call} = 1 \rangle \rightarrow 1, \langle \text{sample} = \text{TCGA-BH-A0B6}, \text{call} = 2 \rangle \rightarrow 1 \} \rightarrow 1 \} \}
\]
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+

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

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):

\[
\text{sum(<v,v_v> in Variants) sum(<g,g_v> in Genes)}
\]

\[
\text{if(g.contig==v.contig&&g.start<=v.start&&g.end>=v.start) then sum(<c,c_v> in v.genotypes)}
\]

\[
\{ <\text{sample = c.sample, gene = g.name, burden = c.call}> -> v_v * g_v * c_v \}
\]

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 \( \Gamma_{gf} \); where \( g \) specifies the set of keys that are partitioned by, and \( f \) specifies the aggregation function. NRC\textsuperscript{agg} is an extended version of the latter with support for two aggregation operators; \textit{sumBy}_f is similar to group-by aggregates in relational algebra, whereas \textit{groupBy}_g only performs partitioning without performing any aggregation.

Figure 6 shows the translation of aggregations in relational algebra and NRC\textsuperscript{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 semiring 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
This expression is translated as the following SDQL expression:

\[
\begin{align*}
\text{let} & \quad \text{tmp } = \text{sum}(x, x_v) \in \text{gv} \{ x.\text{sample } \rightarrow \{ x \rightarrow x_v \} \} \in \text{in} \\
\text{let} & \quad \text{gmb } = \text{sum}(x, x_v) \in \text{tmp} \{ <\text{key}, \text{val}=x_v> \rightarrow 1 \} \in \text{in} \\
\text{sum} & \quad (x, x_v) \in \text{gmb} \{ <\text{key}=x, \text{val}=x_v> \rightarrow 1 \}
\end{align*}
\]

This expression is of type \(<\text{sample: string, burdens: } <\text{key: string, val: real}> \rightarrow \text{int}> \rightarrow \text{int}>\).

**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 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 \text{S} are SDQL expressions of type \(<\text{int } \rightarrow \text{S}>\). 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 \(V = [a_0 \ 0 \ a_1 \ a_2]\) and \(U = [b_0 \ b_1 \ b_2 \ 0]\). These vectors are represented in SDQL as \(<\{ 0 \rightarrow a_0, 2 \rightarrow a_1, 3 \rightarrow a_2 \}>\) and \(<\{ 0 \rightarrow b_0, 1 \rightarrow b_1, 2 \rightarrow b_2 \}>\). The expression \(V \circ U\) is evaluated to \(<\{ 0 \rightarrow a_0*b_0, 2 \rightarrow a_1*b_2 \}>\). As the value associated with the key 3 is zero, this dictionary is equivalent to \(<\{ 0 \rightarrow a_0*b_0, 2 \rightarrow a_1*b_2 \}>\). This value corresponds to the result of evaluating \(V \circ U\), that is the vector \([a_0b_0 \ 0 \ a_1b_2 \ 0]\).
### Vector Operations:

| Name               | Translation                          | Einsum |
|--------------------|--------------------------------------|--------|
| Addition           | \[ V_1 + V_2 \] = \[ V_1 \] + \[ V_2 \] |        |
| Scal-Vec. Mul.     | \[ a \cdot V \] = \[ a \] * \[ V \]   | i->i   |
| Hadamard Prod.     | \[ \sum(x \in V_1) \} x.\text{key} \rightarrow x.\text{val} \} \[ V_2 \} x.\text{key} \} | i,i->i |
| Dot Prod.          | \[ \sum(x \in V_1) \} x.\text{val} \rightarrow \[ V_2 \} x.\text{key} \} | i,i->i |
| Summation          | \[ \sum_{a \in V} a \] = \[ \sum(x \in V) \} x.\text{val} \} | i->i   |

### Matrix Operations:

| Name               | Translation                          | Einsum |
|--------------------|--------------------------------------|--------|
| Transpose          | \[ M^T \] = \[ \sum(x \in M) \} \] \{ \text{<row=x.\text{key}.col, col=x.\text{key}.row> -> x.\text{val} } \} | ij->ji |
| Addition           | \[ M_1 + M_2 \] = \[ M_1 \] + \[ M_2 \] |        |
| Scal-Mat. Mul.     | \[ a \cdot M \} = \[ a \] * \[ M \} | i,j->ij |
| Hadamard Prod.     | \[ M_1 \odot M_2 \} = \[ \sum(x \in M_1) \} \} x.\text{key} \rightarrow x.\text{val} \rightarrow \[ M_2 \} x.\text{key} \} | ij,j->ij |
| Matrix-Matrix Mul. | \[ M_1 \times M_2 \} = \[ \sum(y \in M_2) \} \} x.\text{key} \rightarrow \[ M_1 \} x.\text{val} \rightarrow \{ if(x.\text{key}.col = y.\text{key}.row) then \} \] | i,j,k->ik |
| Mat-Vec. Mul.      | \[ M \cdot V \} = \[ \sum(y \in M) \} \} x.\text{key}.row \rightarrow x.\text{val} \rightarrow \[ V \} x.\text{key}.col) \} | ij,j->i |
| Trace              | \[ Tr(M) \} = \[ \sum(k,v) \} \} if(k.row==k.col) then v \} | ii->i |

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

### 4.2 Matrices

Matrices are considered as dictionaries mapping the row and column indices to the element value. This means that matrices with elements of type \( S \) are SDQL expressions with the type \( \{ \langle \text{row: int, col: int} \rightarrow S \} \). 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 \( 2 \times 4 \): 
\[
\begin{bmatrix}
c_0 & 0 & 0 & c_1 \\
c_0 & c_2 & 0 & 0
\end{bmatrix}
\]
This matrix is in SDQL as \{ \langle \text{row=0, col=0} \rightarrow c_0, \langle \text{row=0, col=3} \rightarrow c_1, \langle \text{row=1, col=1} \rightarrow c_2 \rangle \}. The expression \( M \cdot V \) is evaluated to the following dictionary after translating to SDQL: \{ \langle 0 \rightarrow c_0*a_0+c_1*a_2, 1 \rightarrow c_2*0 \} \}. This expression is the dictionary representation of the following vector, which is the result of the matrix-vector multiplication: \[
\begin{bmatrix}
a_0 \\
a_2
\end{bmatrix}
\].

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

**Point-wise Operations.** In many machine learning applications, it is necessary to support point-wise application of functions such as \( \cos, \sin, \text{and tan} \) on matrices. SDQL can easily support these operators by adding the corresponding scalar functions and using \( \text{sum} \) 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.

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
(a) Vertical fusion of maps in functional collections.

\[
\text{let } R1 = \sum_{(r, r_v) \in R} \{ f1(r) \to r_v \} \quad \sum_{(r, r_v) \in R} \{ f2(f1(r)) \to r_v \}
\]

(b) Vertical fusion of filters in functional collections.

\[
\text{let } R1 = \sum_{(r, r_v) \in R} \text{if } p1(r) \text{ then } \{ r \to r_v \} \quad \sum_{(r1, r1_v) \in R1} \text{if } p2(r1) \text{ then } \{ r1 \to r1_v \}
\]

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

\[
\text{let } Vt = \sum_{(row, x) \in V1} \{ row \to x * V2(row) \} \quad \sum_{(row, x1) \in Vt} \{ row \to x1 * V3(row) \}
\]

(d) Horizontal fusion for the average computation.

\[
\text{let } Rsum = \sum_{(r, r_v) \in R} r.A * r_v \quad \frac{\text{Rsum}}{\text{Rcount}}
\]

\[
\text{let } RsumRcount = \sum_{(r, r_v) \in R} \langle Rsum = r.A * r_v, Rcount = r_v \rangle \quad \frac{\text{RsumRcount.Rsum}}{\text{RsumRcount.Rcount}}
\]

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

\[
\text{let } E = S(x.B) \quad x.A*x_v*E*y.D*y_v
\]

\[
\text{let } E = S(x.B) \quad \sum_{(x, x_v) \in NR} \text{let } E = S(x.B) \quad \sum_{(x, x_v) \in NR} \text{let } E = S(x.B) \quad \sum_{(x, x_v) \in NR}
\]

\[
\text{let } E = S(x.B) \quad \sum_{(x, x_v) \in NR} \text{let } E = S(x.B) \quad \sum_{(x, x_v) \in NR} \text{let } E = S(x.B) \quad \sum_{(x, x_v) \in NR}
\]

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

\[
\text{sum}_{(x, x_v) \in NR} \text{let } E = S(x.B) \quad \sum_{(x, x_v) \in NR} \text{let } E = S(x.B) \quad \sum_{(x, x_v) \in NR}
\]

\[
\text{sum}_{(x, x_v) \in NR} \text{let } E = S(x.B) \quad \sum_{(x, x_v) \in NR} \text{let } E = S(x.B) \quad \sum_{(x, x_v) \in NR}
\]

\[
\text{sum}_{(x, x_v) \in NR} \text{let } E = S(x.B) \quad \sum_{(x, x_v) \in NR} \text{let } E = S(x.B) \quad \sum_{(x, x_v) \in NR}
\]

(g) Loop-invariant code motion for dictionary lookup in nested relations.
\begin{verbatim}
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 }

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) }

let Sp = sum(<s,s_v> in S) { jkS(s) -> g(s) } in
sum(<r,r_v> in R) sum(<s,s_v> in Sp(jkR(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.
\end{verbatim}

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 \( \{<A:\text{real},B:\text{int},C:{<D:\text{real}> \rightarrow \text{int}}\rightarrow \text{int}\} \rightarrow \text{int} \) 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 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 factored out, resulting in the right expression in the same figure.

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 factored out 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 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:

```plaintext
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].

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.

Figure 11 shows the features that are leveraged by each loop optimization. The compositional 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 \( T_1 \times T_2 \Rightarrow T_3 \) results in a function of type \( T_1 \Rightarrow (T_2 \Rightarrow T_3) \). Similarly, dictionaries with a pair key can be curried into a nested dictionary. More specifically, a dictionary of type \( \{ <a: T_1, b: T_2> \Rightarrow T_3 \} \) can be curried into a dictionary of type \( \{ T_1 \Rightarrow \{ T_2 \Rightarrow T_3 \} \} \).

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 \( \{ <a_1:A_1, ..., a_n:A_n> \Rightarrow \text{int} \} \) 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 \( \{ A_1\Rightarrow\cdots\Rightarrow(A_n\Rightarrow\text{int})\cdots\} \).

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 \( \{ \text{int} \Rightarrow \{ \text{int} \Rightarrow S \} \} \).

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

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 12 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:

```plaintext
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:

```plaintext
sum(row in A) sum(x in row.val) { x.key -> \{ y in row.val\}{y.key->x.val*y.val} }
```
16 Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu

\[
\left[ M_1 \times M_2 \right] = \sum (\text{row} \in \left[ M_1 \right]) \{ \text{row.key} -> \\
\sum (x \in \text{row.val}) \sum (y \in \left[ M_2 \right](x.key)) \{ y.key -> x.val \cdot y.val \} \}
\]

Fig. 12. Translation of matrix-matrix multiplication for curried matrices to SDQL.

| Dictionary | Factorized | Row | Columnar |
|------------|------------|-----|----------|
| \(<A=a_1, B=b_1> 1 \) | \(a_1 \) | \(\emptyset \) | \(<A=0, B=1> 1 \) |
| \(<A=a_1, B=b_2> 1 \) | \(b_1 \) | \(<A=a_1, B=b_2> 1 \) | \(<A=0, B=2> 1 \) |
| \(<A=a_2, B=b_3> 1 \) | \(b_2 \) | \(<A=a_2, B=b_3> 2 \) | \(<A=0, B=3> 2 \) |

Fig. 13. Different data layouts for relations.

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 \(R(a_1, \ldots, a_n)\) can be represented as a dictionary of type \{ idx_type \rightarrow \{a_1: A_1, \ldots, 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 \([<a_1: A_1, \ldots, a_n: A_n>]\).

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

In SDQL, columnar layout is encoded as a record where each field stores the array of its values. This representation corresponds to the array of struct representation that is used in many high performance computing applications. Generally, the columnar layout representation of the relation \(R(a_1, \ldots, a_n)\) is encoded as a record of type \(<a_1: [A_1], \ldots, a_n: [A_n]>\) 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 $T$ denote the set of all types of SDQL. $\otimes$ is a well-defined partial operation $T \times T \to T$.

**Proposition 7.2.** Every type/term defined using the inference rules of Figure 2 has a unique kind/type.

**Proof Sketch.** By induction on the structure of types/terms and case analysis on each kinding/typing rule. It is straightforward for most rules using the induction hypothesis. For the typing rules of dictionaries there are two cases on whether the dictionary is empty or not, and the type annotation ensures the property for the empty dictionary. As for $\text{sum}$ and $\text{let}$ which have a bound variable, we use the induction hypothesis on $e_1$ 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 14. A type of kind Type is interpreted as a set, while a type of kind $\text{SM}(S)$ 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 $|T_1|$, in which every element is a finite formal sum of elements of $\llbracket T_2 \rrbracket$. One can show by induction that all our types of kind $\text{SM}(S)$ are free $S$-semi-modules. Hence $\llbracket T_2 \rrbracket$ is a free $S$-semi-module and this implies that the interpretation for a dictionary type can itself be seen as a free $S$-semi-module.

For the semantics of environments $\Gamma = x_1: T_1, \ldots, x_n: T_n$, we use:

$$\llbracket \Gamma \rrbracket = \llbracket T_1 \rrbracket \times \ldots \times \llbracket T_n \rrbracket$$

A term $\llbracket \Gamma \vdash e: T \rrbracket$ is interpreted as a function from $\llbracket \Gamma \rrbracket$ to $\llbracket T \rrbracket$. When it is clear from the context, we use $\llbracket e \rrbracket$ instead of $\llbracket \Gamma \vdash e: T \rrbracket$. We use the notation $v \cdot k$ to mean the vector whose only non-zero component $v$ is at position $k$ in $\bigoplus_{a \in |T_1|} \llbracket T_2 \rrbracket$. We denote by $\gamma$ any assignment of the variables of a context $\Gamma$. The denotational semantics for terms is shown in Figure 14. $\text{Prom}_{S_1 \to S_2}$ maps the elements of the scalar semi-ring $S_1$ to $S_2$. Every scalar type $S$ is a semi-ring and as such admits distinguished elements $0$ and $1$. The action of $S$ on a type $T::\text{SM}(S)$ thus restricts to an action $\ast$ 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 $\text{sum}$, we apply the semantics of $e_2$ component-wise to the formal sum that is the semantics of $e_1$. The resulting real sum is thus over a finite support, and is therefore well-defined.

**Proposition 7.3 (Substitution Lemma).** For all $\Gamma \vdash e_1: T_1$ and $\Gamma, x: T_1 \vdash e_2: T_2$, the following holds: $\llbracket e_2 \llbracket \llbracket e_1 \rrbracket / x \rrbracket = \llbracket e_2[e_1/x] \rrbracket$. 

Fig. 14. Denotational Semantics for types and terms of SDQL.

**Theorem 7.4 (Soundness).** For all closed terms $\vdash e : T$ and $\vdash \nu : T$ where $\nu$ is a value, if $e$ reduces to $\nu$ in the operational semantics, then $e[e] = \nu[e]$. 

**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.

$$\begin{align*}
\text{let } y & = \sum(x \text{ in } e1) \{f1(x.\text{key})->x.\text{val}\} \text{ in } \sum(x \text{ in } y)\{f2(x.\text{key})->x.\text{val}\}\end{align*}$$

**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.

$$\begin{align*}
\sum(x \text{ in } e1) \text{ e2} * f(x) & = \sum_{k \in X} \text{ e2} * f(x)[y] \quad (y' = y[k<\nu, a_k>/x], [e1]_y = \sum_{k \in X} a_k \cdot k)
\end{align*}$$

$$\begin{align*}
\sum_{k \in X} \text{ e2}[y] * \sum_{k \in X} f[y][y'] & = \sum_{k \in X} \text{ e2}[y] \cdot \sum_{k \in X} f[y][y'] \quad (y' = y[k<\nu, a_k>/x], [e1]_y = \sum_{k \in X} a_k \cdot k)
\end{align*}$$
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. 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`.

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 \( D \) 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 \( 0 \rangle_D \triangleq < 0, 0^n, 0^{n \times n} > \) and \( 1 \rangle_D \triangleq < 1, 0^n, 0^{n \times n} > \). 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:

\[
\begin{align*}
    a +_D b & \triangleq < s_a + s_b, v_a + v_b, m_a + m_b > \\
    a \times_D b & \triangleq < s_a * s_b, s_a * v_b + v_a * s_b, s_a * m_a + s_a * m_b + v_a * v_b + v_b * v_a >
\end{align*}
\]

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 15. 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 15. 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 Kleene 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: \(\text{prod}(r \leftarrow \mathcal{R}) \; p(r)\). The commutative monoid structure of multiplication allows for optimizations with a similar impact as horizontal loop fusion (cf. Figure 15).

**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 produces such as batch gradient descent (BGD). The additional expressive power of

---

### Additional Transformation Rules for Language Extensions of SDQL

| SDQL[ring] |  
| --- | --- |
| \(-(-e)\) | \(\sim e\) |
| \(e + (-e)\) | \(\sim 0\) |

| SDQL[closure] |  
| --- | --- |
| \(1 + e \cdot \text{closure}(e)\) | \(\sim \text{closure}(e)\) |
| \(1 + \text{closure}(e) \cdot e\) | \(\sim \text{closure}(e)\) |

| SDQL[prod] |  
| --- | --- |
| \((\text{prod}(x \;\text{in} \; e1)\; f1(x)) \cdot (\text{prod}(x \;\text{in} \; e1)\; f2(x))\) | \(\sim \text{prod}(x \;\text{in} \; e1)\; f1(x) \cdot f2(x)\) |

| SDQL[rec] |  
| --- | --- |
| \(\text{rec}(x \Rightarrow \text{let} \; y=e1 \;\text{in} \; f(x,y))(e2)\) | \(\sim \text{let} \; y=e1 \;\text{in} \; \text{rec}(x \Rightarrow f(x,y))(e2)\) |
this construct comes with limited optimization opportunities; loop fusion and factorization are no longer applicable to them, however, code motion can still be leveraged (cf. Figure 15).

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. 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 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].

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 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 16 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],

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. 17. 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 $100000 \times 100$, and the dimension of the input matrix of the right figure is $N \times 100$ with the density of $2^{-7}$.

Sparse Matrix Processing. First, we consider the task of computing the covariance matrix $X^TX$ (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 17 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 $2\times$ 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 $3\times$ and $2\times$ faster than the COO and CSR representations of SciPy, respectively, thanks to fusion and the efficient low-level code generated by SDQL.

Sparse Tensor Processing. Next, we consider three higher-order tensor workloads on NELL-2, a real world dataset coming from the Never Ending Language-Learning project [Carlson et al. 2010]. Table 2 shows the performance comparison for these workloads. We observe that especially for a medium range of sparsity SDQL is faster than taco (from $1.4\times$ to $23\times$). For sparser scenarios, taco shows better performance (up to $1.3\times$), 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 18 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
Table 2. 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.

| Kernel   | LA Formulation       | Sparsity | 2-11  | 2-9   | 2-7   | 2-5   | 2-3   |
|----------|----------------------|----------|-------|-------|-------|-------|-------|
|          |          |          | SDQL  | taco  | SDQL  | taco  | SDQL  | taco  | SDQL  | taco  |
| TTV      | $A_{ij} = \sum_k B_{ijk} C_k$ | 2-11     | 621.8 | 466.3 | 621.8 | 544.9 | 632.0 | 866.2 | 661.8 | 2088.1 |
| TTM      | $A_{ijk} = \sum_k B_{ijl} C_{kl}$ | 2-9      | 4534.2| 5936.2| 4679.6| 7851.6| 4764.2| 15563.9| 5189.2| 46153.7|
| MTTKRP   | $A_{ij} = \sum_{k,l} B_{ijk} C_k D_{lj}$ | 2-7      | 5.6   | 4.3   | 18.4  | 17.3  | 32.2  | 60.4  | 103.2 | 388.1  |

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

Fig. 18. Run time results for computing covariance matrix over nested and relational data.

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

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 18a 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 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 18b 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 19 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 19 shows that the NumPy-based implementation over the materialized join can have a better performance for relations with the same number of features. The factorized computation starts showing its benefits for larger feature ratios. MorpheusPy is always better than the flat representation of SDQL. This is thanks to vectorization, which shows its impacts further as the feature ratio increases. Finally, we observe a superior performance for SDQL once the curried representation and loop factorization are used. As the tuple ratio increases, the speed up of SDQL over MorpheusPy climbs up to \( 1.7 \times \); this is because of loop factorization enabled by the curried representation for relation \( S \). MorpheusPy expresses aggregations and joins in terms of linear algebra operations using NumPy, which do not benefit from such optimizations.

10 RELATED WORK

In this section, we review the literature. Table 3 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 Subrahmanyan 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 [Coupts 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 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 [Steuer 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 Puschel 2016] exploit the memory hierarchy and make careful decisions on tiling and scheduling decisions. These DSLs exploit the memory hierarchy by relying on searching algorithms for making tiling and scheduling decisions. The generated output is a C function that includes intrinsics to enable SIMD vector extensions. SPL [Xiong et al. 2001] is a language that expresses recursion and mathematical formulas. TACO [Kjolstad et al. 2017] generates efficient low-level code for compound linear algebra operations on dense and sparse matrices. 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
Table 3. 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 Tensors | Dictionary | Semi-rings | Loop Fusion | Loop Hoisting | Loop Memoization | Code Generation | Vectorization |
| SDQL (This Paper)        | ●              | ●                   | ●                 | ●                  | ●              | ○        | ○          | ○              | ○           | ●        | ○        | ○           | ○               | ●                  |
| Query Compilers (HyPer)  | ●              | ●                   | ●                 | ●                  | ●              | ○        | ○          | ○              | ○           | ●        | ○        | ○           | ○               | ●                  |
| Vectorized Query Engines (Vectorwise) | ●           | ●                   | ●                 | ●                  | ●              | ○        | ○          | ○              | ○           | ●        | ○        | ○           | ○               | ●                  |
| Monad Calculus, NRC⁴     | ●              | ●                   | ●                 | ●                  | ●              | ○        | ○          | ○              | ○           | ●        | ○        | ○           | ○               | ●                  |
| Monad Comprehension      | ●              | ●                   | ●                 | ●                  | ●              | ○        | ○          | ○              | ○           | ●        | ○        | ○           | ○               | ●                  |
| 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 (LPLA) | ○             | ●                   | ●                 | ●                  | ●              | ○        | ○          | ○              | ○           | ●        | ○        | ○           | ○               | ●                  |
| DB/LA by unified IR (IPAQ) | ○             | ●                   | ●                 | ●                  | ●              | ○        | ○          | ○              | ○           | ●        | ○        | ○           | ○               | ●                  |
| DB/LA by combined IR (Raven) | ○            | ●                   | ●                 | ●                  | ●              | ○        | ○          | ○              | ○           | ●        | ○        | ○           | ○               | ●                  |

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
Functional Collection Programming with Semi-ring Dictionaries

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 LMFAQ [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 3). 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 database systems that are especially built for database workloads, and thus cannot efficiently handle other use-cases including sparse linear algebra, and in-database machine learning 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).

REFERENCES

Martin 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.
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 Fancellu, Sara Gandini, Pier Giuseppe Pellici, 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, 325–356.

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, doids 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 Idr eos, F Groffen, N N es, 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.
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 Database 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, Frédó 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, Henry 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, Hunq 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 Dashi, 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) *(HPHC ’17)*. 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 Parraux, Lewis Brown, Mohammad Dashi, 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 Parraux. 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 Ghiita, 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.
A TRANSLATION OF RELATIONAL ALGEBRA

In this section, we explain the translation of relational operators to SDQL, as shown in Figure 4.

Selection. Consider the translation of relation R in SDQL, which is represented as ⟦R⟧. The selection operator, represented as 𝜎𝑝(R), filters the elements that satisfy a predicate p. For each element x of this relation, if the predicate is satisfied, we return the singleton set containing x.key. Otherwise, we return an empty set. As the body of the loop returns a set, this loop performs a set union, which results in a filtered relation.

Projection. This operator projects a subset of attributes specified by the function f, and is represented as 𝜋𝑓(R). Similar to selection, we iterate over the elements of ⟦R⟧; at each iteration we return a singleton set with the applied projection function f on each row of relation (f(x.key)).

Union. Set union is achieved by using the + operation of the boolean semi-ring, i.e., boolean disjunction, on the values of elements with the same key. For the elements that exist only in one of the collections, the value associated with the element in the other collection is considered as false.

Intersection. Set intersection is achieved by iterating over the elements of the first collection. At each iteration, if an element with the same key exists in the other collection, a singleton set of that element key is returned, otherwise an empty set is returned.

Difference. This operator is symmetric to intersection with the difference that if the element key exists in the second collection, an empty set is returned. Otherwise, a singleton set is returned.

Cartesian Product. For this operation, we use nested loops iterating over the elements x and y of relations R and S, respectively. For each combination of tuples, we return a singleton set that has the combination of the tuples of these two relations as its element.

Inner Join. This operator is expressed similarly to the Cartesian product operator. For each combination of elements, if the selection predicate is satisfied the joined tuples are emitted.

Semi Join. R left semijoin S is the set of all tuples in R for which there is a matched tuple in S. It can be simulated using a natural join followed by the projection over the attributes of R. The function concat joins two tuples from R and S, whereas projR extracts a record with attributes of R from the joined tuple:

```
let RjS = sum(x in R) sum(y in S) if(join(x.key, y.key)) then { concat(x.key, y.key) } in
sum(x in RjS) { projR(RjS) }
```

Anti Join. R left antijoin S is similar to the semijoin, except that its result is only those tuples of R for which there is no matched tuple in S. It can be expressed by substracting R left semi-join S from R:

```
let RjS = sum(x in R) sum(y in S) if(join(x.key, y.key)) then { concat(x.key, y.key) } in
let RsjS = sum(x in RjS) { projR(RsjS) } in
sum(x in R) if(not(RsjS(x.key))) then { x.key }
```

Right semi/anti join can be expressed similarly.

Outer Join. R left outer join S is expressed as:

```
let RjS = sum(x in R) sum(y in S) if(join(x.key, y.key)) then { <r=x.key, s={y.key}> } in
let Rproj = sum(xy in RjS) { xy.key.r } in
let RpS = sum(x in R) if(not(Rproj(x.key))) then { <r=x.key, s={}> } in
in RjS + RpS
```

The expression RjS corresponds to R inner join with S, Rproj corresponds to the projection over the attributes of R, and RpS corresponds to the elements of R that didn’t join with any element from
S padded with NULL, which is {} in this case. Finally, we compute the union of \( R\cdot J\cdot S \) and \( R\cdot P\cdot S \). The right outer join and full outer joins are expressed similarly.

B TRANSLATION OF NESTED RELATIONAL CALCULUS

**Bag Construction.** The empty bag construction is expressed by an empty dictionary. The singleton bag construction is achieved by constructing a dictionary with the given element as its key, and the multiplicity of one as its value.

**Flattening.** The flattening operation is only performed on an input parameter that is nested. Thus, the translated input is a dictionary where the key is also a dictionary. In order to flatten the input dictionary, one has to union its keys; however, one needs to multiply the multiplicity with the keys to take into account the bag semantics, represented as \( x.val \times x.key \).

**For-comprehensions.** Similar to the flattening operator, bag semantics requires that the translation of the body of the loop be multiplied by the multiplicity.

**Bag Union.** Similar to set union in relational algebra, bag union in \( \text{NRC}^+ \) is also achieved by addition on the translation of the operands.

**Bag Product.** Similar to Cartesian Product in relational algebra, we iterate over each combination of the elements of the two inputs. The key of the result dictionary is a pair of the keys of inputs. The value is the multiplication of the values of two inputs in order to take into account the bag semantics.

C TRANSLATION OF AGGREGATIONS

**Scalar Aggregate.** This operator can be implemented by iterating over the elements of the relation and computing the appropriate aggregate function \( f \) (cf. first and third rules of Figure 6). As the relations have bag semantics, there could be duplicates of an element in the input relation, the multiplicity of which is shown by \( x.v \); thus, the aggregate result for each element needs to be multiplied by \( x.v \). The following example shows the translation of sum and count queries:

\[
\text{SELECT SUM(R.A) FROM R} \quad \leadsto \quad \text{sum}<r,r_v> \text{ in R} \quad \rightarrow \quad r_v \times r.A \\
\text{SELECT COUNT(*) FROM R} \quad \leadsto \quad \text{sum}<r,r_v> \text{ in R} \quad \rightarrow \quad r_v
\]

**Group-by Aggregate.** As opposed to its scalar variant, a group-by aggregate returns a single dictionary with the key specified by the grouping function \( g \), and the value specified using the aggregate function \( f \) (cf. second and fourth rules of Figure 6). The following example shows the translation of group-by sum and group-by count queries:

\[
\text{SELECT SUM(R.A) FROM R} \quad \leadsto \quad \text{let tmp} = \text{sum}<r,r_v> \text{ in R} \quad \rightarrow \quad r.B \rightarrow r_v \times r.A \\
\text{GROUP BY R.B} \quad \quad \quad \quad \quad \text{in sum}<x,x_v> \text{ in tmp} \quad \rightarrow \quad \{ \text{<key=x, val=x_v> -> 1} \} \\
\text{SELECT COUNT(*) FROM R} \quad \leadsto \quad \text{let tmp} = \text{sum}<r,r_v> \text{ in R} \quad \rightarrow \quad r.B \rightarrow r_v \\
\text{GROUP BY R.B} \quad \quad \quad \quad \quad \text{in sum}<x,x_v> \text{ in tmp} \quad \rightarrow \quad \{ \text{<key=x, val=x_v> -> 1} \}
\]

**Nest.** This operator performs grouping without aggregation. This means that the output is a nested relation, that is only supported by \( \text{NRC}^{agg} \), not relational algebra. Similar to the group-by aggregate operator, at each iteration it returns a singleton dictionary with the key specified by the function \( g \), and the value is a singleton dictionary with \( x \) as key and \( x.v \) as value (cf. last rule of Figure 6).

D TRANSLATION OF LINEAR ALGEBRA

**Vector Addition.** This operation is expressed as the addition of the translated dictionaries in SDQL, which results in a dictionary where the values of the elements with the same key are summed.

**Scalar-Vector Multiplication.** The multiplication of a scalar value with a vector is translated to the multiplication of the translated scalar SDQL expression with the translated dictionary. The
result expression is evaluated to a dictionary with the same keys as the translated dictionary, with the values multiplied by the scalar expression.

**Vector Hadamard Product.** The Hadamard product of two vectors, or the element-wise multiplication, is achieved by iterating over the elements of the translated dictionary of the first vector, and constructing a singleton dictionary with its key, and the value multiplied by looking up the value associated with the translated dictionary of the second vector. If an element with the same key (index) does not exist in the second dictionary, the singleton dictionary will have a zero value. When the result dictionary is constructed, singleton dictionaries are ignored when computing the union of the intermediate dictionaries.

**Vector Dot Product.** This operation is achieved by iterating over the elements of the translation of the first vector, and summing the multiplication of the associated value and the corresponding value of the second vector.

**Vector Summation.** Finally, the summation of the elements of a vector is expressed by iterating over the elements of the vector and adding its values.

**Matrix Transpose.** The transposition of a matrix is expressed by iterating over the elements of the translated dictionary and constructing a dictionary where the key is a record with the same value, but with the row and column swapped.

**Matrix Addition, Scalar-Matrix Multiplication, Matrix Hadamard Product.** These operators are expressed similarly to the corresponding vector operators.

**Matrix-Matrix Multiplication.** This operator is expressed by iterating over each combination of the elements of two matrices. At each iteration, if the column of the element from the first matrix is the same as the row of the element of the second matrix, a singleton dictionary is created, where the key is with the row of the first element, and column of the second element, and the value is the multiplication of both elements. Otherwise, and empty dictionary is created.

**Matrix-Vector Multiplication.** This operator is expressed by iterating over the elements of the translated matrix, and constructing a singleton dictionary where the key is the row of this element, and the value is the multiplication of this element with the element from the vector associated with its column.

**Matrix Trace.** The trace of a matrix is the result of summation of the elements of a diagonal of a matrix. This can be expressed by iterating over the elements of a matrix and adding the value of that element if its row and column are identical, and otherwise adding zero.

## E TRANSLATION OF CURRIED LINEAR ALGEBRA

Figure 20 shows the translation of matrix operator, in the case of using curried representation for them.

## F CORRECTNESS OF LOOP OPTIMIZATIONS

**Proposition F.1.** The horizontal loop fusion rules of Figure 8 are sound.

**Proof.**

\[
\begin{align*}
&\text{let } y_1 = \sum_{x \in e_1} f_1(x) \text{ in let } y_2 = \sum_{x \in e_1} f_2(x) \text{ in } f_3(y_1, y_2) \\
&= \text{let } y_2 = \sum_{x \in e_1} f_2(x) \text{ in } f_3(y_1, y_2) \\
&= \text{let } y_2 = \sum_{x \in e_1} f_2(x) \text{ in f3(y1, y2)} \\
&\quad (y' = y[\sum_{x \in e_1} f_1(x)/y_1]) \\
&= \sum_{x \in X} f_3(y_1, y_2) \\
&\quad (y' = y[\sum_{x \in e_1} f_1(x)/y_1, \sum_{x \in e_1} f_2(x)/y_2]) \\
&= \sum_{x \in X} \sum_{k \in X} f_3(y_1, y_2) \\
&\quad (y' = y[\sum_{x \in e_1} f_1(x)/y_1, \sum_{x \in e_1} f_2(x)/y_2]) \\
&= \sum_{x \in X} \sum_{k \in X} a_k \cdot k
\end{align*}
\]
We now give a standard call-by-value small-step operational semantics to SDQL. The syntax for records and dictionaries with multiple arguments, the evaluation order is from left to right. Next, a semi-ring with zero denoted by $0_T$ is a macro, defined by induction on $T$ as follows. $0_T$ is the constant 0 of the scalar type $S$. 0 is a semi-ring with zero denoted by $0_T$. $0_T$ is a macro defined by induction on $T$. For construction of records and dictionaries with multiple arguments, the evaluation order is from left to right. Next, we introduce some lemmas.

**Proposition F.2.** The rewrite rule for loop-invariant code motion in Figure 8 is sound.

**Proof.**

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

**Proposition F.3.** The rewrite rules for loop memoization in Figure 8 are sound.

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

**G OPERATIONAL SEMANTICS**

We now give a standard call-by-value small-step operational semantics to SDQL. The syntax for evaluation context and values as well as reduction rules are shown in Figure 21. All our types form a semi-ring with zero denoted by $0_T$. $0_T$ is a macro, defined by induction on $T$ as follows. $0_T$ is the constant 0 of the scalar type $S$. $0_T$ is a semi-ring with zero denoted by $0_T$. $0_T$ is a macro defined by induction on $T$. For construction of records and dictionaries with multiple arguments, the evaluation order is from left to right. Next, we introduce some lemmas.
Evaluation contexts

\[ E ::= \text{sum}(x \text{ in } E) \mid E(e) \mid \text{let } x = E \text{ in } e \mid \text{if}(E) \text{ then } e \text{ else } e \mid \{ v \rightarrow v, \ldots, E \rightarrow e, \ldots \} \mid \langle a_1 = v, \ldots, a_i = E, \ldots \rangle \mid E.a \mid E + e \mid v + E \mid \text{promote}_{S,S}(E) \mid [] \]

Values

\[ v ::= \{ v \rightarrow v, \ldots \} \mid \langle a = v, \ldots \rangle \mid n \mid r \mid \text{false} \mid \text{true} \mid 0 \]

---

**Fig. 21. Reduction rules for SDQL.**

**Lemma G.1 (Confluence).** Let \( \Gamma \vdash e : T \). If \( e \rightarrow e_1 \) and \( e \rightarrow e_2 \), there exists \( e' \) such that \( e_1 \rightarrow^* e' \) and \( e_2 \rightarrow^* e' \).

**Proof Sketch.** By inspection, the only non deterministic cases are dictionary addition and \( \text{sum} \) (that requires ranging over a dictionary). Technically, our dictionaries are unordered. This allows \( + \) on semi-ring dictionaries to be commutative. \( \square \)

**Lemma G.2 (Type Preservation).** If \( \Gamma \vdash e : T \) and \( e \rightarrow e' \), then \( \Gamma \vdash e' : T \).

**Proof Sketch.** By induction on the structure of \( e \) and case analysis on each reduction rule. \( \square \)

**Lemma G.3 (Fundamental Lemma).** For every \( x_1 : T_1, \ldots, x_n : T_n \vdash e : T \) and every value \( v_1 : T_1, \ldots, v_n : T_n \), \( e[v_1/x_1, \ldots, v_n/x_n] \) reduces to a value.

**Proof Sketch.** By induction on the structure of \( e \), then case analysis on each typing rule. As usual, the quantification is for all \( n \) and not for fixed \( n \). \( \square \)
Theorem G.4. Every closed and well-typed term e reduces to a unique value.

Proof Sketch. By choosing $\Gamma = \emptyset$ in Lemma G.3.

H SOUNDNESS OF THE DENOTATIONAL SEMANTICS

Proof of the substitution lemma. We only show the non standard cases.

• Case of dictionary creation:

\[
\left[\left\{ k_i \rightarrow v_i \right\} \right]_{\gamma}(\left[ e \right]_{\gamma}/x) \\
= \sum_i \left[ k_i \right]_{\gamma} \cdot \left[ v_i \right]_{\gamma}(\left[ e \right]_{\gamma}/x) \\
= \sum_i \left[ k_i[e/x] \right]_{\gamma} \cdot \left[ v_i \right]_{\gamma}(\left[ e \right]_{\gamma}/x) \\
= \left[ \left\{ \left(k_i[e/x] \rightarrow v_i[e/x] \right) \right\} \right]_{\gamma}
\]

= \left[ \left\{ k_i \rightarrow v_i \right\}[e/x] \right]_{\gamma}

• Case of sum introduction:

\[
\left[ \sum (x \text{ in } e1) e2 \right]_{\gamma}(\left[ e \right]_{\gamma}/y) \\
= \sum_{x \in X} \left[ e2 \right]_{\gamma}(\left[ e \right]_{\gamma}/y) \\
= \sum_{x \in X} \left[ e2[e/y] \right]_{\gamma}(\left[ e \right]_{\gamma}/y) \\
= \left[ \left( \sum (x \text{ in } e1[e/y]) e2[e/y] \right) \right]_{\gamma}
\]

PROOF OF THE SOUNDNESS THEOREM. Most rules follow from the S-semi-module structure of types, or standard denotational semantics in sets and functions. The only non standard case is sum, but the result follows from associativity of addition, and 0 being the unit of addition.

I CORRECTNESS OF OPTIMIZATIONS USING OPERATIONAL SEMANTICS

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

Proposition I.1 (Correctness of Vertical Loop Fusion). The vertical loop fusion rules of Figure 8 are sound.

Proof Sketch. The correctness of the first rule can be proved by performing induction on the value of the dictionary $d=\{k_1 \rightarrow v_1, \ldots, k_{n+1} \rightarrow v_{n+1}\}$ where $e1 \rightarrow^* d$. The correctness of the base case $d=\{\}$ is obvious. For the induction step, one has to consider different cases based on whether $f1(k_{n+1})$ is equivalent to $f1(k_i)$ for $i \leq n$. If this is the case the proof is straightforward. If this is not the case, there will be two further cases. Assuming $f1(k_{n+1}) \rightarrow^* k'_p+1$, either $f2(k'_p+1)$ is equivalent to $f2(k'_j)$ for some $j \leq p$. In each case, both LHS and RHS are evaluated to the same value.

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

Proposition I.2 (Correctness of Horizontal Loop Fusion). The horizontal loop fusion rules of Figure 8 are sound.

Proof Sketch. Straightforward by induction on the value of dictionary $d$ which is the result of evaluating $e1$.

Proposition I.3 (Correctness of Loop Factorization). The loop factorization rules of Figure 8 are sound.
Proof Sketch. By induction on the values of the dictionary $d$ which is the result of evaluating $e_1$. For the inductive step, we use the distributive law of the semi-ring structure.

Proposition I.4 (Correctness of Loop-Invariant Code Motion). The rewrite rule for loop-invariant code motion in Figure 8 is sound.

Proof Sketch. $e_1$ reduces to a value $\{ k_1 \rightarrow v_1, \ldots, k_n \rightarrow v_n \}$. The LHS reduces to $\sum_i (\text{let } y = e_2 \text{ in } 1)*f(x, y)[k_i, v_i/x]$, where $\sum_i g_i$ is a shorthand for $g_1 + \ldots + g_n$. Assuming $e_2$ reduces to a value $v$, the first element of the summation reduces to $f(x, y)[k_1, v_1/x, v/y]$. This term then reduces to a value $f_1$. Similarly, for each $i$, $f(x, y)[k_i, v_i/x, v/y]$ reduces to $f_i$. Hence, the LHS eventually reduces to $\sum_i f_i$. In the RHS, $e_2$ reduces first to the value $v$. Then the RHS reduces to $\sum x \in e_1 f(x, y)[v/y]$. We then conclude as before. $e_1$ reduces to a value $\{ k_1 \rightarrow v_1, \ldots, k_n \rightarrow v_n \}$ and the RHS reduces to $\sum f_i$. In summary, what makes this optimization correct is that substituting $x$ then $y$ is the same as substituting $y$ then $x$.

Proposition I.5 (Correctness of Loop Memoization). The rewrite rule for loop memoization in Figure 8 is sound.

Proof Sketch. By induction on the dictionary $d$ which is the result of evaluating $e_1$. 

□