Empowering big data analytics with polystore and strongly typed functional queries

Annabelle Gillet, Eric Leclercq, Marinette Savonnet, Nadine Cullot

Empowering Big Data Analytics with Polystore and strongly typed functional queries

Annabelle Gillet
LIB Univ. Bourgogne Franche Comté
Dijon, France
annabelle.gillet@depinfo.u-bourgogne.fr

Éric Leclercq
LIB Univ. Bourgogne Franche Comté
Dijon, France
eric.leclercq@u-bourgogne.fr

Marinette Savonnet
LIB Univ. Bourgogne Franche Comté
Dijon, France
marinette.savonnet@u-bourgogne.fr

Nadine Cullot
LIB Univ. Bourgogne Franche Comté
Dijon, France
nadine.cullot@u-bourgogne.fr

ABSTRACT

Polystores are of primary importance to tackle the diversity and the volume of Big Data, as they propose to store data according to specific use cases. Nevertheless, analytics frameworks often lack a uniform interface allowing to fully access and take advantage of the various models offered by the polystore.It also should be ensured that the typing of the algebraic expressions built with data manipulation operators can be checked and that schema can be inferred before starting to execute the operators (type-safe).

Tensors are good candidates for supporting a pivot data model. They are powerful abstract mathematical objects which can embed complex relationships between entities and that are used in major analytics frameworks. However, they are far away from data models, and lack high level operators to manipulate their content, resulting in bad coding habits and less maintainability, and sometimes poor performances.

With TDM (Tensor Data Model), we propose to join the best of both worlds, to take advantage of modeling capabilities of tensors by adding schema and data manipulation operators to them. We developed an implementation in Scala using Spark, providing users with a type-safe and schema inference mechanism that guarantees the technical and functional correctness of composed expressions on tensors at compile time. We show that this extension does not induce overhead and allows to outperform Spark query optimizer using bind join.

CCS CONCEPTS

• **Information systems → Query languages**; **Data structures**.

KEYWORDS

High performance data analytics, Polystore, Query language, Tensor

ACM Reference Format:
Annabelle Gillet, Éric Leclercq, Marinette Savonnet, and Nadine Cullot. 2020. Empowering Big Data Analytics with Polystore and strongly typed functional queries. In *24th International Database Engineering & Applications Symposium (IDEAS 2020), August 12–14, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3410566.3410591

IDEAS 2020, August 12–14, 2020, Seoul, Republic of Korea
2020. ACM ISBN 978-1-4503-7503-0/20/06.
https://doi.org/10.1145/3410566.3410591

1 INTRODUCTION AND MOTIVATIONS

The variety and the volume of Big Data have changed the storage needs. Rather than having one storage system to keep everything, a multitude of new specialized storage engines have emerged (e.g. NewSQL, NoSQL, Column Stores, Distributed File Systems, Graph databases), each one corresponding to a use case more or less specific. The well-known article of M. Stonebraker [37], "one size does not fit all", explains that the full potential of data will be better exploited with a polystore architecture. A polystore refers to a system that integrates heterogeneous database engines, storage systems and multiple data manipulation or programming languages using different paradigms [14]. The use of polystore brings several advantages: it allows to organize data according to particular use cases (e.g. graph DBMS well support linked data and graph traversal or path queries); it enables parallel processing among several data store according to the specificities of each kind of system in the polystore [4, 22].

Researches on polystores try to overcome the limitation of traditional tools. Extract Transform Load (ETL) processes and warehousing technologies as well as in-database analysis operators are not sufficient to support complex analytics pipelines. ETL processes are expensive and time consuming tasks, and transforming multiple datasets into a single data model in a data warehouse can impact negatively performances and reduce the expressivity of the original data model. In-database analysis cannot take easily into account new algorithms, as they require a specific development for each database model in order to fit the data structure required by the algorithm [26, 27]. So researches on polystores are directed towards ETL streaming systems [13, 41], multi-database query language [14], unification models [15], and parallel query processing as well as the integration of polystore with analysis frameworks [22]. This article focus on the last three points.

Analytics tasks require multiple kinds of algorithms based on different theoretical foundations, such as linear algebra, statistics, graph theory. Algorithms are implemented using different computing paradigms such as GPU, map-reduce, concurrent, parallel, or functional programming, and they are used as operators over data in data analytics pipelines. With the adoption of different data models combined with data analysis frameworks and new techniques such as machine learning, we are able to extract more information from data and to have a deeper understanding of the studied phenomena.

Nevertheless, data processing pipelines are becoming complex and heterogeneity is no longer observed only at the data level but also at the analysis level [2]. Developing a pivot data model which can generalized polystore underlying data models and which can facilitate data transformations to feed quickly algorithms implemented with frameworks is a major challenge [17, 19, 26].

Multiple data analysis frameworks built on different computing paradigms are available, such as Spark [42] (with SparkSQL, GraphX, MLlib), Tensorflow [1], PyTorch [33], Theano [3], TensorLy [23], NumPy [40]. However, frameworks often focus exclusively on applying algorithms, and not on manipulating or transforming data, even if it is a time-consuming and error-prone task. Each of these tools lacks one or several important properties, such as type-safe functions (operators) which guarantee at compile time that a composition of operations is valid (a property that is automatically lost when using dynamic typing, leading to errors during execution), or schema inference when manipulating data. With the absence of these properties, two categories of errors can arise: 1) type errors, when operators are not applied on the right attribute type and 2) functional errors, when operators are applied on the right attribute type but not on the attribute representing the desired information. The second category of errors is the hardest to detect, but also the most dangerous, because the error will be unnoticed, the computations will occur but will give an incorrect result. For example, an inversion between two columns of a dataset[1] has already led to a major mistake, that has been discovered only years later, and that has resulted in the retraction of five articles and impacted the work of other researchers that were using the erroneous result in their work. This ascertainment highlights the need of checking the correctness of analysis workflows, as shown by the evolution of the data structure in Spark: from *RDD*s with unstructured data, to *DataFrame*s with a columnar format, and finally to *Dataset*s with a type-safe layer over *DataFrame*s. Thus, if composition of operators in programs could have a mean to prevent this type of mistakes, without needing user intervention and without adding complexity to the use, data scientists could focus more on the core of the analysis rather than on controlling the result at different steps of the workflow.

In recent analytics frameworks, tensors play an important role. They are abstract and powerful mathematical objects used in multiple data analytics tools [38], including deep learning to deal with multi-dimensional data or data mining to analyze latent relationships using tensor decompositions [21, 32]. However, their pure mathematical definition is relatively abstract, far away from user needs. Their implementation in tools does not really suit coding standards and brings several bad habits, usually banned from good practices of software development, that reduce evolution, reuse, collaborative work, etc. In frameworks such as Tensorflow, Theano or NumPy, tensors are usually defined as multidimensional arrays; users assign an implicit meaning to dimensions' integer indexes and at best put comments in their code to remember the meaning of constructions [35]. It also may concern dimension names or dimension order before or after tensor transformations are applied. This can lead to errors which cannot be easily understood because the computation is technically correct but functionally incorrect.

[1]https://people.ligo-wa.caltech.edu/~michael.landry/calibration/S5/getsignright.pdf

Moreover, tensors are disconnected from data models, data schemas and data sources, failing to take advantage of the expressiveness of data models and semantically well-defined data manipulation operators. It is necessary to use intermediate data structures to perform complex data manipulations before applying tensorial operators such as decompositions. This is even more true when handling multiple data sources, and strengthens the need of a pivot data model.

In [24] we have formally defined a semantically rich pivot data model, the Tensor Data Model (TDM), by adding schema to the tensor mathematical object. It provides users with operators that can be combined to express complex data transformations. We have showed that tensors make it possible to generalize common data models, and that virtual or materialized views can be defined among multiple data sources (polystore) using operators on tensors. In this article our contribution are the following: we propose a set of mechanisms to ensure type-safe property and schema inference. As we stated, strong static typing is of primary importance in Big Data analytics because it allows to determine errors before the execution and thus to avoid expensive buggy calculation phases which will end with errors or inconsistencies. We describe our implementation of the operators on the top of Spark that fulfill the type-safe and schema inference properties and includes a mechanism to connect to a polystore.

The article is organized as follow: section 2 is a related work on analytics frameworks and their use of tensors, including the role of pivot data model and query processing in polystores; section 3 gives an overview of the key features of TDM; section 4 describes the mechanism for type-safety and schema inference in functional queries; section 5 describes experiments on the top of Spark and shows how to perform optimisations on tensor construct queries.

2 RELATED WORK

This section describes three kinds of inter-related researches: i) support of tensor in analytics frameworks and linear algebra in programming languages ; ii) multidimensional arrays data models in databases, query languages and dataframes ; iii) polystores and their integration in analytics frameworks.

NamedTensor [35] is a first step to make tensors safer and usable in complex workflows. Built on Torch tensor [33], it proposes to use *String* names for dimensions instead of *Integer* indexes. However, Python dynamic typing system does not guarantee safety, and naming dimensions with *String* does not put away the risk of a typo or the referencing to an old dimension that was removed in a code update. In [10], Chen with the Nexus[2] prototype pushes the tensor safety further than NamedTensor, by providing a statically typed tensor abstraction using Scala. Classical tensor operators are defined, but from a mathematical point of view and not from a data model point of view. So the abstraction level is low. Data transformations are performed using ad-hoc program constructions and not by using well-defined data manipulation operators or a query language.

The works of Muranushi et al. [29] and Griffioen [16] propose a typed linear algebra system, that leads to a more functional and index-free matrix and allows to infer the type of any linear algebra

[2]https://github.com/ctongfei/nexus

expression. Their implementation in Haskell can detect typing error at compile time. In [29] they apply it for encoding units of measure in astrophysics data.

In the field of data models and query languages, Bauman [7], Libkin et al. [25] have proposed a query language for multidimensional arrays. More recently Brijder et al. [8] study the expressive power of a language for matrix manipulation including linear algebra and graph operations. Barceló et al. [6] study the expressiveness of Lara, a language and a data model built on associative array. The model generalizes standard data models and can also represent tensor but as the building block is an associative array, it forces users to decompose complex relationships into binary ones. Typed arrays are also part of the SQL standard as multi-dimensional arrays [28], but they are mainly designed to be used with an in-database analysis approach or with homogeneous systems, and not with machine learning frameworks and map-reduce paradigm. So, it is much an extension of SQL with new data type rather than a model which is intrinsically based on multi-dimensional structures and can benefit of theoretical results. Spark [42] is a major actor in this field. Its most advanced data structure, the *Dataset*, provides type-safe guarantee at compile time. Unfortunately, schema transformations such as joining two *Dataset*s do not carry on automatically the type-safe property nor automatic schema inference[3].

To integrate different systems in a polystore and to connect it with analytical tools, two approaches can be distinguished: the model approach and the language approach. The goal of the model approach [6, 15] is to build a pivot model that can support all data models of the polystore. However, this approach has often a low level of abstraction, and thus reduces the expressiveness. The language approach [4, 22] defines a multidatabase language to manipulate data. It can be mixed with native queries to directly access a specific storage system. However, operators that need to be applied on a set of results from different data sources have to use the common language, that is often close to the SQL standard. Some works have tried to propose an evaluation framework to measure the ability of a query processing technique over heterogeneous data models [39], that use multiple criteria: the heterogeneity (dealing with several databases without losing expressivity), the autonomy (each store can be managed independently of the polystore system), the transparency (having an easy access to data), the flexibility (building multiple kind of workflows) and optimality (benefiting of the optimization of the stores composing the polystore).

To sum up, tensors are a powerful tool for complex analytics pipelines and for generalizing data models in a polystore architecture. Existing implementations of tensors in analytics frameworks stick to their mathematical nature and lack of operators for manipulating data, though at the center of analysis. Furthermore, the need of a type-safe property for analytics tools is essential, as shown by different works on tensors or on specific data structures. However, type-safety cannot be achieved easily in a dynamically typed language such as Python, and complex data manipulation operators that lead to schema transformation can induce a loss of the type-safe property.

[3]See for example https://medium.com/@pahomov.egor/spark-datasets-are-not-as-type-safe-as-you-think-56a8a9ea0fc

3 AN OVERVIEW OF TDM DATA MODEL

Our aim is to define a pivot model for polystores (figure 1) to benefit from all the different underlying data models, without loosing in expressivity. High level of expressivity is achieved by using tensors, as they have facilities to map to various models, and by allowing native queries for accessing the stores. Moreover, native queries can take advantage of the capabilities of each database individually. The execution of analytics algorithms is then facilitated, as their input data structures are obtained and transformed from TDM and not from each data model underneath. Moreover, the tensorial nature of TDM allows also naturally the use of rich tensorial operators, such as decompositions [34].

3.1 TDM: Algebraic Structure and Operators

Tensors are abstract mathematical objects which can be considered according to various points of view such as family of elements, or multi-linear applications. To be closer to usual definition of data model using set theory we will retain the definition of a tensor as an element of the set of the functions from the product of N sets $I_j, j = 1, \ldots, N$ to $\mathbb{R}$: $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, where N is the number of dimensions of the tensor or its order.

For adding the useful notion of schema to tensors we have formally defined the notion of typed associative array and typed tensor [24]. A named and typed associative array is a triple $(Name, \mathbf{A}, T)$ where $Name$ is a unique string that represents the name of a dimension, $\mathbf{A}$ is the associative array (i.e. a map $K \longrightarrow \mathbb{N}$ where K is a set of keys), and T is the type of the associative array. The schema of a named typed associative array is $Name : K$. Dom_{Name} is the domain of values taken by the keys of $\mathbf{A}$, i.e., a subset of K. A typed tensor $\boldsymbol{\mathcal{X}}$ is a tuple $(Name, D, V, T)$ where $Name$ is the name of the tensor, D is a list of named typed associative arrays, i.e., one per dimension, V is the values of the tensor and T is the type of the tensor, i.e., the type of its values. The schema of a typed tensor is $Name(S) : T$ where S is the list of schemas of its dimensions, i. e., associative arrays of D. More strictly and by analogy with the relational model, the formal schema of a tensor is the list of names of dimensions to which the name of the tensor is added. For example, the typed tensor $\boldsymbol{\mathcal{UHT}}(User : String, Hashtag : String, Time : Long) : Long$ with the dimensions $User$, $Hashtag$ and $Time$ is used to store the number of times a hashtag is used in tweets produced by a user per time slice.

With its straightforward mapping to diverse data models, TDM is a useful pivot model for polystore architectures. These mappings are presented in detail in [24]. To summarize the main ideas, we focus on the mapping from the most popular data models to TDM:

- **Relational and column**: The mapping from a relation R to typed tensors produces a set of tensors $\boldsymbol{\mathcal{X}}_i$ where the dimensions D are the n attributes that form together the key of R and for the $k - n$ remaining attributes we create a tensor for each. The keys of each D are formed of different values of each attribute domains.
- **Key-value**: most of key-value stores save data as $(key, value)$ pairs in a distributed hash table. As typed tensors are described by associative arrays, set of $(key, value)$ pairs in NoSQL stores are mapped to a 1-order tensor, with the dimension storing the keys.

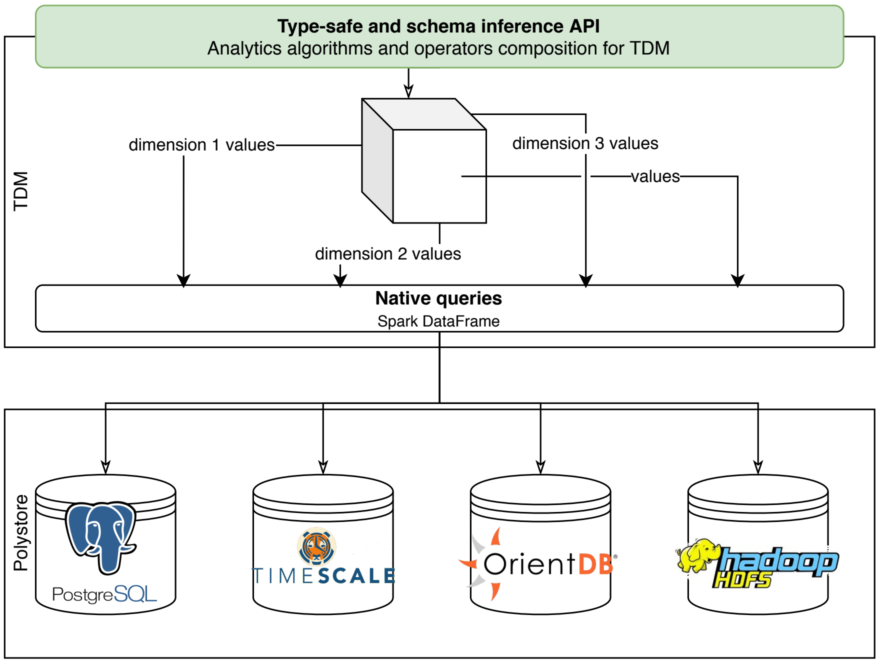

Figure 1: Role of TDM in a polystore system

- **Graph**: a graph can be represented by an adjacency matrix, i.e. a 2-order tensor representing one of the matrix (e.g. adjacency, Laplacian) which describes the graph.

The definition of typed tensor and tensor schema are the starting point to formally define data manipulation operators on tensors. We have specified projection, selection on the values of tensor, restriction on values of dimension, union, intersection, nesting and natural join. These operators respect the closure property thus allow to define compositions of operators as queries over multiple data sources (i.e. a polystore). By adding difference and Cartesian Product the set of operators is relationally complete [12]. We invite the reader to look at [24] for a formal definition of the operators, that is based on the formalism of [18] used to define the relational model. Our definitions are constructed with two levels:

- a description of the operator's behavior on the schema, i.e., restrictions on operand schemas and specification of the result schema;
- a specification of the operator's semantics on values.

Due to a lack of space we give an outlook of some operators (figure 2 and table 1) in order to illustrate the two levels on which the operators are working on (i.e., the schema level with constraints on the associative arrays values and the value level). The selection, restriction, union and intersection work only at the value level, while the projection, nesting and natural join work at both the value and the schema level.

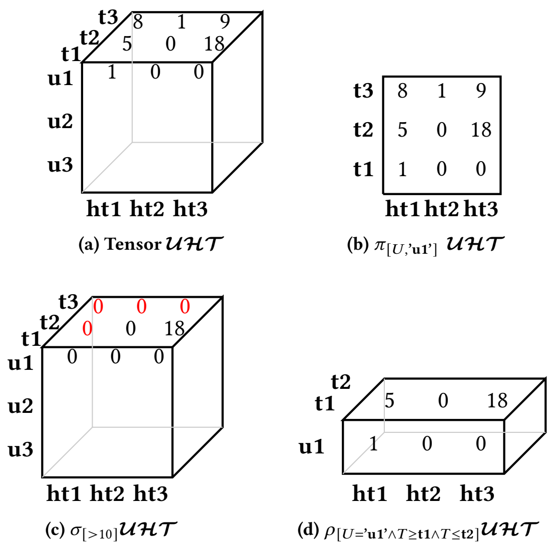
t3 8 1 9
t2 5 0 18
t1 1 0 0
u1
u2
u3
ht1 ht2 ht3
(a) Tensor $\mathcal{UHT}$
t3 8 1 9
t2 5 0 18
t1 1 0 0
ht1 ht2 ht3
(b) $\pi_{[U,\text{'u1'}]}\ \mathcal{UHT}$
t3 0 0 0
t2 0 0 18
t1 0 0 0
u1
u2
u3
ht1 ht2 ht3
(c) $\sigma_{[>10]}\mathcal{UHT}$
t2 5 0 18
t1 1 0 0
u1
ht1 ht2 ht3
(d) $\rho_{[U=\text{'u1'}\wedge T\geq\text{t1}\wedge T\leq\text{t2}]}\mathcal{UHT}$

Figure 2: Projection, selection and restriction operators applied to the tensor $\mathcal{UHT}$ for the values of u1 (other values are not shown in the example)

3.2 Theoretical Complexity

The storage of tensors depends on the nature of data, nevertheless representing and analyzing Big Data with tensor models produces sparsity. For example, in the graph theory, a graph $G = (V, E)$ is considered to be sparse if $|E| = O(|V|)$ resulting in sparse adjacency matrix representation. Compressed Sparse Column (CSC) or Compressed Sparse Row (CSR) are common data structures used to represent sparse matrices [11]. They can be applied directly to tensors, if we consider tensors as set of matrices [9] obtained by unfolding operations. These representations have been extended

Operator	Signification	Equivalent Relation Algebra Expression	Complexity
$\pi_{[expr]}\boldsymbol{\mathcal{X}}$	projection on some specific value of a dimension, reduce the order by removing the dimension on which the projection is applied	$\pi(\{d_1, ..., d_N\} - d, \boldsymbol{\mathcal{X}})\sigma(d = c)T_{\boldsymbol{\mathcal{X}}}$	$O(1 + \mu_{d,c})$
$\sigma_{[expr]}\boldsymbol{\mathcal{X}}$	selection of values, can reduce the set of values in dimensions	$\sigma(expr)T_{\boldsymbol{\mathcal{X}}}$	$O(1 + nz_{\boldsymbol{\mathcal{X}}})$
$\rho_{[expr]}\boldsymbol{\mathcal{X}}$	reduce a dimension to only values that match the expression, can also reduce the values in other dimensions	$\sigma(expr)T_{\boldsymbol{\mathcal{X}}}$	$O(1 + nz_{\boldsymbol{\mathcal{X}}})$
$\boldsymbol{\mathcal{X}}_1 \cup_\theta \boldsymbol{\mathcal{X}}_2$	union of two tensors having the same schema and perform the θ operation on values having the same keys	$(T_{\boldsymbol{\mathcal{X}}_1} \cup T_{\boldsymbol{\mathcal{X}}_2} - \pi(d_i^{\boldsymbol{\mathcal{X}}_1}, \boldsymbol{\mathcal{X}}_3)T_{\boldsymbol{\mathcal{X}}_3}) \cup \pi(d_i^{\boldsymbol{\mathcal{X}}_1}, \boldsymbol{\mathcal{X}}_1\theta\boldsymbol{\mathcal{X}}_2)T_{\boldsymbol{\mathcal{X}}_3}$ with $T_{\boldsymbol{\mathcal{X}}_3} := \sigma(d_i^{\boldsymbol{\mathcal{X}}_1} = d_i^{\boldsymbol{\mathcal{X}}_2})(T_{\boldsymbol{\mathcal{X}}_1} \times T_{\boldsymbol{\mathcal{X}}_2})$	$O(2 + nz_{\boldsymbol{\mathcal{X}}_1} + nz_{\boldsymbol{\mathcal{X}}_2})$
$\boldsymbol{\mathcal{X}}_1 \cap_\theta \boldsymbol{\mathcal{X}}_2$	intersection of two tensors having the same schema and perform the θ operation on values having the same keys	$\pi(d_i^{\boldsymbol{\mathcal{X}}_1}, \boldsymbol{\mathcal{X}}_1\theta\boldsymbol{\mathcal{X}}_2)T_{\boldsymbol{\mathcal{X}}_3}$	$O(2 + nz_{\boldsymbol{\mathcal{X}}_1} + nz_{\boldsymbol{\mathcal{X}}_2})$
$\boldsymbol{\mathcal{X}}_1 \bowtie \boldsymbol{\mathcal{X}}_2$	join of two tensors having at least one dimension in common and keep the values of the first tensor	$\pi(\bigcup_{\substack{j=1,2 \\ i=1,...N^{\boldsymbol{\mathcal{X}}_j}}} d_i^{\boldsymbol{\mathcal{X}}_j}, \boldsymbol{\mathcal{X}}_1)T_{\boldsymbol{\mathcal{X}}_3}$	$O(\sum_{d \in D^{\boldsymbol{\mathcal{X}}_2}} \lvert Dom_d^{\boldsymbol{\mathcal{X}}_2}\rvert + nz_{\boldsymbol{\mathcal{X}}_1})$

Table 1: TDM operators with their meaning, expression in relational algebra and their complexity using a sparse representation with hashtables for values of dimensions

to sparse tensors with Compressed Sparse Fiber (CSF) format [36]. CSR, CSC and CSF are effective only for some operators such as multiplication and thus are not suitable for supporting the variety of TDM operators.

We will consider two different hypotheses, the first one is the storage of tensors as tuples or elements of a dataframe, the second one, more suitable for in memory storage, is the extension of Knuth structure for sparse matrices [20] (p.302-306) to tensors using hash tables to have direct access to elements sharing the same value on a dimension. The table 1 gives, at the third column, for each operator its cost of execution as a relational algebra expression for the first hypothesis and at the last column its theoretical complexity for the second hypothesis. Notations are the followings: $nz_{\boldsymbol{\mathcal{X}}}$ is the number of existing values in tensor $\boldsymbol{\mathcal{X}}$, $d \in D^{\boldsymbol{\mathcal{X}}}$ is one of the dimension of a tensor, $Dom_d^{\boldsymbol{\mathcal{X}}}$ is the domain, i.e. the set of values for dimension d of a tensor $\boldsymbol{\mathcal{X}}$, $\mu_{d,c}^{\boldsymbol{\mathcal{X}}} = nz_{\boldsymbol{\mathcal{X}}}/|Dom_d^{\boldsymbol{\mathcal{X}}}|$ is an estimation of the number of elements in a sub-tensor when a dimension d is set to a specific value, $T_{\boldsymbol{\mathcal{X}}}$ is a set of tuples in a representation of tensor using relational table or dataframe.

4 STRONGLY TYPED COMPOSITION OF OPERATORS AS A FUNCTIONAL QUERY LANGUAGE

This section presents the mechanisms that allow to leverage the type-safe and schema inference properties. As we put in evidence in sections 1 and 2, these properties are of primary importance for the manipulation and the transformation of data. Once established, they make it possible to build a functional query language based on the composition of TDM operators, possibly including other analysis operators. TDM is developed in Scala, as this language has facilities to establish the mechanisms needed, partially thanks to its strong statically typing system.

4.1 Type-safe and schema inference

In order to have type-safe and schema inference properties in TDM, several mechanisms are needed. In this subsection, we outline phantom types, the shapeless library and implicits, that are the three components of our seeked properties.

Phantom types are types that can never be instantiated. They are used to apply constraints over type, without the need of creating a new object. Their use helps to propagate the type-safe functionality, by allowing the compiler to use these types to check more precise constraints directly over types.

In the TDM library, tensor's dimensions are defined as phantom types, that extends *TensorDimension[T]* with a given type *T*:

```
object User extends TensorDimension[String]
object Hashtag extends TensorDimension[String]
object Time extends TensorDimension[Long]
```

By doing so, several properties are given to dimensions: 1) each dimension can have a meaningful name, while being of a simple type (such as *String* or *Long*); 2) each dimension is easily identifiable, because it is referenced with a type rather that just a name (e.g. a *String* or an object instance); 3) tensor's dimensions can be controlled more finely, by accepting only once each phantom type but multiple time the same simple type (e.g. for a tensor representing coordinates, two phantom types are used: Longitude and Latitude, each extending *TensorDimension[Double]*); 4) multiple tensors can share a same phantom type, and use it as a constraint to apply operators, thus enforcing the type-safe capability at the schema granularity, with the help of implicits (see below).

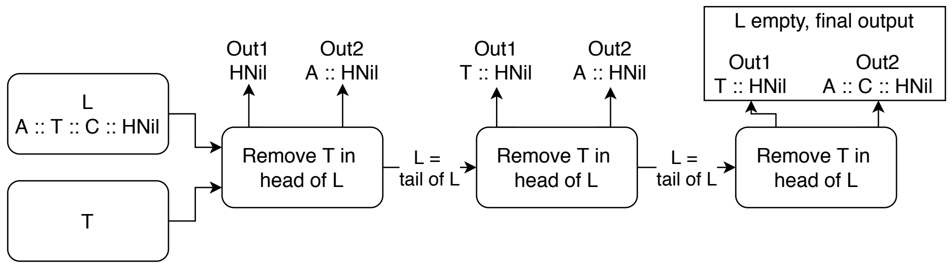

Figure 3: Example: removing all elements of type *T* in a *HList L* with the help of implicits and path dependent type

With **the shapeless library**[4], some structures to manipulate objects at type level are available. It is the case for the *HList* (Heterogeneous List), that allows to build a list with different types, while keeping the detail of each type and not using a common supertype to work with the list.

In the TDM implementation, a *HList* of phantom types (e.g. *User::Hashtag::Time::HNil*) is used to represent the schema of a tensor, and the type of the tensor is made of a parameterized type. This sort of implementation provides us with tools and methods to interact with the schema of a tensor, that can be triggered with implicits.

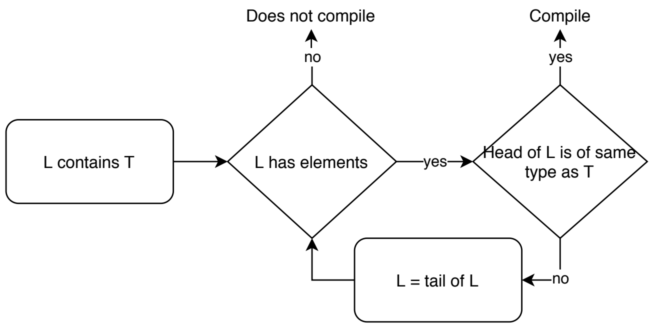

Figure 4: Steps to check if a type *T* is in a *HList L* with the help of implicits

In Scala, **implicits** [30, 31] are a mean to delegate some code logic to the compiler. When a function defines a parameter as implicit, if the user does not provide explicitly the parameter, the compiler will check in the current scope if it is possible to find a value with the corresponding type to use it for the function call.

This functionality can be pushed to be used to check advanced constraints, or to infer a result type depending on input parameters. If we want to verify that a type *T* is in a *HList L*, we can use implicits in a recursive way (figure 4). We first check if the head of *L* is of the same type as *T*, if it is the case, the implicit compiles because it found a correspondence in its scope. If the head of the list is not of the same type as *T*, the implicit must resolve itself the same implicit, this time called on *T* and on the tail of *L*. If no correspondence is found when reaching the end of *L*, the whole chain of implicits does not compile, thus invalidating the function that used the implicit at the root of the call. By composing these types of constraints, we can build more complex ones, and **use them to enforce validity of operators depending on the schema**.

Implicits can also be used to **infer automatically the schema** of the resulting tensor when applying an operator that alters the schema of the current tensor (e.g. projection, join). For this, implicits are combined with path dependent types [5], that allow to compute a type given one or multiple types in input. The implicit application is the same that for the constraint, but we add the expected output type in the implicit call, and when resolving the implicit, the output type is built. An example of this use is to remove all elements of a given type in a *HList* (figure 3). For an input *HList L*, and a type *T*, we can build two output types: one (*Out1*) as the *HList* containing all the elements of type *T* that were a part of *L*, and one (*Out2*) as the *HList* containing all the elements of *L* except those of type *T*. This removal can be used to apply a projection operator on a tensor, as it removes the dimension on which we want to focus on. The schema of the tensor resulting of the execution of this operator will depend of the schema of the input tensor and of the dimension on which we want to do the projection.

4.2 Towards a functional query language

Scala is a language of choice to develop TDM, because it is statically typed and allows to strongly check type constraints at compile-time (see section 4.1). Our implementation of TDM (available at https://github.com/AnnabelleGillet/TDM) is based on Spark and uses the shapeless library that enables the development of dependent type based generic programs. Above Spark's *DataFrame* we add schema, data manipulation operators and tensor decompositions to implement TDM in a type-safe way. TDM implementation is user-friendly: it hides all the shapeless details from the user and detects errors at compile-time.

A TDM tensor is built in three steps: 1) dimensions are defined; 2) these dimensions are added to the tensor (and can be reused in other tensors) and 3) values are obtained by querying a data source or manually added. For example, to build the tensor $\mathcal{UHT}$, its three dimensions are created as:

```
object User extends TensorDimension[String]
object Hashtag extends TensorDimension[String]
object Time extends TensorDimension[Long]
```

Notice that dimensions defined in this way, on top of providing properties defined in section 4.1, are also used to help users to produce values for a dimension and to express conditions for data

[4]https://github.com/milessabin/shapeless

manipulation operators. Dimensions are then used with an object *TensorBuilder*, from which a tensor of type *Long* is instantiated:

```
val tensorUHT = TensorBuilder[Long]
    .addDimension(User)
    .addDimension(Hashtag)
    .addDimension(Time)
    .build()
tensorUHT.addValue(User.value("u1"), Hashtag.value("ht1"),
     Time.value(1))(1)
```

Alternatively, a tensor can also be created by retrieving directly its values from a data source. In this case, users must supply:

(1) a *Properties* object embedding information of connection to the *TensorBuilder*;
(2) the mappings between dimensions and the name of each attribute returned by the query and;
(3) the query to execute and the name of the attribute which contains the tensor's values.

```
val props = new Properties()
...
val query = """SELECT user_screen_name AS user, hashtag,
     published_hour AS time, COUNT(*) AS value
  FROM tweet t INNER JOIN hashtag ht ON t.id = ht.tweet_id
  GROUP BY user_screen_name, hashtag, published_hour HAVING
     COUNT(*) > 5 """
val tensorUHT = TensorBuilder[Long](props)
    .addDimension(User, "user")
    .addDimension(Hashtag, "hashtag")
    .addDimension(Time, "time")
    .build(query, "value")
```

This mechanism allows all storage systems for which a JDBC driver is available to be used as a data source. By using the Spark's data access layer, data sources without a JDBC driver can also be added easily.

TDM operators add data manipulation capabilities to tensors and infer automatically the schema of the resulting tensor, even when the operator induces a schema transformation, e.g. for the natural join. For example, using a projection on the previously defined tensor $\mathcal{UHT}$ for the dimension *User* and the value $u1$ will yield a new tensor $\mathcal{HT}$ populated with the values corresponding to the dimension and the value specified:

```
val tensorHT = tensorUHT.projection(User)("u1")
```

With the naming possibilities of Scala, we can also call the projection operator as:

```
val tensorHT = π(tensorUHT)(User)("u1")
```

Result of this operator can be shown using access to tensor values with the following expressions:

```
tensorHT(Hashtag.value("ht1"), Time.value(1)) // Some(1.0)
tensorHT(Hashtag.value("ht2"), Time.value(2)) // None
```

Other operators can associate two tensors in various ways. For example, the union produces a new tensor with all values from both tensors, and intersection produces a new tensor with only common values between the two tensors. Both operators take a function in parameter that determines the function to apply when keys are in common between tensors.

```
val t1 = tensor1.union(tensor2)((v1,v2) => max(v1,v2))
val t2 = tensor1.intersection(tensor2)((v1,v2) => v1 + v2)
```

The natural join allows to merge schema of two tensors when they have at least one dimension in common. The values kept are those of the first tensor for which the keys combination exists in the second tensor.

```
val t3 = tensor1.naturalJoin(tensor3)
```

TDM operators are implemented with strong type constraints, which can be grouped in three categories: 1) for operators that work at the tensor value level such as selection, the parameter of the condition used must match the tensor value type, 2) for operators working on dimensions that need a dimension parameter, such as the projection or the restriction. The dimension parameter used has to be a part of the schema of the tensor and 3) for binary operators such as union, intersection, natural join or difference. The schema of both tensors must match according to the operators, e.g., for the union and the intersection, the schema of tensors have to be the same and for the natural join the tensors must have at least one dimension in common.

For the internals, TDM uses Spark's *DataFrame*, that correspond to our first hypothesis in section 3.2, and the benefit is multiple: 1) working with a well defined and scalable structure, and 2) keeping the optimization capabilities of Spark. The *DataFrame* is used with n-1 columns for the dimensions' values, and the last column for the value of the tensor associated to these dimensions' values.

The type-safe guarantee is obtained at compile-time by combining shapeless and Scala's implicits, as explained in section 4.1. A compilation-error warns the user if an inconsistency is detected. The use of implicits gives also the capability to define custom compilation-error messages that fit the tensor context, in order to avoid unclear default messages. The following example shows inconsistencies detected at compile time[5]:

```
tensorHT.addValue(Hashtag.value(1), Time.value(2))(2.0) //
     Wrong type of dimension's value
tensorHT.addValue(Hashtag.value("ht2"))(2.0) // Wrong number
      of dimensions
tensorHT.projection(User)("u2") // Dimension not in tensor
tensorHT.union(tensorUHT) // Different schemas of tensors
```

By using the different mechanisms (phantom types and implicits), we showed how the implementation of TDM supports type-safe and schema inference properties. The method used to build tensors by using native queries takes advantage of each database of the polystore, and thus allows to perform fine grained optimizations during the construction with specific techniques such as the bind join.

[5]See also https://github.com/AnnabelleGillet/TDM/tree/master/src/test/scala/tdm/core for examples of detected inconsistencies

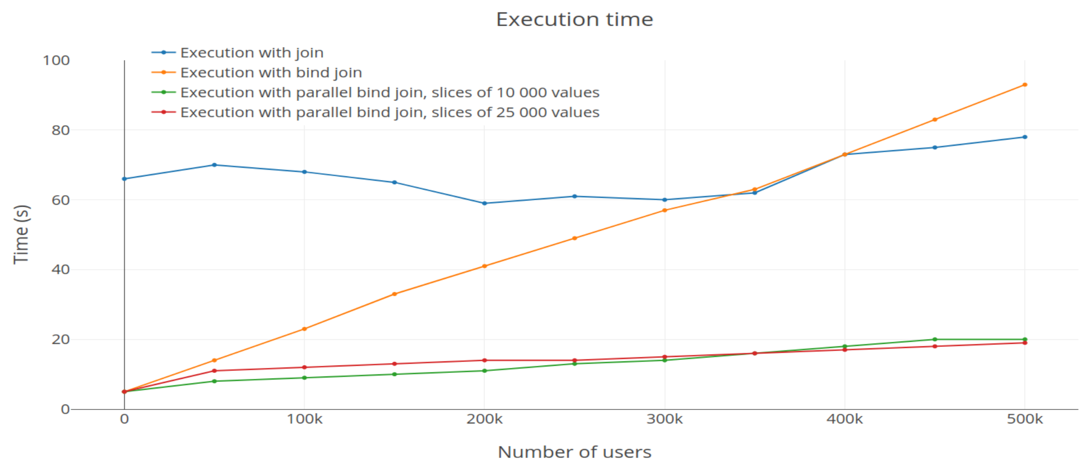

Figure 5: Execution time for a join, a bind join and a parallel bind join, with slices of 10 000 and 25 000 values

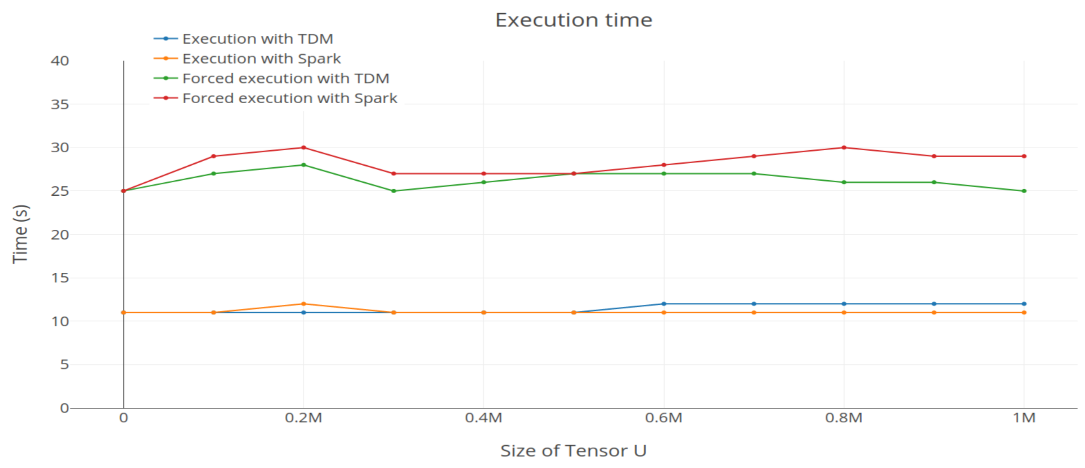

Figure 6: Execution time with TDM and Spark

5 BIND JOIN OPTIMIZATION AND TDM OVERHEAD STUDY: EXPERIMENTS AND RESULTS

In this section, we present two kind of experiments: 1) to show, through a bind join, the benefit of exploiting a polystore and knowing the characteristics of data, and 2) to execute the same combination of operators with TDM and with Spark, in order to see if TDM produces some overhead compared to Spark. The experiments were performed on a Dell PowerEdge R740 server (Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz, 20 cores, 256Go RAM).

For the first experiment, we compare a naive join against a bind join that takes advantage of some knowledge about the data. A bind join is an optimisation technique to join a smaller and a bigger sets, where the values of the smaller set are collected and directly sent to the bigger set, in order to minimize data transfers and to limit the bigger set only to values that can actually match with the smaller set. For a set of users $\mathcal{U}$, we want to get $\mathcal{T}$ containing the tweets that a subset of the users have published. These data are stored in a PostgreSQL database, in a table **user** that have 1M elements, and in a table **tweet** that have 50M elements. To see the evolution of the execution time, the subsets of **user** start from 0 to 500 000 elements, with steps of 50 000, and the average execution time of five repetitions is kept.

First, we perform a naive Spark join between $\mathcal{T}$ and each subset of $\mathcal{U}$ (the blue line on figure 5). Then, as we know that $\mathcal{U}$ is smaller than $\mathcal{T}$, we perform a bind join: we only retrieve the values of $\mathcal{T}$ that match those of the subset of $\mathcal{U}$, rather than trying to join all the values. This time, we obtain the orange line in figure 5. The execution time is significantly better than the naive join when the subset of $\mathcal{U}$ is smaller than 350 000 values. Second, we can push further the exploitation of the knowledge of the data: we know that only one user can have published a tweet, so for two distinct subsets of $\mathcal{U}$, two distinct subsets of $\mathcal{T}$ will be matched. We can split the subset of $\mathcal{U}$ in several slice, run each small bind join against $\mathcal{T}$ in parallel, and then perform an union on all the results and still get the expected result. The green and red lines of figure 5 represent this parallel bind join, with slices of 10 000 and

25 000 values respectively. As we can see, the execution time of our optimization technique is significantly better than the naive and the bind join. At the end of the curve, the bind join with slices of 10 000 values takes a little more time than the one with 25 000 values: it can be explained by the multiplication of the unions that have to be made between each slice, and by the number of queries that have to be sent to the database, when the processor does not have enough cores to handle all the queries at the same time.

With this experiment, we show that a good knowledge of the data and of the behaviour of the database can significantly improve performances compared to a uniform and naive approach.

For the second experiment, we compare an execution with Spark and with TDM. Spark is a well-known and powerful analytics engine, so, an important objective of developing TDM as a layer over Spark is to keep performance equivalent to Spark while taking advantage of the capabilities of TDM. To estimate the eventual overhead induced by TDM, we run an experiment in Spark and compare the result with the same experiment using TDM. The different phases are: 1) building a tensor $\mathcal{U}$ with the users and the number of tweets published by each user as tensor's values, 2) building a tensor $\mathcal{UHT}$ with the number of hashtags published by users for 1h time slices, 3) performing a selection on tensor $\mathcal{U}$ to keep only users who have published at least 100 tweets, 4) joining $\mathcal{U}$ and $\mathcal{UHT}$ to keep the values of $\mathcal{UHT}$ only for active users. We vary the size of tensor $\mathcal{U}$ from 0 to 1M elements by steps of 100 000.

This experiment is carried out in two cases: first by forcing the computation at each operation, and second by forcing the execution only at the last operation, in order to witness the optimization of Spark. The executions are repeated five times, and the average time is measured. A real anonymized data set is available on the github of the experiment [6]. As we can see in figure 6 the execution with TDM does not induce overhead compared to Spark, and the optimization capabilities of Spark are preserved (bottom of fig. 6).

6 CONCLUSION

TDM is a tensor based pivot data model that bridges the gap among data sources and analytics frameworks with a unification of the different theoretical foundations of data models (graph, matrix, relation). The type-safe property and the closure of the operators set are major prerequisites for Big Data analytics. Our library demonstrates that tensors can be manipulated in a safer way, and are well-suited for a data centric use with well-defined data manipulation operators.

The type-safe and schema inference properties of TDM library are implemented by constraints carried out by parameterized types and Scala's implicits. They allow to detect schema inconsistencies and incompatibilities of parameters in operator expressions at compile time. TDM operators allow to build complex expressions over tensors, which can be used as a traditional query language. TDM goes beyond Spark *DataFrame* by providing a layer that keeps the performance of Spark and does not induce overhead, as shown by the comparative experiment between Spark and TDM.

[6]https://github.com/AnnabelleGillet/TDM-experiments/tree/master/SparkComparison

We now focus on developing advanced tensorial operators and algorithms such as hierarchical tensor decomposition, as well as integrating data manipulation operators available for Spark *DataFrame*s that we can use on tensors. We are also studying the capabilities of TDM to take advantage of each database of a polystore depending on the operation optimizations allowed by its model. We plan to develop a mechanism that could optimize the functional queries, by using a context provided by an expert user in order to guide the evaluation depending on the database, as we showed in the particular case of the bind join.

ACKNOWLEDGMENT. This work is supported by ISITE-BFC (ANR-15-IDEX-0003) coordinated by G. Brachotte, CIMEOS Laboratory (EA 4177), University of Burgundy.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283, 2016.
[2] A. Agrawal, R. Chatterjee, C. Curino, A. Floratou, N. Gowdal, M. Interlandi, A. Jindal, K. Karanasos, S. Krishnan, B. Kroth, et al. Cloudy with high chance of DBMS: A 10-year prediction for Enterprise-Grade ML. In *Conference on Innovative Data Systems Research (CIDR)*, 2020.
[3] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, et al. Theano: A Python framework for fast computation of mathematical expressions. *arXiv:1605.02688*, 2016.
[4] R. Alotaibi, D. Bursztyn, A. Deutsch, I. Manolescu, and S. Zampetakis. Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1660–1677, 2019.
[5] N. Amin, T. Rompf, and M. Odersky. Foundations of path-dependent types. *ACM SIGPLAN Notices*, 49(10):233–249, 2014.
[6] P. Barceló, N. Higuera, J. Pérez, and B. Subercaseaux. On the Expressiveness of LARA: A Unified Language for Linear and Relational Algebra. *arXiv preprint arXiv:1909.11693*, 2019.
[7] P. Baumann. Management of multidimensional discrete data. *The VLDB Journal*, 3(4):401–444, 1994.
[8] R. Brijder, F. Geerts, J. Van den Bussche, and T. Weerwag. MATLANG: Matrix operations and their expressive power. *ACM SIGMOD Record*, 48(1):60–67, 2019.
[9] A. Buluc and J. R. Gilbert. On the representation and multiplication of hypersparse matrices. In *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11, 2008.
[10] T. Chen. Typesafe abstractions for tensor operations. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, pages 45–50. ACM, 2017.
[11] A. Cichocki, R. Zdunek, A. H. Phan, and S. Amari. *Nonnegative matrix and tensor factorizations: applications to exploratory multi-way data analysis and blind source separation.* John Wiley & Sons, 2009.
[12] E. Codd. Relational completeness of data base sublanguages. *Computer*, 1972.
[13] R. C. Fernandez, P. R. Pietzuch, J. Kreps, N. Narkhede, J. Rao, J. Koshy, D. Lin, C. Riccomini, and G. Wang. Liquid: Unifying Nearline and Offline Big Data Integration. In *Conference on Innovative Data System Research (CIDR)*, 2015.
[14] V. Gadepally, P. Chen, J. Duggan, A. Elmore, B. Haynes, J. Kepner, S. Madden, T. Mattson, and M. Stonebraker. The BigDAWG Polystore System and Architecture. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2016.
[15] V. Gadepally, J. Kepner, W. Arcand, D. Bestor, B. Bergeron, C. Byun, L. Edwards, M. Hubbell, P. Michaleas, J. Mullen, et al. D4M: Bringing associative arrays to database engines. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2015.
[16] P. Griffioen. Type inference for array programming with dimensioned vector spaces. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages*, page 4. ACM, 2015.
[17] H. Jananthan, Z. Zhou, V. Gadepally, D. Hutchison, S. Kim, and J. Kepner. Polystore mathematics of relational algebra. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 3180–3189. IEEE, 2017.
[18] P. C. Kanellakis. Elements of relational database theory. In *Formal models and semantics*, pages 1073–1156. Elsevier, 1990.
[19] J. Kepner, V. Gadepally, H. Jananthan, L. Milechin, and S. Samsi. AI Data Wrangling with Associative Arrays. *arXiv preprint arXiv:2001.06731*, 2020.
[20] D. Knuth. *The art of computer programming. Vol. 1: Fundamental algorithms.* Addison-Wesley, 1978.

[21] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[22] B. Kolev, O. Levchenko, E. Pacitti, P. Valduriez, R. Vilaça, R. Gonçalves, R. Jiménez-Peris, and P. Kranas. Parallel polyglot query processing on heterogeneous cloud data stores with LeanXcale. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 1757–1766. IEEE, 2018.

[23] J. Kossaifi, Y. Panagakis, A. Anandkumar, and M. Pantic. Tensorly: Tensor learning in python. *The Journal of Machine Learning Research*, 20(1):925–930, 2019.

[24] É. Leclercq, A. Gillet, T. Grison, and M. Savonnet. Polystore and Tensor Data Model for Logical Data Independence and Impedance Mismatch in Big Data Analytics. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XLII*, pages 51–90. Springer, 2019.

[25] L. Libkin, R. Machlin, and L. Wong. A query language for multidimensional arrays: design, implementation, and optimization techniques. In *ACM SIGMOD Record*, volume 25, pages 228–239. ACM, 1996.

[26] Z. H. Liu, J. Lu, D. Gawlick, H. Helskyaho, G. Pogossiants, and Z. Wu. Multi-model database management systems-a look forward. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*, pages 16–29. Springer, 2018.

[27] J. Lu and I. Holubová. Multi-model databases: a new journey to handle the variety of data. *ACM Computing Surveys (CSUR)*, 52(3):1–38, 2019.

[28] D. Mišev and P. Baumann. *SQL Support for Multidimensional Arrays.* IRC-Library, Information Resource Center der Jacobs University Bremen, 2017.

[29] T. Muranushi and R. A. Eisenberg. Experience report: Type-checking polymorphic units for astrophysics research in Haskell. In *ACM SIGPLAN Notices*, volume 49, pages 31–38. ACM, 2014.

[30] M. Odersky, L. Spoon, and B. Venners. *Programming in scala.* Artima Inc, 2008.

[31] B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. *ACM SIGPLAN Notices*, 45(10):341–360, 2010.

[32] E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos. Tensors for data mining and data fusion: Models, applications, and scalable algorithms. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(2):16, 2017.

[33] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[34] S. Rabanser, O. Shchur, and S. Günnemann. Introduction to tensor decompositions and their applications in machine learning. *arXiv preprint arXiv:1711.10781*, 2017.

[35] A. Rush. Tensor Considered Harmful. Technical report, Harvard NLP, 2010.

[36] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis. Splatt: Efficient and parallel sparse tensor-matrix multiplication. In *IEEE International Parallel and Distributed Processing Symposium*, pages 61–70, 2015.

[37] M. Stonebraker and U. Cetintemel. " one size fits all": an idea whose time has come and gone. In *21st International Conference on Data Engineering (ICDE'05)*, pages 2–11. IEEE, 2005.

[38] J. Sun, D. Tao, and C. Faloutsos. Beyond streams and graphs: dynamic tensor analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 374–383. ACM, 2006.

[39] R. Tan, R. Chirkova, V. Gadepally, and T. G. Mattson. Enabling query processing across heterogeneous data models: A survey. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 3211–3220. IEEE, 2017.

[40] S. Van Der Walt, S. C. Colbert, and G. Varoquaux. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22, 2011.

[41] P. Vassiliadis and A. Simitsis. Near real time ETL. In *New trends in data warehousing and data analysis*, pages 1–31. Springer, 2009.

[42] M. Zaharia and B. Chambers. *Spark: The Definitive Guide.* O'Reilly Media, 2018.