Design of a Portable Programming Abstraction for Data Transformations

Johannes Luong, Dirk Habich and Wolfgang Lehner
Database Systems Group, Technische Universität Dresden, 01062 Dresden, Germany

Keywords: Database Programming Languages, System Integration, Parallel Programming Models, Data Analyses.

Abstract: Novel data intensive applications and the diversification of data processing platforms have changed data management significantly over the last decade. In this changed environment, the expressiveness of the traditional relational algebra is often insufficient and data management systems have started to provide more powerful special purpose programming languages. However, these languages create a tight coupling between applications and specific systems that can hinder further development on both sides of the equation. The goal of this article is to start a discussion on the future of platform independent programming models for data processing that re-establish the separation of application logic and implementation details that used to be a cornerstone of data management systems. As a guide for that discussion, we introduce several recent related works on that topic and also outline our own contribution, the *Analytical Calculus*.

1 INTRODUCTION

Over the last decade, two important drivers of specialization and adaptation have shaped the landscape of data processing. On the one hand, new data heavy applications, such as advanced statistical analyses, have created the need for new expressive programming models that exceed the capabilities of the traditional relational algebra. On the other hand, nonfunctional requirements on data processing, such as data volume or low latency, have resulted in the creation of a large number of special purpose systems and libraries that achieve very high performance in particular scenarios. In the early days of the *one size does not fit all* era (Stonebraker et al., 2007), those two forces have been reconciled in a mostly ad-hoc fashion where individual applications use low-level data processing APIs to implement their business logic. Unfortunately, this ad-hoc reconciliation has created a strong binding between applications and special purpose processing systems that can hinder further development on both sides. Applications are bound to specific system level APIs and can not be migrated to better technology without significant rewrites and system providers have to guarantee backwards compatibility in order to maintain the good will of their customers. Furthermore, system level programming requires specialized knowledge and experience which increases the cost of using these technologies.

The desire to provide an easy to use high level programming interface and to separate application logic from system level details is certainly well known in the database community. In database systems these problems have been solved with the widespread adoption of SQL and the relational algebra as programming abstraction. SQL is an abstract declarative language with operations for the selection, combination, filtering, and aggregation of relational datasets. The semantics of these operations are defined on the abstract data type *Relation* and SQL does not stipulate any further nonfunctional constraints on possible implementations. Database system providers have used the strong physical abstraction provided by SQL to create a diverse set of relational database management systems with widely varying nonfunctional properties. Application developers write their business logic using the relatively easy to use abstract SQL interface and have the freedom to chose an adequate database system later on.

The relational model provided an excellent solution until the first driver of specialization, a new set of popular data heavy applications, has created the need for a more flexible and expressive approach to data-oriented programming. In the *big data* community, this need was first answered by flexible but system specific low-level APIs. But once the analyses of large datasets had become more widespread, the drawbacks of system level programming became ap-

Luong, J., Habich, D. and Lehner, W.
Design of a Portable Programming Abstraction for Data Transformations.
DOI: 10.5220/0006945004000408
In *Proceedings of the 7th International Conference on Data Science, Technology and Applications (DATA 2018)*, pages 400-408
ISBN: 978-989-758-318-6

parent very soon and popular systems, such as Apache Hadoop[1], Apache Spark[2] or TensorFlow[3] began to introduce higher level languages[4][5][6] that make programming of these systems much more accessible. Although these languages succeed in simplifying the use of *big data* platforms, most of them are special purpose solutions that bind applications to a particular system. Further, some of these languages still expose low level system details, such as explicit caching of intermediate results, explicit repartitioning of data, and so forth.

Recently, several authors have recognized and discussed the problems of system specific programming models and leaky abstractions in data processing. Jennie Duggan and colleagues (Duggan et al., 2015) discuss issues that arise when multiple independent data processing systems have to be integrated to work on a common goal. They propose the BigDAWG polystore system which introduces a unified query interface and organizes data movements between participating systems. Ionel Gog and colleagues (Gog et al., 2015) observe that several *big data* cluster processing systems use quite similar internal program representations that can be mapped onto each other in an automated fashion. Based on this finding they are able to decouple high-level front end languages from the languages native execution environments and make programs in those languages portable on all supported engines. Shoumik Palkar and colleagues (Palkar et al., 2017; Palkar et al., 2018) investigate a similar scenario where an application uses multiple data processing libraries and data movement between those libraries becomes a bottleneck. They propose *Weld*, a low-level in-memory storage and processing system that libraries can use to access and share datasets. Holger Pirk and colleagues (Pirk et al., 2017) look at the problem at a slightly different angle and investigate a lower level programming model that can be automatically mapped to efficient multicore parallelism, vector parallelism, as well as GPU programs. Despite the more technical focus, they tackle the same issue of decoupling data processing from low level system details by introducing a language abstraction.

The goal of this paper is to start a discussion on the decoupling data centric applications from processing engines with the intention to achieve greater flexibility, portability, and adaptivity. With this goal in mind, in Section 2 we begin with a detailed discussion of each of the previously mentioned papers. In Section 3 we provide an introduction of some of our own work on this topic and in Section 4 we end the paper with a short conclusion.

[1]http://hadoop.apache.org/
[2]https://spark.apache.org/
[3]https://www.tensorflow.org/
[4]http://pig.apache.org/
[5]http://mahout.apache.org/
[6]https://spark.apache.org/mllib/

2 RELATED WORK

In recent years, several new flexible and portable programming abstractions for data intensive applications have been published. In the following sections we are going to discuss four exemplary papers that provide a good overview of the *practical* work that has been done recently and that showcase important design issues of that space. Our selection is not meant to be comprehensive and we focus exclusively on practical approaches that try to introduce a layer of abstraction between system specifics and applications. We also leave out some important works, such as Alexandrov's comprehension interface for Apache Flink (Alexandrov et al., 2015) or Microsoft's LINQ (Meijer and Bierman, 2011; Yu et al., 2008), because they do not raise significant additional points with regard to our focus.

2.1 BigDAWG

Jennie Duggan and colleagues (Duggan et al., 2015) motivate their *BigDAWG* system with an interesting application scenario that encompasses the use and integration of several specialized data types, such as waveform data, plain text, structured records, and semi structured documents. They argue that each of these datatypes should be stored in a specialized database system to benefit from the superior performance and query interface that a dedicated system can provide. However, effectively accessing four different database systems increases application complexity and achieving overall high performance might entail a significant effort in application specific tweaking and optimization.

To remedy these issues, Duggan et al. propose the *BigDAWG* polystore system. Similar to a federated database, *BigDAWG* provides a unified access interface to multiple database systems. But, in contrast to the federated systems from the 80s and 90s (Chawathe et al., 1994; Carey et al., 1995; Stonebraker et al., 1996), *BigDAWG* supports several data types and query languages instead of just the relational model. A given data model and its corresponding query language, such as relations and SQL, forms a so called *data lake* and each data lake can be backed by multiple database engines that implement that model. For each data lake, *BigDAWG* defines a canonical variant

of the corresponding query language, such as the cannonical SQL variant and so forth. To integrate with *BigDAWG*, engines have to provide a *shim* program that can translate a data lake's canonical language variant into the engine's own query language. For example, a particular relational database system has to be able to translate *BigDAWG*'s SQL into its own SQL dialect and so forth. Users can combine queries of different data lakes using special `SCOPE` and `CAST` operators. The authors provide the following example to showcase this feature:

```
RELATIONAL(
  SELECT *
  FROM R, CAST(A, relation)
  WHERE R.v = A.v
);
```

In this case, `RELATIONAL` is a scope operator that annotates the following program as SQL query and `CAST` converts an array `A` into a relation so it can be used in the relational context. Unfortunately, the authors do not provide any more comprehensive code examples which makes it hard to judge the convenience of combining data lakes in a real world setting.

BigDAWG choses a mostly hands-off approach to query languages. For the most part, it reuses existing database languages and simply forwards queries to implementations of the respective data lakes. This approach has several benefits that, in theory, make *BigDAWG* easy to adopt and easy to extend. First, users can easily pick up the system and integrate it with their existing software as they don't have to adopt a new language. Second, the reuse of existing languages allows *BigDAWG* to rely on the query optimization that is provided by many existing database engines. Third, in principle, the hands-off approach also facilitates the straight forward incorporation of a broad set of data lakes into the *BigDAWG* polystore, because interactions between lakes are minimal. However, each data lake still requires a canonical query language and a shared data format. The canonical language acts as a common denominator for all possible implementations and therefore has to be conservative with language features. This might especially hinder the integration of data lakes that do not offer a widely accepted query language, such as vector query languages or flexible UDF oriented interfaces.

2.2 Musketeer

Ionel Gog and colleagues (Gog et al., 2015) investigate *big data* cluster processing engines, such as Hadoop, Spark, or Naiad (Murray et al., 2013), and the high-level query languages that these engines provide. The authors find that different engines show widely different performance characteristics for the same workload and conclude that it is advisable to chose an engine based on the specific task at hand. Further, the authors also find that the engine specific high-level languages for relational and graph workloads create a certain *lock in* effect that prevents users from switching engines once they have mastered its languages.

Gog et al. solve these issues with their *Musketeer* system which decouples high-level languages from their native runtimes and can automatically select an adequate processing engine for a given workload. Conceptually, *Musketeer* implements a modern compiler architecture where a set of frontend languages are translated into a common internal representation and the internal representation is compiled into executable code for a particular runtime environment. But instead of generating executables, *Musketeer* produces workloads for big data processing engines and is even able to split a program into partial workloads that are executed on different engines. The internal representation is a data flow language that is especially designed for parallel data processing. The data flow language provides a set of typical data-parallel operators, such as `MAP`, `GROUP BY`, `JOIN`, and `AGG`, but also a dynamic `WHILE` operator for iterative algorithms.

Musketeer provides parsers for Hive SQL[7], Lindi (Murray et al., 2013), and the authors' own *BEER* DSL that supports relational and graph processing. It can create workflows for Hadoop[8], Spark[9], Naiad (Murray et al., 2013), PowerGraph (Gonzalez et al., 2012), GraphChi (Kyrola et al., 2012), and Metis (Mao et al., 2010) and for testing purposes it can also generate serial C code. When *Musketeer* receives a new program it first translates it, using the adequate frontend, into the data flow representation. SQL like languages can be easily mapped to the internal representation because the relational algebra is a data flow language with data parallel operators itself. Even more, *Musketeer* provides operators, such as `GROUP BY`, `JOIN`, or `AGG`, which more or less directly implement relational semantics. Graph processing, on the other hand, has a somewhat different processing model, making the translation more complicated and the resulting intermediate representation shares little resemblance with the input program. In general, there are several different graph processing models but in this article the authors only consider the popular *Gather Apply Scatter* model that is also

[7] https://hive.apache.org/
[8] http://hadoop.apache.org/
[9] http://spark.apache.org/

used by systems like Pregel (Malewicz et al., 2010). In this model, processing happens as a sequence of uniform steps and each step consists of three phases: 1) each node of the graph *gathers* incoming messages from its neighbours, 2) each node updates its internal state using an update function, and 3) each node scatters outgoing messages to its neighbours. The authors do not go into detail on how they translate graph programs but one possible approach would create an internal data flow program that *groups* messages by node IDs, *joins* the message groups with node states, and *maps* over each *(messages, state)* tuple to generate new messages and node states for the next processing step.

A similar mismatch between processing models arises at the other end of the translation when internal programs are mapped to executable workloads. Processing engines, such as Hadoop, Spark, or Naiad, use parallel data flow languages themselves and *Musketeer* operators can be mapped to those languages in a straight forward manner[10]. Dedicated graph engines on the other hand, usually implement a graph specific processing model like *Gather Apply Scatter* which can neither express all *Musketeer* data flow programs in a straight forward manner nor efficiently execute those programs. The authors solve this issue by introducing the notion of *code idioms*. A code idiom is a certain well known data flow pattern that can be easily recognized by program analyses and mapped to a different processing model like *Gather Apply Scatter*. To make this approach work, frontends have to be careful to actually generate the appropriate data flow patterns, otherwise the backend will not be able to detect the idioms and can not target graph engines.

In contrast to *BigDAWG*, *Musketeer* does not chose a hands off approach to query languages but defines its own internal processing model. The data flow model is broadly applicable and can express typical query languages in a natural way. However, the model is also more generic than some of the processing engines that the authors want to address. This makes it necessary to add a meta language of idioms which can capture the semantics of certain compositions of data flow operators.

2.3 Weld

Shoumik Palkar and colleagues (Palkar et al., 2017; Palkar et al., 2018) investigate modern analytics applications and how they make use of external libraries such as Pandas[11] or NumPy[12]. In contrast to database systems that usually apply sophisticated query optimization, these libraries are often not able to perform any kind of cross function optimizations, especially if the functions belong to different libraries. To improve the performance of analytics libraries, Palkar et al. propose *Weld* an in-memory data store for shared memory systems that offers a flexible processing oriented query interface. If libraries are adapted to use *Weld* as storage and low-level processing backend, the system can perform important optimizations, such as pipelining or loop fusion, across function calls and across libraries. To achieve this cross function behaviour, *Weld* implements a lazy query evaluation approach, where computations are only executed once their results are actually required. Applications and libraries use the *Weld* runtime API to allocate memory objects and to perform data parallel operations on those objects. The result of these operations is an abstract handle that can be passed to the host application and subsequently to other *Weld* enabled libraries. Only when a library has to return an actual value will it force evaluation of the handle and the *Weld* runtime can chose an optimized plan to perform the requested transformations.

Weld offers a flexible functional query interface that revolves around parallel loops and a set of *builders*. The data model includes primitive scalars, structures, vectors, and dictionaries. Loops are used to process individual elements of vectors or dictionaries and builders are used to aggregate values. Listing 1 shows a basic sample query. Some of the builders include:

- `appender[T]` which appends values of type `T` to a vector of type `vector[T]`
- `merger[T, func, id]` which aggregates values of type `T` using an associative function `(T, T)` $\rightarrow$ `T` and an identity value
- `vecmerger[T, func]` which inserts values of type `(Int, T)` into a vector at a specific position using `func` to merge the new value with the previous value at that position.

Weld makes the bold choice to ignore existing query languages and instead proposes its own flexible language that has close ties to monad and monoid comprehensions (Grust, 2003; Fegaras and Maier, 1995). In their 2018 paper (Palkar et al., 2018), the authors demonstrate that *Weld* can be integrated into popular libraries, handle real world application scenarios, and significantly improve overall performance in many cases. In contrast to the previously discussed papers,

[10] With the exception of the `WHILE` operator which sometimes has to be handled in an external driver program.

[11] https://pandas.pydata.org/

[12] http://www.numpy.org/

Listing 1: Basic Weld example.

```
// vector literal
in := [1, 2, 3];

// appender creates a new vector
a  := appender[int];

// for returns its builder
// merge inserts into a builder
q  := for(in, a,
        (in, v) => merge(in, v*v)
      );

// returns [1, 4, 9]
result(q);
```

Weld binds its programming model to a particular execution environment which limits the portability of applications that decide to use the system. However, shared memory performance is an important use case that is often overlooked in the design of modern analytics programming models and the authors clearly demonstrate the importance of optimized low-level machine access. What is more, we have good reasons to believe that the *Weld* programming model could be ported to other data engines without too much effort as well.

2.4 Vodoo

Similar to *Weld*, Holger Pirk and colleagues (Pirk et al., 2017) investigate a programming model for data intensive processing on shared memory systems. However, their work is less focused on creating a convenient processing environment for libraries and end users, but rather on providing a portable way to address different types of hardware parallelism, such as multi core parallelism, SIMD vector instructions, or GPU programming environments. *Vodoo* is a programming environment and runtime system that allows users to easily tune their programs for different hardware scenarios. That is, users define their application logic in an abstract data flow language but annotate the abstract logic with additional information on how the data is to be partitioned and distributed at runtime. These annotations are used by *Vodoo* in a predictable manner, to decide whether an operator should use a multi threaded implementation, a SIMD vector implementation, or a GPU implementation and how each of these possibilities is configured in detail. In contrast to the previously discussed models, this approach gives users fine grained control over how their applications are executed.

Listing 2 shows a basic *Vodoo* example that calculates the sum of a vector of doubles. A program of a typical data flow language would probably only consist of the first and last line of the program and leave it to a runtime or compiler to decide how to best compute the sum over a vector. *Vodoo* however offers much more explicit control over the desired execution strategy. In particular, the input data is first partitioned into batches of size 1024 that are aggregated independently before computing the overall result. This particular configuration will result in a multithreaded execution strategy where *Vodoo* assigns data batches to a set of worker threads. However, by simply changing the calculation of the IDs to

```
// assign values to 2 SIMD lanes
IDs        := range(data) % 2
```

the execution strategy can be changed into to a SIMD based implementation, where four values are added in parallel using vector instructions.

Listing 2: Basic Vodoo example.

```
data       := load("DoubleVec")

// create batches of size 1024
IDs        := range(data) / 1024
positions  := partition(IDs)
partitions := scatter(
                zip(data, IDs),
                positions
              )

// compute the sum of each batch
partialSum := foldSum(
                partitions.val,
                partitions.id
              )

// compute the overall sum
sum        := foldSum(partialSum)
```

Vodoo offers an interesting approach to application portability with regards to low-level parallelism. The system makes it easy to explore implementation variants and facilitates hardware specific tuning of applications. Similar to GPU programming models, *Vodoo* code is supposed to be compiled lazily, *just in time* when the results are needed. This approach enables very dynamic changes to execution strategies even during runtime of an application. Unfortunately, the authors limit their discussion to the implementation of traditional relational databases and do not explore richer semantics at the moment. We believe, that the approach could be of good use in a broader environment as well.

3 THE ANALYTICAL CALCULUS

The separation of logic and execution in data intensive applications is an important theme in our own research as well. In a recent paper (Luong et al., 2017), we have introduced the *Analytical Calculus* which is our own proposal for a flexible, rich, and portable programming interface for data analytics. The *Analytical Calculus* is a lighweight pure functional language with a small core library of abstract data types for parallelized data processing. The language has a simple static type system and contains few constructs besides functions and function application. In the current version, recursion is prohibited but we plan to enable certain important recursion patterns (Meijer et al., 1991) in future versions. A deliberate choice of core types and the explicit support of domain specific concepts allow the *Analytical Calculus* to support a wide range of application scenarios. At the same time these properties also allow the *Analytical Calculus* to drive a large number of execution strategies that scale from fast shared memory systems, such as *Weld*, to large systems of systems, such as *BigDAWG*. The *Analytical Calculus* enables adaptivity in data management by separating application logic from application execution and by facilitating flexibility on both sides of that separation. This flexibility can be exploited, for example, by a dynamic runtime system that quickly adapts physical execution strategies to changed workloads or data characteristics.

Similar to the relational algebra, the *Analytical Calculus* is not meant to be written manually, but acts as an intermediate representation that is generated by high-level frontend languages, such as SQL, and that is consumed by a runtime that translates the abstract statements into physical operations. The lack of side effects, the static type system, and the structured recursion simplify code analyses and transformations to a great degree and therefore make the *Analytical Calculus* a very good fit for a flexible internal program representation. Another similarity to the relational algebra is that the *Analytical Calculus* is not designed for a specific runtime system but is supposed to establish a general programming abstraction for data processing that can be implemented by various runtimes. For example, we are currently developing a hybrid runtime that uses the *Analytical Calculus* to drive a system that integrates a PostgreSQL[13] database, a MongoDB[14] database, and a Spark cluster. The *Analytical Calculus* is not designed for a particular frontend language but can serve as intermediate language for a wide set of purposes. However, we are in the process of developing the *ACQL* query language, a superset of SQL that will contain extensions for linear algebra. *ACQL* is the first language for the *Analytical Calculus* and will help us understand the capabilities and limits of our model.

$$\langle BagUnion \rangle (\lambda c\, n.\ \{(c.name, c.phone, n.name)\} \mid c \leftarrow customer,\ n \leftarrow nation,\ \lambda c\, n.\ c.nationkey = n.key)$$

with $BagUnion := (Bags, \cup, \emptyset)$

Figure 1: Join with a monoid comprehension.

3.1 Monoid Comprehensions

The *Analytical Calculus* uses monoid comprehensions (Fegaras and Maier, 1995) as core computational concept to perform tasks such as filtering, transforming, aggregating, and grouping values and to build joins. Figure 1 shows how a monoid comprehension can be used to perform a natural equivalence join and subsequent projection over two relations. The definition consists of two main parts: (i) a monoid and (ii) a comprehension over that monoid. A monoid is an algebraic structure that consists of a set, an associative binary operation over that set, and a neutral element for the operation. In the example, the *BagUnion* monoid consists of (i) the set of all finite bags, i.e. multisets, (ii) the bag union operation, and (iii) the empty bag. A comprehension over a monoid is a function that uses one or several datasources to generate an element of its monoid. For example, the comprehension in Figure 1 uses the two datasource *customer* and *nation* to create a bag of tuples. Comprehensions consist of two parts that are separated by a vertical bar: the head and the tail. The tail is a sequence of *bindings* that bind the elements of a datasource to a variable and *filters* that accept or discard the bindings left of the filter. A tail that contains multiple bindings computes the cross product of all bound sources. The head is a function over the comprehension's bindings that returns an element of the comprehension's monoid. For example, the head function in Figure 1 returns a bag that contains an individual result tuple. The head is applied to each sequence of bindings that is not discarded by one of the filters and all head results are eventually combined using the monoid's binary operation.

Figure 2 shows an example of a monoid comprehension which is defined over the set of integers with the addition as operation and zero as neutral element. The comprehension's tail contains the two bindings n and d which will generate cross product of the two

[13] https://www.postgresql.org/
[14] https://www.mongodb.com/

$$\langle Sum \rangle (\lambda n d.\ \frac{n}{d} \mid n \leftarrow [1,2,3,4,5,6,7,8], d \leftarrow [2,3,4], \lambda n d.\ n \bmod d = 0)$$

with $Sum := (\mathbf{N}, +, 0)$

Figure 2: Aggregation using a comprehension.

sources: $(1,2)$, $(1,3)$, $(1,4)$, $(2,2)$, $(2,3)$, $(2,4)$, ..., $(8,2)$, $(8,3)$, $(8,4)$. The filter accepts all bindings where the second number is a divisor of the first number, such as $(2,2)$, $(3,3)$, $(4,2)$, or $(4,4)$. The head simply returns the fraction of first and second number for each accepted binding tuple and the comprehensions builds the sum of all head results:

$$\frac{2}{2}+\frac{3}{3}+\frac{4}{2}+\frac{4}{4}+\ldots+\frac{8}{2}+\frac{8}{4}$$

This example demonstrates how different monoids can be used to produce various result types. Even greater expressiveness can be achieved by nesting comprehensions which allows, for example, to build groups and outer joins.

We are, of course, not the first to promote the use of comprehensions in data processing systems. Authors like Grust (Grust, 2003) or Fegaras and Maier (Fegaras and Maier, 1995) have pointed out the theoretical benefits of comprehensions, such as expressiveness and strong support of optimizations, a long time ago. Microsoft's *language integrated queries* (Meijer and Bierman, 2011) provide an implementation of comprehensions in a mainstream programming language that can be used to access a variety of data sources in a convenient manner. More recently, Alexandrov and colleagues (Alexandrov et al., 2015) have demonstrated that comprehensions can be nicely mapped to the data flow model of the big data stream processing system *Apache Flink*[15] and in Section 2 we have discussed the *Weld* system which uses a language that is closely related to comprehensions to drive a shared memory system and applies a variety of important optimizations, such as pipelining and loop fusion, to achieve very good performance in that environment. In summary, we are very confident in the usefulness of comprehensions as a flexible and widely applicable basic building block of the *Analytical Calculus*.

[15]https://flink.apache.org/

3.2 Explicit Domain Representation

A monoid comprehension is a generic transformation operator that can be used to express a wide array of important data processing functions. For example the entirety of the relational algebra can be easily mapped onto comprehensions and the same is true for many basic operations of the linear algebra. Even basic path matching in graphs can be achieved using comprehensions. However during this mapping from a special purpose model to the more generic comprehensions, some information can be lost or obfuscated. For example, it might be easy to map a graph analysis into a comprehension representation, but it is much less obvious how to reverse this mapping and decide whether a sequence of comprehensions represents a graph analysis. However, this reverse transformation can be useful for several reasons. First, many special purpose domains, such as the relational algebra, the linear algebra, graph analysis, statistical analysis, and so forth, define domain specific optimization rules that can not be applied conveniently in the comprehension representation. Therefore it would be beneficial to reverse the comprehension mapping and apply the optimizations in the original representation. Second, for some of these special purpose domains there are dedicated processing systems with optimized support for the particular domain, such as relational database systems or graph engines. The goal of the *Analytical Calculus* is to be usable as common intermediate language for all data intensive processing and it should therefore be able to drive these special purpose systems by reversing the mapping of domain logic to monoid comprehensions.

In our discussion of *Musketeer* in Section 2, we have already encountered this issue as well. Gog and colleages (Gog et al., 2015) want to use a generic data flow model to drive special purpose graph processing engines and rely on implicit code idioms to enable the necessary reverse mapping. However, in contrast to the implicit idioms of *Musketeer*, we decided give domain specific functions an explicit representation in the *Analytical Calculus*. For this purpose, we add a set of *domain libraries* to the *Analytical Calculus* that capture domain specific concepts with a set of well known functions. These functions, such as *Select*, *Filter*, *NaturalJoin*, or *GroupBy* are implemented using ordinary language elements of the core *Analytical Calculus* libraries, but their names are visible in the program code and can be used to create domain specific behaviour in optimizations or code generation. In contrast to implicit idioms, these explicit domain functions can give guarantees. For example, the *Filter* function of the *relational library* can guarantee that the provided predicate definition can be translated

Listing 3: The *Analytical Calculus* `filter` function.

```
filter(relation: BagT, pred: FuncT) {
  comprehension(
    relation, // binding
    pred,     // filter
    bagOf,    // head
    bagUnion  // monoid operation
    bagEmpty  // monoid identity
  )
}

main() {
  pred = Func(row: RowT) {
    qty    = row("quantity")
    gt100 = qty > 100
    prc    = row("price")
    gt25   = prc > 25.0
    gt100 && gt25
  }
  li = catalog.sym("lineitem")
  filter(li, pred)
}
```

into a SQL *WHERE* statement and reject incompatible predicates at compile time.

Listing 3 shows a simplified version of the relational `filter` function of the *Analytical Calculus*. The function takes a target relation and a predicate function as arguments and simply forwards those parameters to the `comprehension` function. The relation is used as only binding of the comprehension and the predicate function is used as comprehension filter. The `BagOf` constructor provides the comprehension's head function and the bag union and empty bag constructor define the comprehension's monoid. In the `main` function, `filter` is used to select lineitems that have a quantity greater than 100 and a price greater than 25. In itself, the `filter` function is rather unremarkable. However, the significance of the function lies in its well known name "`filter`" which is visible to analysis, transformations and and code generators. Using that name, it is relatively easy to provide an analysis that checks whether predicate functions can actually be translated into SQL or not.

4 CONCLUSIONS

Over the last decade, novel data intensive applications and the need for scaleable and very fast hardware architectures have reshaped the landscape of data processing. At the beginning of this transition, applications and processing engines were closely coupled into expensive single purpose solutions. Over time, more accessible *big data* systems started to emerge and these systems often provide their own dedicated programming languages to simplify application development. However, these languages create a tight coupling between application and processing system that can hinder further development of both applications and processing engines.

The goal of this article is to start a discussion on the future of processing models for data intensive applications. In the first part of the article we provided an in-depth look at four recent related works: *BigDAWG*, *Musketeer*, *Weld*, and *Vodoo*. The *BigDAWG* polystore system integrates a set of dedicated data processing engines behind a unified query interface that mostly reuses existing query languages and their optimizers. *Musketeer* defines a unified data flow language that can be used to decouple special purpose languages from their native processing engines. *Weld* is a data processing engine for shared memory systems that can be used by data analytics libraries to coordinate and optimize computations and memory access and *Vodoo* is an abstract data processing language and code generator that gives users the ability to easily access different types of hardware parallelism.

In the second part of the article, we have outlined the *Analytical Calculus*, our own proposal for a modern programming model for data processing. The *Analytical Calculus* is a small functional language that uses monoid comprehensions as primary computational abstraction. The *Calculus* is used as an intermediate language that can be used as translation target of high-level frontend languages, such as SQL, and that can drive a wide array of data processing runtimes. The *Analytical Calculus* contains domain specific libraries with well known function names to facilitate domain specific optimizations and to enable code generation for special purpose data processing systems, such as RDBMS.

ACKNOWLEDGMENTS

The authors would like to thank the German Federal Minstry of Education and Research (BMBF) for the opportunity to do research in the VAVID project under grant 01IS14005.

REFERENCES

Alexandrov, A., Thamsen, L., Kunft, A., Kao, O., Katsifodimos, A., Herb, T., and Markl, V. (2015). Implicit Parallelism through Deep Language Embedding. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 47–61.

Carey, M. J., Haas, L. M., Schwarz, P. M., Arya, M., Cody, W. F., Fagin, R., Flickner, M., Luniewski, A. W., Niblack, W., Petkovic, D., et al. (1995). Towards heterogeneous multimedia information systems: The garlic approach. In *Research Issues in Data Engineering, 1995: Distributed Object Management, Proceedings. RIDE-DOM'95. Fifth International Workshop on*, pages 124–131. IEEE.

Chawathe, S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., and Widom, J. (1994). The tsimmis project: Integration of heterogenous information sources.

Duggan, J., Elmore, A., Stonebraker, M., Balazinska, M., Howe, M., Kepner, J., Madden, S., Maier, D., Mattson, T., and Zdonik, S. (2015). The BigDAWG Polystore System. *ACM Sigmod Record*, 44(2):11–16.

Fegaras, L. and Maier, D. (1995). Towards an effective calculus for object query languages. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 47–58.

Gog, I., Schwarzkopf, M., Crooks, N., Grosvenor, M. P., Clement, A., and Hand, S. (2015). Musketeer: all for one, one for all in data processing systems. *EuroSys'15*, pages 1–16.

Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. (2012). Powergraph: distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2.

Grust, T. (2003). Monad Comprehensions: A Versatile Representation for Queries. *The Functional Approach to Data Management*, pages 288–311.

Kyrola, A., Blelloch, G. E., and Guestrin, C. (2012). Graphchi: Large-scale graph computation on just a pc. USENIX.

Luong, J., Habich, D., and Lehner, W. (2017). AL: Unified Analytics in Domain Specific Terms.

Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM.

Mao, Y., Morris, R., and Kaashoek, M. F. (2010). Optimizing mapreduce for multicore architectures. In *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep*. Citeseer.

Meijer, E. and Bierman, G. (2011). A co-relational model of data for large shared data banks. *Communications of the ACM*, 54(4):49.

Meijer, E., Fokkinga, M., and Paterson, R. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. pages 124–144.

Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. (2013). Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM.

Palkar, S., Thomas, J., Narayanan, D., Thaker, P., Palamuttam, R., Negi, P., Shanbhag, A., Schwarzkopf, M., Pirk, H., Amarasinghe, S., Madden, S., Zaharia, M., Palkar, S., Thomas, J., Narayanan, D., Thaker, P., Palamuttam, R., and Negi, P. (2018). Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *PVLDB*, 11(9):1002–1015.

Palkar, S., Thomas, J. J., and Shanbhag, A. (2017). Weld: A common runtime for high performance data analytics. *Conference on Innovative Data Systems Research (CIDR)*.

Pirk, H., Moll, O., Zaharia, M., and Madden, S. (2017). Voodoo -A Vector Algebra for Portable Database Performance on Modern Hardware.

Stonebraker, M., Aoki, P. M., Litwin, W., Pfeffer, A., Sah, A., Sidell, J., Staelin, C., and Yu, A. (1996). Mariposa: a wide-area distributed database system. *The VLDB Journal—The International Journal on Very Large Data Bases*, 5(1):048–063.

Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., and Helland, P. (2007). The End of an Architectural Era (It's Time for a Complete Rewrite). *Vldb*, 12(2):1150–1160.

Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Kumar, P., Jon, G., Gunda, P. K., and Currey, J. (2008). DryadLINQ : A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 1–14.