Architecture of a Multi-domain Processing and Storage Engine

Johannes Luong, Dirk Habich, Thomas Kissinger and Wolfgang Lehner
Database Technology Group, Technische Universität Dresden, 01062 Dresden, Germany

Keywords: Database Architectures, Query Languages, Data Types, Code Generation.

Abstract: In today's data-driven world, economy and research depend on the analysis of empirical datasets to guide decision making. These applications often encompass a rich variety of data types and special purpose processing models. We believe, the database system of the future will integrate flexible processing and storage of a variety of data types in a scalable and integrated end-to-end solution. In this paper, we propose a database system architecture that is designed from the core to support these goals. In the discussion we will especially focus on the multi-domain programming concept of the proposed architecture that exploits domain specific knowledge to guide compiler based optimization.

1 INTRODUCTION

Recent developments in the available amounts of data as well as the number and scope of typical use cases, have created the need for scalable data management systems with a focus on complex analytical processing. In many cases, applications process multiple types of data such as tables, matrices, graphs or unstructured data (Abadi et al., 2014). In addition, a greater variety of people is involved in the overall process of generating, processing, and consuming data. Therefore, usability becomes a major challenge for data management.

In this article, we propose an architecture for a data management system that combines scalable and efficient processing of a variety of data types with an easy to use but flexible programming model. To emphasize our architecture's support for multiple datatypes and related programming models, we will call it the *multi-domain architecture* throughout the article.

The *multi-domain architecture* (figure 1) comprises three layers : the programming interface layer, the translation and optimization layer, and the storage and processing layer. The *storage and processing layer* at the bottom of the architecture defines an efficient and scaleable data management API that operates on a group of physical data formats. The API provides a set of operators for each physical format and an operator orchestration language that is used to compose larger workloads. Most of the operators apply a user defined function (UDF) to stored data objects according to a predefined scalable and efficient data access pattern. Operators that process data using a UDF are called *processing operators*.

In our approach, the key to efficient processing is an extensible compiler framework for domain specific languages. This compiler framework forms the *translation and optimization layer* at the middle of the architecture. It accepts user defined data processing programs as input and translates them into optimized physical workloads, which can be executed by the *storage and processing layer* eventually. Access to statistics and meta data allows the compiler to apply advanced optimizations similar to query optimization in traditional DBMS.

At the top of the architecture, the *programming interface layer* provides a multi-domain programming language for user defined processing tasks. The language consists of domain specific elements such as SQL clauses[1] or linear algebra statements, low-level operators that are mirrored from the *storage and processing layer*, and a slim procedural core language. Additional syntactic elements can be added to the language by extending the *translation and optimization layer* with additional compiler components.

Our architecture allows users to write programs that compose different data domains at the level of individual statements. The compilation framework uses domain specific as well as general rules to optimize these programs and translates them into executable workloads. Finally, the *storage and processing layer*

[1]To fit with our model, SQL syntax has to be adapted slightly.

Luong, J., Habich, D., Kissinger, T. and Lehner, W.
Architecture of a Multi-domain Processing and Storage Engine.
DOI: 10.5220/0006009301890194
In *Proceedings of the 5th International Conference on Data Management Technologies and Applications (DATA 2016)*, pages 189-194
ISBN: 978-989-758-193-9

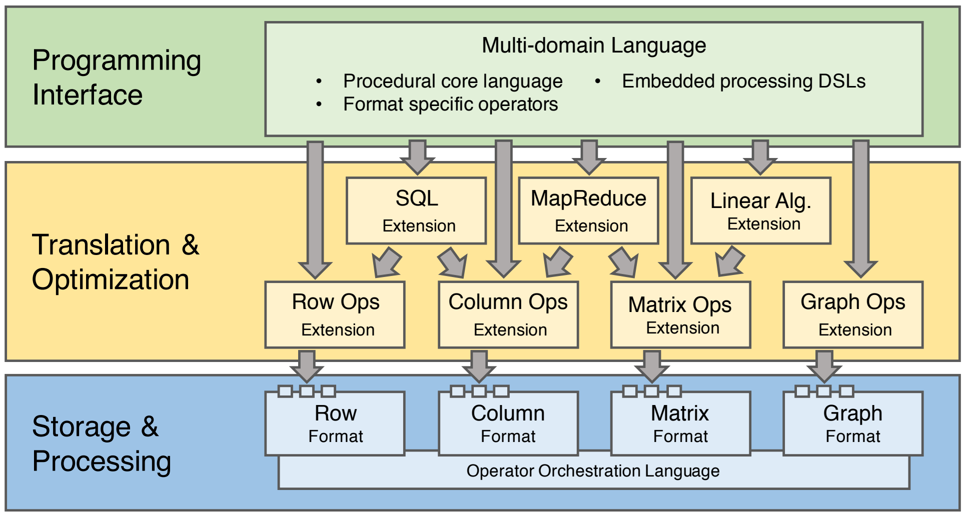


Figure 1: The three layers of the *multi-domain architecture*.

provides a set of optimized physical data formats and processing operators that implement the concrete execution environment of the architecture.

In the following sections we are going to discuss the *multi-domain architecture* in detail. In section 2, we will examine the individual layers and their interfaces. In section 3, we provide a brief review of related work and in the last section we will discuss and summarize our approach and give an outlook on future work.

2 THE MULTI-DOMAIN ARCHITECTURE

The *multi-domain architecture* comprises three main layers. The *programming interface layer* contains the system's multi-domain programming language, the *translation and optimization layer* provides the compiler framework and extensions, and the *storage and processing layer* consists of physical data formats, low-level processing operators, and the operator orchestration language.

2.1 Programming Interface

Users define data processing workloads, using the multi-domain programming language defined in the *programming interface layer*. The language consists of a slim procedural core and a set of domain and format specific language extensions which add their own operators and data types. The procedural core defines primitive types and provides statement composition and scopes (`blocks`), variables, and a set of standard control flow structures for conditional (`if, else`) and repeated (`while, for-each`) execution. The core language is used as a general means for operator orchestration and for the definition of user defined functions (UDF).

Listing 1: SQL language extension example.

```
def query() {
  val customers =
    SELECT("c_name","c_key")
    .FROM("customer")
    .WHERE("c_acctbal" < 0.0)

  val nations =
    SELECT("n_name", "n_key")
    .FROM("nation")

  log(
    SELECT("c_name", "n_name")
    .FROM(customers, nations)
    .WHERE("c_key" == "n_key"))
}
```

Language extensions add new operators and types to the core language. Listing 1 defines a program that uses elements of the relational language extension to define and compose simple SQL-like queries. The extension provides statements such as `SELECT`, `FROM`, and `WHERE` and adds additional definitions for standard operators as < and == that are used when needed.

Listing 2 demonstrates the composition of statements from multiple language extensions. The first six lines use the relational extension to select three columns of the relation `objects` and store them in `C`. Lines 7-14 use operators and types of the linear algebra extension (`*`, `sum`, `Mat` etc.) to create a rotation matrix `R` and to multiply `R` with a transposed copy of `C`. Lines 16-24 use the `ColMap` operator of the matrix format extension to apply a UDF to each column vec-

Listing 2: Multi-domain program.

```
 1 val C =
     SELECT("x", "y", "z")
 3   .FROM("objects")
     .ORDER_BY("id")
 5
   val a = 0.8
 7 val R = Mat(3, 3)(
       cos(a),   0,   sin(a),
 9     0,        1,   0,
       - sin(a), 0,   cos(a)
11   )

13 val Cr = R * transpose(C)

15 val Crn = ColMap(Cr) {
     (col) =>
17     val length =
         sqrt(col(0)*col(0)
19         + col(1)*col(1)
           + col(2)*col(2))
21
       return Mat(3, 1)(
23       col(0) / length,
         col(1) / length,
25       col(2) / length
       )
27 }
```

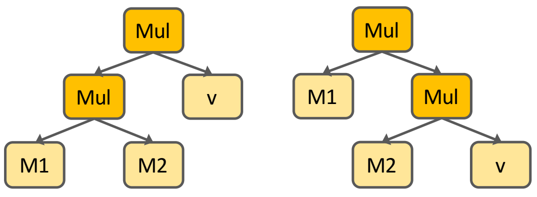
tor of `Cr`.

The `transpose` operation in line 12 is defined on matrices and therefore implies a format transformation of `C` which is initially stored in a table format. Although `ColMap` on line 16 is not defined in the linear algebra extension, but instead in the matrix format extension, it does *not* require a transformation because both extensions operate on the same physical format. In our current model, format transformations are inserted implicitly by the compilation framework and do not require user intervention.

2.2 Translation and Optimization

The efficiency and performance of the *multi-domain architecture* is predicated on the ability of the *translation and optimization layer* to compile multi-domain programs into efficient operator workloads. Domain specific optimizations, efficient use of format transformations, and the generation of code that limits runtime dynamism, are the compiler's main techniques to achieve this goal.

The general compilation process consists of three steps: i) derive an abstract, tree-shaped intermediate representation (IR) from the input program, ii) apply optimization rules to the IR, iii) traverse the IR tree and invoke code generation rules that match the encountered node types.



(a) `(M1 * M2) * v` (b) `M1 * (M2 * v)`

Figure 2: Simple Tree IR optimization.

When a program is transformed into the tree IR, all operator and function calls, as well as their arguments, are turned into IR nodes. Each IR node has a type that identifies the kind of operator, function or object that the node represents. Figure 2 shows two simplified tree IR representations of the linear algebra expression $M1 * M2 * v$. Tree 2(a) is the result of the default left to right evaluation order of the matrix multiplication operator. Code generation for that version would perform the matrix-matrix multiplication first ($res0 \leftarrow M1 * M2$) and the matrix-vector multiplication second ($res1 \leftarrow res0 * v$). On the other hand, linear-algebra domain knowledge tells us i) that vector-matrix multiplications are in general much cheaper than matrix-matrix multiplications, ii) that the result of a matrix-vector multiplication is another vector, and iii) that matrix multiplications are associative. In summary, domain knowledge tells us to perform the matrix-vector multiplication first (tree 2(b)), in order to replace the expensive matrix-matrix multiplication with a cheaper matrix-vector multiplication.

Compilers of general purpose languages do not incorporate domain knowledge for special purpose data structures such as matrices, relations, or graphs. They are therefore unable to perform domain specific optimizations as the one we just described. The compiler framework of the *multi-domain architecture* on the other hand, is explicitly designed to support a specific set of data processing domains and therefore includes the types and operations of these domains as IR nodes. Once these IR nodes are available, pattern matching can be used to define the matrix multiplication optimization and similar rules.

Transformations that require larger parts of the tree as context can be implemented using dedicated IR tree traversals. This technique is especially important during the translation of abstract domain specific operations such as `SELECT, FROM, WHERE` into concrete operations of the *storage and processing layer*. The transition from abstract domain specific to concrete physical operations is carried out, once all domain specific optimizations have been applied.

At that point, a *lowering traversal* replaces domain nodes with operator nodes, in a process that often involves the merging of multiple domain nodes into a single operator node or vice versa the splitting of one domain node into several operator nodes. For example, relational selection and projection nodes can often be merged into a single table scan node, but a single join node requires multiple scan nodes. When all domain specific IR nodes have been translated into physical operator nodes, additional physical transformations can be carried out. Eventually, the IR tree reaches its final form and a last traversal invokes code generation rules for each node to generate the executable *storage and processing layer* workload.

2.2.1 Physical Format Transformations

One aspect that has been left open so far, is physical format transformation. In listing 2 of section 2.1, the linear algebra operation `transpose` is applied to a relation `C`. The relation consists of three numerical columns *X, Y, Z*. Therefore there is a semantically sound transformation of the relation into a 1x3 matrix representation. On a technical level, there are two ways to make this call legal in the statically typed multi-domain language: i) there is an implicit type conversion from relations to matrices and ii) there are multiple overloaded versions of `transpose`, one of which accepts a relation as its input type and possibly also returns a relation as its output.

Option i) implies the existence of dedicated transformation operations that can be represented explicitly in the IR tree. This approach has the advantage that a single set of transformation operations can be reused wherever a type mismatch occurs. The transformation operations can even be targeted in optimization rules and, depending on the implementation, can be potentially merged with other operators. One disadvantage of this approach is that possibly expensive format transformations are carried out independent of the cost of subsequent operations. Sometimes, the performance penalty of an operation that is overloaded for a sub-optimal physical representation might be much smaller than the added transformation overhead.

Option ii) is to overload domain specific operations for multiple physical formats, accepting the degraded performance of some of these implementations. This approach is beneficial in cases where the cost of a format transformation outweighs its benefit. The largest disadvantage of this approach is the necessity to provide multiple implementations for domain operations.

The ideal solution will probably combine both approaches into a single solution. For example, the compiler could use a cost model to decide whether to use a format transformation or an overloaded version of an operator. In general, the topic of format transformations is still an open research question of the architecture and we count on future work to find the best and most practical approach.

2.3 Storage and Processing

The storage and processing layer at the bottom of the *multi-domain architecture* defines the physical data management API. All higher-level domain specific constructs have to be mapped onto this API eventually. The data management API consists of a set of operators that take a number of arguments and an operator orchestration language that is used to compose individual operators into larger workloads.

The first argument of an operator always identifies the data container on which the operator is to be applied. A data container stores the physical data of a single logical data object in a specific data format. For example, a `RowContainer` stores a relation in row format and a `SparseMatrixContainer` stores a matrix in a sparse format.

Some operators, such as *InsertColumnRecord*, perform a simple predefined function on their data container. However, the largest class of operators does not implement a fixed functionality, but instead applies a user defined function (UDF) to a data container. We call these operators *processing operators*. Each *processing operator* implements a well-known, reusable, and efficient data access pattern on its underlying physical format. This is the most important property of *processing operators* and their main benefit compared to classical data retrieval operators that do not combine data access and processing.

The $UnorderedScan(Container, UDF(Record))$ operator, that is defined for both relational formats, is a typical example of a *processing operator*. The operator applies its UDF to batches of records of the target container in arbitrary order. The UDF can not make assumptions regarding which specific records or what number of records will be included in a batch. Typical use cases of the $UnorderedScan$ operator are therefore limited to the context of a single record, as the relational projection and selection operations.

The data access pattern of the $UnorderedScan$ operator can be summarized as: unordered access to individual records. Knowledge of this access pattern enables a scaleable data-parallel operator implementation where UDFs are applied concurrently to multiple batches of records. Other operators provide different guarantees to their UDFs, but the guiding principle is always to enable some form of scaleable data parallel processing.

The ability to achieve implicit data parallelism is the main benefit of known data access patterns, but not the only one. Another benefit is the possibility to fuse multiple equivalent operator calls into a single one. Two *UnorderedScan* operations on the same input container can be fused into a single *UnorderedScan* by simply concatenating the two UDFs.

The remaining component of the storage and processing layer is the operator composition language which is used to combine operators into larger programs. In our current model we use a procedural language for this task. This is a straightforward choice as it enables trivial compilation of the multi-domain core language of the *programming interface layer*. If this choice should turn out to be problematic, we might switch to a more restricted model in the future.

3 RELATED WORK

In the following, we provide a brief discussion of related work on multi-domain processing and generative programming.

3.1 Multi-domain Processing

The need for multi-domain data management systems has been widely recognized. In this article, we propose to tightly integrate multiple storage formats and programing models into a single system. An alternative approach that has been discussed lately is the integration of several DBMS behind a data management middleware layer.

The BigDAWG *polystore* system (Duggan et al., 2015) provides an example for this approach. The authors hide multiple *"of the shelf"* DBMS behind a central management layer, which defines a unified querying interface for all attached systems. The management layer accepts multi domain queries, splits these queries into parts that can be processed by one of the attached DBMS and sends the partial queries to the respective engines. The management layer also handles cases, where one DBMS depends on data that is currently stored in another DBMS and initiates the necessary data transfers.

In contrast to the *multi-domain architecture*, BigDAWG does not need to reimplement data management functionality and can instead reuse proven solutions. What is more, BigDAWG can incorporate DBMS that vary widely in important systems characteristics. BigDAWG could for example attach a relational in-memory DBMS and a classical file based DBMS at the same time and trade off processing speed versus durability guarantees on a per table basis.

On the other hand, data exchange between DBMS is a rather expensive operation as it depends on network communication. Frequent format changes are therefore much more feasible in the integrated single system approach, proposed by the *multi-domain architecture*. Furthermore, BigDAWG implies the administration of multiple separate DBMS, which increases the management cost compared to our integrated approach.

To summarize, the middleware approach provides greater flexibility with regard to non-functional properties but incurs a higher price for format transformations. In addition, administration of the system is more complex.

3.2 Generative Programming

Many big data applications repeatedly execute the same lines of code for millions or billions of data elements. Even expensive optimization becomes viable in that environment as their cost is amortized over time. This realization has sparked interest in runtime code compilation and compiler based optimizations. These two techniques trade additional one time compilation overhead for very efficient code that saves a couple of instructions for every data element.

Beckmann et al. introduce an embedded DSL for C++ that can be compiled, optimized, and executed at runtime of a host program (Beckmann et al., 2004). They use the DSL to write image manipulation kernels that get optimized for specific transformation matrices and generate code that significantly outperforms standard solutions, given large enough image sizes. The primary efficiency gains of the generated code are based on removed indirections and runtime checks that are unavoidable in more general solutions. Newburn et al. follow a very similar approach and provide an embedded C++ DSL that is specifically targetted at data parallelism in multi-core systems (Newburn et al., 2011). They use code generation to specialize performance critical code passages to the specific runtime environment of their programs.

Another group (Alexandrov et al., 2015) uses code generation, domain specific languages, and specific data access patterns to derive implicit parallelism in an approach that is very similar to the one described in this article. On the other hand they are not concerned with multi-domain integration and do not optimize for specialized data formats.

4 SUMMARY AND FUTURE WORK

In this article we have proposed a programming model and system architecture for a data management system that integrates multiple specialized data domains and programming models into an efficient and flexible end to end solution.

In the language layer, at the top of our architecture, we extend a flexible host language with a set of embedded domain specific languages that cover different use cases.

Underneath the language layer, we use a compiler framework to translate abstract multi-domain programs into efficient physical workloads. The framework incorporates domain knowledge which makes it possible to apply powerful domain specific optimization rules. In addition, the compiler generates format transformation code to enable the composition of operations that are defined on different physical representations.

At the bottom of the architecture, we propose to use a storage and processing engine that supports multiple optimized physical data formats. Actual data access is implemented by a set of processing operators that apply user defined functions to data objects according to predefined data access patterns. These patterns enable data parallel operator implementations and advanced optimizations such as operator fusion.

We are currently in the process of developing a prototypical implementation of the proposed architecture. In order to fully focus on the multi domain aspects of our approach, we reuse previous work as extensively as possible possible. Most importantly, we use the in-memory data store ERIS (Kissinger et al., 2014) as the starting point for our multi format data management engine. ERIS uses data parallel processing operators to achieve vertical scaleability on large shared memory multi-processor machines. In addition, ERIS already provides column and row formats for the storage of relations and is therefore designed to support multiple physical storage formats. We plan to extend ERIS and add an additional matrix format to support the important use case of relational and linear algebra integration.

The compilation layer uses the DSL compilation framework LMS (Rompf and Odersky, 2010). LMS uses Scala as host language for embedded DSLs and provides the generation of a tree IR with custom node types. Further, LMS defines a flexible component based extension mechanism, which allows the integration of new DSL elements, IR nodes, optimization rules, and code generation.

We propose the *multi-domain architecture* as our vision for a data management system that provides tight and efficient integration of multiple processing domains. In future work, we want to evaluate the impact of frequent format transformations on the performance of the system and expect to find specific multi-domain optimizations to mitigate these effects.

ACKNOWLEDGEMENT

This work is partly funded by the German Federal Ministry of Education and Research (BMBF) in VAVID project under grant 01IS14005 as well as by the German Research Foundation (DFG) in the Collaborative Research Center 912 Highly Adaptive Energy-Efficient Computing.

REFERENCES

Abadi, D. et al. (2014). The beckman report on database research. *ACM SIGMOD Record*, 43(3):61–70.

Alexandrov, A., Kunft, A., Katsifodimos, A., Schüler, F., Thamsen, L., Kao, O., Herb, T., and Markl, V. (2015). Implicit parallelism through deep language embedding. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 47–61. ACM.

Beckmann, O., Houghton, A., Mellor, M., and Kelly, P. H. (2004). Runtime code generation in c++ as a foundation for domain-specific optimisation. In *Domain-Specific Program Generation*, pages 291–306.

Duggan, J., Elmore, A. J., Stonebraker, M., Balazinska, M., Howe, B., Kepner, J., Madden, S., Maier, D., Mattson, T., and Zdonik, S. (2015). The bigdawg polystore system. *ACM SIGMOD Record*, 44(2):11–16.

Kissinger, T., Kiefer, T., Schlegel, B., Habich, D., Molka, D., and Lehner, W. (2014). ERIS: A numa-aware in-memory storage engine for analytical workload. In *ADMS Workshop at VLDB*, pages 74–85.

Newburn, C. J., So, B., Liu, Z., McCool, M., Ghuloum, A., Toit, S. D., Wang, Z. G., Du, Z. H., Chen, Y., Wu, G., et al. (2011). Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *Code generation and optimization (CGO), 2011 9th annual IEEE/ACM international symposium on*, pages 224–235. IEEE.

Rompf, T. and Odersky, M. (2010). Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Acm Sigplan Notices*, volume 46, pages 127–136. ACM.