D3.1 Language Design Specification

Integrated Data Analysis Pipelines for Large-Scale Data Management, HPC, and Machine Learning

Version 1.2

PUBLIC

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 957407.

Document Description

This report on the language design specification summarizes the design of DAPHNE's domain-specific language (DSL) and related user-facing application programming interfaces (APIs). Similar to the initial system architecture, described in deliverable D2.1, this language design is a result of many discussions with the entire consortium, especially WP2-WP7 because the language abstractions define operational capabilities of the DAPHNE system infrastructure, but also WP8/9 in order to align with the requirements of the use cases and broader benchmarking. The DAPHNE DSL is further closely related to the DAPHNE intermediate representation (IR) and compilation chain. Therefore, this report also summarizes the current DaphneIR, and related compiler design, as well as early ideas on extensibility at language, compiler, and runtime level. Since February 2021, a prototype of the DAPHNE infrastructure is under development, which already implements the core of the presented language design. The initial open source release of the DAPHNE prototype is planned for early 2022, through which we also aim to share the subsequent D3.2 and D3.3 demonstrator deliverables on the compiler prototype.

D3.1 Language Design Specification			
WP3 – DSL Abstractions and Compilation			
Type of document	R	Version	1.2
Dissemination level	PU		
Lead partner	ETH		
Author(s)	Matthias Boehm (KNOW), Ce Zhang (ETH), Patrick Damme (KNOW)		
Reviewer(s)	Benjamin Steinweder (KAI), Marcus Paradies (DLR)		

Revision History

Version	Revisions and Comments	Author / Reviewer
V1.0	Initial structure & outline, discussion w/ all WP partners	Matthias Boehm
V1.1	Initial write-up language abstractions	Matthias Boehm
V1.2	Incorporated review comments and suggestions by Benjamin, Patrick, and Marcus	Matthias Boehm

1 Abstract

DAPHNE aims to provide an open and extensible system infrastructure for integrated data analysis (IDA) pipelines that combine data management (DM) and query processing, high-performance computing (HPC), and machine learning (ML) training and scoring. Although state-of-the-art systems in these areas rely on a variety of different programming paradigms and language abstractions, they share many compilation and runtime strategies and thus, have intrinsic commonalities that ultimately allow us to define common language abstractions and their holistic optimization. Increasing specialization of applications, execution strategies, data types, and the underlying hardware further motivate data independence and extensibility as key design principles. In this report, we introduce the context of existing language abstractions and their limitations, overall design principles, and the initial design of the DAPHNE domain-specific language (DSL), user-facing application programming interfaces (APIs) for bridging the gap to the use cases, as well as the underlying DAPHNE intermediate representation (IR) and compilation chain. In detail, we utilize MLIR [LP+20] as a compiler infrastructure with increasing adoption and devise DaphneIR as a new MLIR dialect. Finally, we describe the initial plans for future extensibility at language, compiler, and runtime level.

2 Introduction

Context and Background: Developing and deploying efficient integrated data analysis (IDA) pipelines is still a major challenge as it requires orchestrating a number of different systems and data formats, and lacks the ability for holistic optimizations across entire IDA pipelines. Together with the increasing specialization of applications, execution strategies, data types, and the underlying hardware, we observe major productivity, overhead, and utilization challenges. First, tuning individual IDA pipelines for emerging hardware and changing data characteristics requires substantial manual effort and is often unsustainable in real-world scenarios. Second, orchestrating IDA pipelines with a variety of specialized systems reduces the effort but causes overhead for boundary crossing (e.g., materialization of intermediates), static resource allocation (temporal and/or spatial under-utilization), and lacks the ability of optimization and redundancy elimination (data and computation) if IDA pipeline primitives are mixed in repetitive or iterative computations. A necessary requirement for an open and extensible system infrastructure for entire IDA pipelines is a common language abstraction.

Existing Language Abstractions: Data management, HPC, and ML systems are all well-established fields with long histories of their language abstractions.

- First, in data management the introduction of the relational model [C70] spawned a variety of query languages like QUEL [SW+76] (based on tuple calculus) and SEQUEL [CB74] (based on relational algebra), which ultimately led to the SQL standard. The success of SQL was driven by its declarative nature (what, not how), flexibility of composing arbitrarily complex queries (closure property), automatic optimization of plans, and physical data independence (applications independent of data organization). Over time, SQL has been extended with good support for user-defined functions (UDF, e.g., in PostgreSQL [S16]), procedural dialects like PL/SQL, TSQL Froid [RP+17], and SQL

Script [BMM13]. Similar to internal execution plans in column stores like MonetDB [MBK00], recently also data-frame operations are widely used for data preparation and query processing, in both local and distributed environments, with systems like SparkSQL [AX+15], Dask [R15], and DuckDB [RM20].

- Second, in HPC the focus was – and largely still is – primarily on custom application codes and optimizing programming language compilers for FORTRAN and C. Major programming models include shared memory abstractions like OpenMP (open multi-processing) and X10 [CG+05,MG+11], as well as message passing like MPI (message passing interface) and related collective operations. Specific applications, e.g., for multi-resolution simulations, also led to more specialized programming models like stencil-based computations. Over time, a rich ecosystem of libraries for numerical computations and simulations like FEM (finite element method) or CFD (computational fluid dynamics) have been developed. A central abstraction, however, are multi-dimensional arrays and related operations, which allow the reuse of highly-tuned, hardware-vendor-provided BLAS and LAPACK libraries.
- Third, ML systems are in comparison to DM and HPC, still in their infancy but rapidly evolving. Language abstractions range from UDF-based systems [HR+12] (in DBMS or data-parallel frameworks), libraries of hand-crafted ML algorithms, over graph-based processing systems, and linear-algebra-based systems [BBY13,BD+16], to more specialized high-level and low-level frameworks for deep neural networks (DNN), model management, and feature-centric tools like DeepDive [SW+15], Overton [Re20], and Ludwig [MDM19]. Major classification dimensions include the language abstractions (operator libraries, algorithm libraries, computation graphs, linear algebra, layers/optimizers), execution strategies, distribution strategies, and underlying data types. Given the approximate nature of machine learning, there is also a wide variety of optimization objectives such as, minimize time subject to memory constraints, minimize monetary cost s.t. time constraints, and maximize accuracy s.t. time constraints.

Despite these very different language abstractions, programming models, and optimization objectives, there is common ground: all of these systems fundamentally work with combinations of data frames (tables with different types per column), and matrices/tensors (multi-dimensional arrays with homogeneous type). Providing the necessary basic functionality and extensibility on top of these abstract data types has the potential to allow a seamless specification and execution of integrated data analysis pipelines.

Design Principles: Before describing the DAPHNE language abstractions in detail, we first want to lay out key design principles that govern the individual design decisions:

- **DP1: Frame and Matrix Operations:** Many ML algorithms, analytical query processing, and numerical computation can be expressed via coarse-grained frame and matrix operations. Recent work on ML-assisted data cleaning [DR18], mapping of tree-based algorithms to matrix operations [NS+20], and even complex enumeration-based ML model debugging via linear algebra [SB21] show its wide applicability. Although such coarse-grained operations might appear restrictive, they preserve the semantics of operations, and tremendously simplify the parallelization and lowering to kernels for

emerging hardware. Overall, we aim to provide a hierarchy of language abstractions for ML algorithms and composite primitives based on frame and matrix operations.

- **DP2: Data Independence:** In contrast to many state-of-the-art ML systems, we aim to follow the success of SQL and provide data independence by default. Instead of requiring users to specify data representations like dense, sparse, and compressed; or data locations like local CPU, local GPU, or distributed, users only work with abstract data types and the compiler and runtime optimize the IDA pipeline for the given data and deployment characteristics. This principle is crucial, especially for libraries of composite DSL-based primitives during whose development the concrete deployment environment and data characteristics are still unknown.
- **DP3: Extensibility:** Data independence and automatic optimizations are great aspirations but face challenges with regard to extensibility for new operations, data types, and hardware. Given the expectation of increasing specialization across the software and hardware stack, DAPHNE's design and language abstractions aim for good extensibility to allow researchers to quickly experiment with new prototypes and extensions of system infrastructure for new specific environments.

Contributions: Following these three design principles, we created an initial design of the DaphneDSL and DaphneLib. In this report, we share the designs of these language abstractions and related compiler, as well as selected implementation details related to both the language abstractions and intermediate representation. The technical contributions include:

- *DSL and API Design:* First of all, in Section 3, we introduce the DAPHNE domain-specific language DaphenDSL including data types, operations, control flow, and additional language abstractions. In this context, we also introduce the DAPHNE Python API (DaphneLib) and parsers for other DSLs and embedded language abstractions like SQL.
- *Compiler Design:* Subsequently, in Section 4, we discuss the MLIR-based compilation chain and the mapping of DaphneDSL into executable runtime plans. This discussion also includes an overview of major compiler components.
- *Hooks for Extensibility:* Given the design principle of extensibility, in Section 5, we then discuss plans for extensibility at DSL, compiler, and runtime level in order to allow researchers to experiment with new operations, data types, optimizations, and emerging hardware devices in end-to-end IDA pipelines.
- *Related Work:* Finally, in Section 6, we summarize related projects and systems that also focus on compiler or system infrastructure for more complex IDA and ML pipelines, as well as aspects of extensibility.

3 APIs, DSL and Language Abstractions

The overall system architecture, including the goals of the DaphneLib and DaphneDSL, has been laid out in Deliverable 2.1 [D2.1]. Figure 1 shows this overall architecture. From a user perspective, the DaphneDSL – as a domain-specific language for data management, HPC, and ML training/scoring – is the main entry point, but additional APIs are provided to allow a seamless integration into typical workflows and environments of data scientists, ML and data researchers, as well as experts of specific application domains. In this section, we describe the

detailed designs of the DaphneDSL and DaphneLib (a Python API with lazy evaluation), as well as means for integrating other DSLs and application libraries.

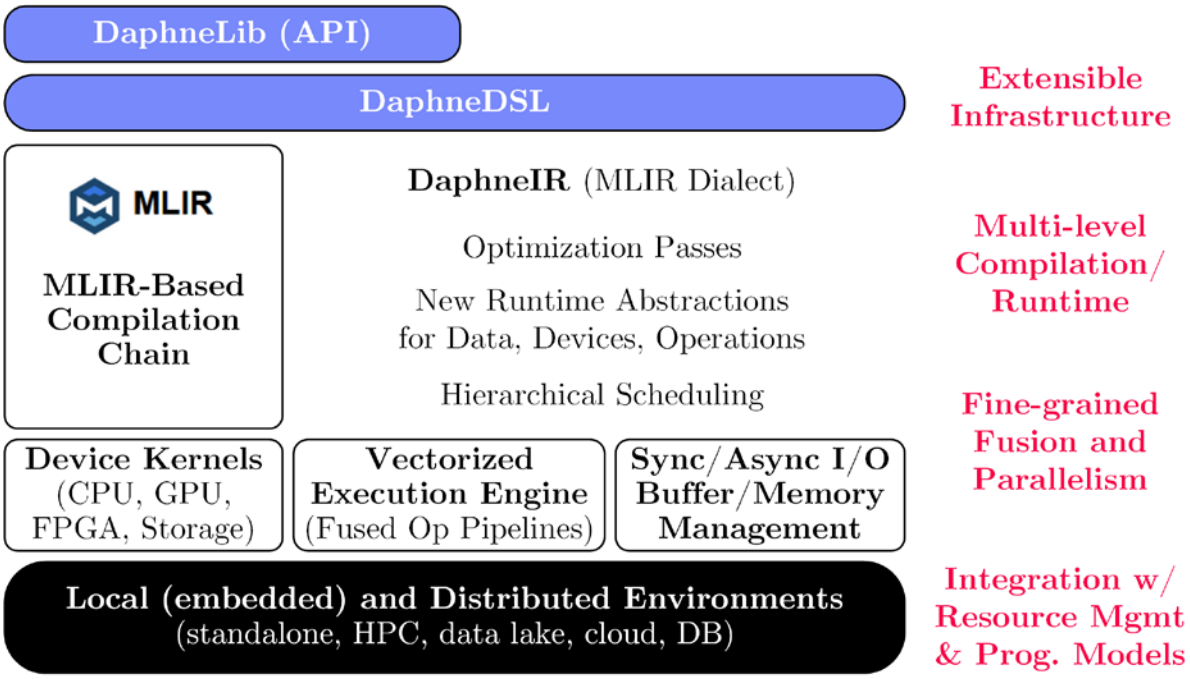

Figure 1: DAPHNE System Infrastructure [D2.1].

3.1 DaphneDSL: A Domain-specific Language

Overview: The DaphneDSL is a domain-specific language inspired by ML systems as well as languages and libraries for numerical computation like Julia [BK+12], Python NumPy [HM+20], R [MH+12], and SystemDS DML [BA+20]. At a high-level, this DSL supports conditional control flow, typed and untyped functions; abstract data types of frames, matrices, and scalars; various built-in operations (i.e., functions and operators), and additional second-order language abstractions. A user creates a simple text file with a DSL script (e.g., `example.daphne`) and can parse, compile, and execute this script via

```
./build/bin/daphnec example.daphne
```

In order to parse this user script `example.daphne` into DaphneIR (DAPHNE's intermediate representation), we use ANTLR4, for which we provide a DSL grammar file and generate the respective parser in an offline manner on grammar updates. In the future, we will further simplify this invocation to `'daphne example.daphne'`. The DaphneIR representation is then the input to the DAPHNE compiler and runtime as described in Section 4.

Data and Value Types: DaphneDSL differentiates data types and value types. Supported data types includes frames (a table with columns of potentially different value types), matrices (homogeneous value type) and scalar values, but in the future we will likely extend this by tensors and named/unnamed lists to group instances of such data types and access by name and/or position (e.g., `params["lr"]`, or `params[7]`). Value types specify the representation of

individual values and currently we support: SI8, SI32, SI64, UI8, UI32, UI64, FP32, FP64 (various integer and floating point representations). In addition, we also support strings but currently only for scalars. The combination of data and value types gives already powerful data representations such as Matrix<FP32> and Frame<SI32, SI8, FP64>.

Basic Built-in Operations: The supported frame and matrix operations include both relational algebra and linear algebra as well as various aggregation and statistical functions. In detail, the list of operations includes 100s of operations of the following categories:

- Matrix multiplications, and various decompositions/solvers
- Elementwise operations (e.g., unary, binary, ternary, n-ary)
- Aggregations and statistical functions (e.g., sum(), rowSums(), colSums(), median())
- Indexing and reorganization (e.g., transposition, extraction and insert, reshaping)
- Deep neural network layers and optimizers
- Set operations with set and multiset semantics, Cartesian product
- Selection and extended projection (with arithmetic operations)
- Joins (inner, outer, theta, semi, anti), and group-by aggregation
- Deduplication, sorting, renaming, and casting
- Read and write of common formats (e.g., csv, matrix market, parquet, arrow, hdf5)

Example DSL Program: For instance, the following DaphneDSL program computes the connected components (connected subgraphs) of a co-author graph.

```
G = readCOO("./AuthorCOO.csv");  // n-by-n Boolean matrix
n = nrow(G);   // get the number of rows (i.e., number of vertexes)
maxi = 1000;
c = seq(1, n); // initialize n-by-1 matrix of vertex IDs (1 through n)
diff = inf;    // initialize diff to +Infinity
iter = 1;

// iterative computation of connected components
while(diff>0 & iter<=maxi) {
  u = max(rowMaxs(G * t(c)), c); // propagate to neighbors, take new max
  diff = sum(u != c);            // number of changed vertex states
  c = u;                         // update vertex assignment
  iter = iter + 1;
}
```

We read a CSV file in coordinate format of row indexes, column indexes, and values (ones) into an expanded (likely sparse) matrix representation, where each row and column refers to an author (i.e., vertex or node in the graph) and non-zero cells refer to a co-author relationship. We then initialize the state of each vertex with a unique ID and iteratively propagate the current state to all neighbors (here, `G*t(c)` performs a matrix/row-vector elementwise multiplication with broadcasting of the transposed vector). The new vertex states are computed as the maximum IDs received from neighbors and current state. That way, the maximum vertex ID per component propagates through the entire subgraph, and once a fixpoint (no more changes) is reached, we terminate and obtain the assignment of nodes to connected components.

Control Flow and Function Calls: In order to enable complex user programs, we also support conditional control flow with loops, branches, and function calls. The basic control flow

constructs are mapped to the existing MLIR dialect SCF (structured control flow), but we are not limited to its components. First, regarding loops, we support for, while-do, and do-while constructs, but will further add parfor (parallel for) loops [BT+14] as a hint for parallelization strategies (see extensibility). Second, branches use the classic if-elseif-else syntax. Both loops and branches allow for arbitrary nesting levels. Third, functions are one of the oldest and most powerful abstractions of computer science. In order to build a hierarchy of DSL-based primitives, we aim to support both typed and untyped functions. For example, consider wrapping the above example DSL program for connected components into a function `components()` and making it available as a registered built-in function.

```
def components(G, maxi, verbose) { ... }

def components(Matrix G, int maxi, bool verbose) { ... }

def components(Matrix<UI1> G, UI32 maxi, UI1 verbose) { ... }

def components(Matrix<UI1> G, UI32 maxi, UI1 verbose)
  return (Matrix<UI64>) { ... } // one output matrix of UI64 value type
```

These alternative specifications allow for very good flexibility (untyped functions are compiled and specialized on demand according to types at a call site) as well as typing (for data and/or value types) if the types are known during development. For example, the given connected component algorithm assumes Boolean input graphs (1 bit integers) and can make this explicit. Note that we took inspiration from Python type hints (e.g., `def foo(x:T) -> T:`) and Julia type assertions (`function foo(x::T)::T`). Multiple function returns also require multi-assignments such as `X,Y = foo(Z)` for a function that returns two results and binds them to X and Y, respectively. Function call arguments can be passed by position (e.g., `u = components(myGraph,100,1)`) or name (`u = components(G=myGraph)`), were the latter allows arbitrary argument orderings and defaults. For both DSL-based functions and built-in operations, we aim to allow, similar to Julia [BC+18], multiple dispatch where function calls are dispatched to the most specific type combination of inputs. In contrast to general-purpose programming languages, we only allow top-level function declarations together with namespaces, which allows the packaging of function libraries without conflicts.

Scoping and Type Polymorphism: Related to conditional control flow, we use bounded scoping from traditional programming languages as shown below in the example on the left. If an intermediate variable (e.g., X) is created in a certain nested scope, it is deleted at the end of this scope. This scoping is in contrast to R's unbounded scoping, which is useful due to missing variable declarations. However, via simple matrix/frame constructors, as shown below in the example on the right, we can easily overcome the need for type declarations. Variable shadowing is not supported, so an assignment of X overwrites the outer scope's variable but in a function-local manner (no side effects on global variables outside the function).

```
if( sum(Y) > 0 ) {
  X = Y @ Z; // matmult
  print(sum(X));
}
// delete X on exit
```

```
X = matrix(0, 0, 0) // needed
if( sum(Y) > 0 )
  X = Y @ Z1;
else
  X = Y @ Z2;
```

Furthermore, the DaphneDSL has copy-on-write semantics by default, where assignments like `A = B`, and function calls are copy-by-reference, but any modification like `B[i,] = C`, implicitly copies `B`, performs the partial update and assigns the new intermediate to `B`, while `A` remains unmodified. This approach is consistent with R's copy-on-write semantics, whereas other languages like Julia use update-in-place by default and require users to perform explicit `A=copy(B)` operations if implicit updates to multiple objects are unintended. Internally, the DAPHNE compiler and runtime then help to avoid unnecessary copies (e.g., via update-in-place flags and/or reference counting). Finally, DaphneDSL provides limited type polymorphism in terms of non-polymorphic data types but polymorphic value types as shown in below example:

```
X = matrix(0, 2, 2, SI32)
if( sum(Y) > 0 )
  X = sum(Y); // X can't assign scalar
else if( sum(Y) < 0 )
  X = matrix(7.3, 10, 10, FP32) // ok
```

Here, X is initialized as a 2-by-2 integer matrix, and thus cannot be assigned a scalar data type (which would give the user an error during compilation), but a matrix data type of different value types (float here) and shapes. This approach ensures that, despite conditional control flow, the compiler can infer data types at all times, while the user gets enough flexibility in terms of value types that are not always obvious to infer. Kernels for different value types can also more easily be handled at compiler and runtime level via dispatch mechanisms while, for example, distributed operations for scalars are not meaningful.

Higher-level Built-in Operations: In addition to the basic built-in operations, we further aim to provide higher-level built-in operations. This includes DSL-based functions and second-order functions. First, DSL-based functions are composite functions written in DaphneDSL that are registered in packages that can be imported in other DaphneDSL scripts. Good examples are packages for ML algorithms and individual DNN layers that can be directly called by applications. Second, we also aim to support various second-order functions that take functions as arguments. In detail, these functions include built-in functions for (a) executing SQL queries over registered tables, (b) primitives like parameter servers for data-parallel mini-batch training, and (c) user-defined functions with different data bindings.

```
// (a) SQL query processing
registerView("XTab", X);                    // X:= [a SI32, b SI8, c FP64]
Y = sql("SELECT DISTINCT a, b FROM XTab"); // Y:= [a SI32, b SI8]

// (b) primitives for mini-batch training
Mp = paramserv(model=M, features=X, labels=y,
               upd=updateGrad, agg=updateModel, utype=ASP,
               freq=BATCH, epochs=200, batchsize=128, ...);

// (c) user-defined functions (axis: 0 cell, 1 row, 2 column)
Y = map(X, foo);       // DSL function foo applied to every cell of X
Y = map(X, "v -> v.length + 1"); // C++ snippets with pre-defined env
Y = map(X, "v -> atoi(v) + 1", axis=1, lang="C++");
```

These primitives are either compiled to dedicated frame and matrix operations, or are mapped to dedicated infrastructure that repeatedly calls the passed function arguments. For example,

the parameter server (or similar distributions strategies) allows to establish temporary workers, repeatedly runs gradient and model updates and, after termination, returns the model and thus, acts as a stateless function like any other basic operation.

3.2 DaphneLib: A Python API

Overview: Python is currently undoubtedly the primary entry point to ML systems, but increasingly often also to data management and query processing as well as numerical computations. Accordingly, we aim to provide DaphneLib as a simple user-facing Python API that allows calling individual basic and higher-level DAPHNE built-in functions. The overall design follows similar abstractions like PySpark [ZC+12] and Dask [R15] by using lazy evaluation, but when evaluation is triggered assembles and executes a DaphneDSL script that reuses the entire DAPHNE compilation chain with all related optimization passes.

Lazy Evaluation: The entry point for DaphneLib is a DaphneContext that can create DAPHNE matrices or frames from passed pandas data-frames or NumPy arrays. These matrices and frames are essentially meta data objects with a reference to the leaf data. Subsequent operations can be directly invoked on these meta data objects. An example of calling our connected components built-in function will look as follows:

```
dc = DaphneContext()
G = dc.from_numpy(npG)
G = (G != 0)
c = components(G, 100, True).compute()
```

While Spark differentiates transformations and actions (where actions trigger computation), Dask [R15] provides an explicit `compute()` function. In order to make our API easily understandable, we follow this design of explicit triggers. In the example above, we create a DAPHNE matrix from a NumPy array, convert it to a Boolean matrix for robustness via `(G!=0)` in case the original graph contains edge weights or similar values (e.g., number of co-authored papers), and call the `components()` built-in function (which expects a Boolean matrix). All these operations only build a local dependency graph of operations where each metadata object points to its operation node and subsequent operations take the dependencies of inputs and place their operation on top of the dependency graph. On `compute()`, we then traverse this dependency graph in a depth-first manner (processing children of a node, and then the node itself) in order to construct a DaphneDSL script. Each node adds a line of DSL script, and the depth-first traversal with memorization (do not descent if a node was already processed) ensures an ordering by data dependencies without unnecessary redundancy if a node is reachable over multiple paths in the dependency graph. The resulting DaphneDSL script is parsed, compiled, and executed through `daphnec`, and the results are converted to NumPy arrays or pandas data-frames. In addition to exposing all basic and higher-level built-in functions and operations, we further provide an operation that directly takes DaphneDSL as a string. Given this seamless integration, users can then mix and match DAPHNE computations with other Python libraries and plotting functionality.

3.3 Parsers for External Languages (SQL, DML)

While conceptually very clear, building the DSL-based built-in functions for a wide variety of data analytics and ML algorithms from scratch could take years. Accordingly, we aim to leverage existing libraries of pre-processing primitives, ML algorithms, and DNN layers and optimizers with similar language abstractions (for subsets of functionality), and automatically convert them to DaphneDSL. For this reason, we aim to integrate parsers for external languages in the form of tools used in an offline and/or online (during script compilation) manner. As shown in Figure 2, using DaphneDSL as the common target ensures consistency of compilation utilizing both conditional control flow as well as matrix and frame operations.

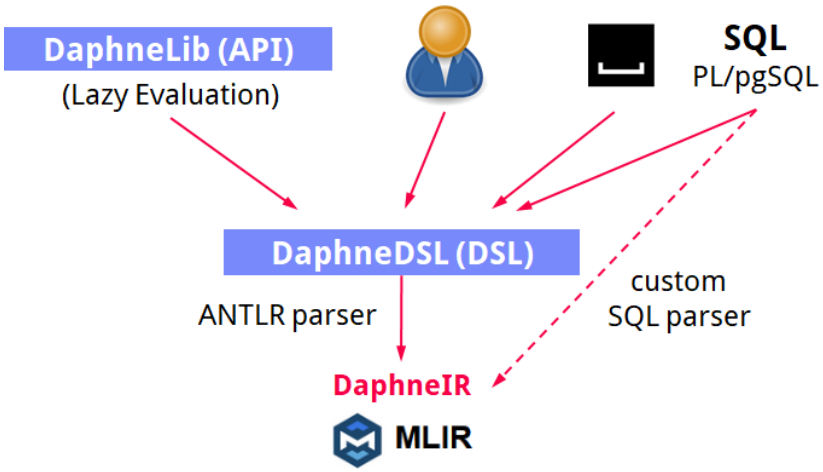

Figure 2: DAPHNE APIs and DSLs.

SQL Parser: Initially we contemplated reusing and extending the PostgreSQL parser, but later decided to start from scratch and incrementally build up a tailor-made SQL parser that currently converts directly to frame operations in DaphneIR. In the future, we will add dedicated components for PostgreSQL dialect and its procedural extensions PL/pgSQL.

SystemDS DML: Apache SystemDS [BA+20], as the successor of Apache SystemML, exposes a DSL called DML (declarative machine learning language) with an R-like syntax. Most importantly it contains a linear-algebra-based hierarchy of primitives for data cleaning and data augmentation, hyper-parameter tuning, feature selection, a variety of ML algorithms, DNN layers and optimizers, many graph algorithms, as well as model debugging, evaluation and scoring. We aim to create an offline tool to parse and convert these DML scripts into DaphneDSL, allowing to materialize and adapt these primitives for DaphneDSL language abstractions, while capitalizing on the wealth of functionality, the liberal Apache-v2 license, and SystemDS contributors in the DAPHNE consortium. Finally, we will also explore additional converters from TensorFlow [A+16], PyTorch [P+19], and Scikit-learn [P+12] pipelines in order to quickly bridge the gap between applications and the DAPHNE system infrastructure.

3.4 Integration of External Libraries

The integration of external libraries is very important in order to allow an incremental adoption of the DAPHNE infrastructure by potential users. This integration focuses on two main aspects, low overhead data exchange with UDFs and libraries, as well as exposing the tuning knobs (e.g., degree of parallelism) of libraries to optimization and hierarchical scheduling of IDA pipelines.

Data Transfer: The basic integration of external libraries is through materialized intermediates and the sketched abstractions for user-defined functions. Additionally, we aim to support zero-copy data formats like Apache Arrow (https://arrow.apache.org/), chunked data transfers as known from R-integrations [DSB+10,GLW+11], and careful tuning of buffer management as done for data exchange of database systems with user-defined functions [RMT+17].

Scheduling: Additionally, we aim to annotate the called UDFs in special script-level scopes to expose tunable parameters such as the degree of parallelism, and other MPI / OpenBLAS configurations of HPC libraries. Exposing and utilizing these parameters during optimization will allow for seamless scheduling and better utilization of hardware resources in complex, composite IDA pipelines that otherwise would need to make certain assumptions and treat such external libraries as black-box functions, blocking any other concurrent operations. Besides manually exposing such tuning knobs, an interesting research direction is the development of custom OpenMP backends that intercept tasks of the external libraries and forwards these tasks to the DAPHNE scheduler similar to the integration of custom application codes in analytical database systems [WPM+15].

4 Overview Compiler Design

Figure 1 also shows the overall compiler design. The initial compiler prototype will be shared as part of the future demonstrator deliverable D3.2, but in this section, we already describe the compiler design for a holistic discussion of language abstractions, compilation chain, and means of extensibility of both language abstractions as well as compiler and runtime.

4.1 DaphneIR Dialect (Intermediate Representation)

MLIR Background: The ANTLR parser converts given DaphneDSL scripts into DaphneIR, an MLIR dialect. MLIR [LP+20] is a customizable compiler infrastructure for reuse and low-cost domain-specific compilers. Programs are represented in static single assignment (SSA) form – which allows only a single assignment to an otherwise immutable variable – and then lowered to LLVM. Its basic concepts are modules, functions, and regions that can contain sequences of blocks, which in turn contain sequences of operations. The basic philosophy is that everything, even loop structures with arbitrarily complex body programs, are operations (that may or may not contain regions) and everything is designed for customization. Using MLIR allows for reuse of basic infrastructure and various optimization passes as a library (along with existing documentation), and allows future extensibility by other MLIR dialects. For control structures like branches and loops, we already use the SCF (structured control flow) dialect.

DaphneIR: The DAPHNE MLIR dialect, called DaphneIR, defines the types, operations, and various traits (e.g., for schema, type, and shape inference) in so-called TableGen [TD21] records, from which C++ code is automatically generated. Operations produce values of a certain type. For example, the following snippets show the TableGen specifications of basic scalar value types, the matrix multiplication operation, and the shape inference interface:

```
def SIntScalar : AnyTypeOf<[SI8, SI32, SI64], "signed integer">;
def UIntScalar : AnyTypeOf<[UI8, UI32, UI64], "unsigned integer">;
def IntScalar : AnyTypeOf<[SIntScalar, UIntScalar], "integer">;
def FloatScalar : AnyTypeOf<[F32, F64], "float">;
def NumScalar : AnyTypeOf<[IntScalar, FloatScalar], "numeric">;

def Daphne_MatMulOp : Daphne_Op<"matMul", [
  DeclareOpInterfaceMethods<VectorizableOpInterface>,
  NumRowsFromIthArg<0>, NumColsFromIthArg<1>
]> {
  let arguments = (ins MatrixOf<[NumScalar]>:$lhs, MatrixOf<[NumScalar]>:$rhs);
  let results = (outs MatrixOf<[NumScalar]>:$res);
}

def InferShapeOpInterface : OpInterface<"InferShape"> {
  let description = [{
    Interface to infer the shape(s) of the data object(s)
    returned by an operation.
  }];

  let methods = [
    InterfaceMethod<
      "Infer the shape(s) of the output data object(s).",
      "std::vector<std::pair<ssize_t, ssize_t>>", "inferShape", (ins)>
  ];
}
```

Here `NumScalar` refers to any supported integer or floating point type, and is used to parameterize the value type of the inputs and outputs of the matrix multiplication. The `InferShapeOpInterface` specifies an `inferShape` method that returns a vector of matrix dimensions (pair of rows and columns, for each result of the operation), which for the matrix multiplication takes the number of rows from the left-hand-side and number of columns from the right-hand-side. Additional traits exist for example, for operator fusion (vectorization), distributed operations, and type inference. During parsing, we instantiate the individual operations, blocks, and regions.

Example IR Program: Returning to our connected components example, the DSL script (currently in slightly modified form and with random data generation) is parsed into the following DaphneIR program (in its textual representation):

```
module  {
  func @main() {
    %0 = "daphne.constant"() {value = 2000 : si64} : () -> si64
    %1 = "daphne.constant"() {value = 1 : si64} : () -> si64
    %2 = "daphne.constant"() {value = 1 : si64} : () -> si64
    %3 = "daphne.constant"() {value = 1.000000e-03 : f64} : () -> f64
    %4 = "daphne.constant"() {value = -1 : si64} : () -> si64
    %5 = "daphne.cast"(%0) : (si64) -> index
    %6 = "daphne.cast"(%0) : (si64) -> index
    %7 = "daphne.randMatrix"(%5, %6, %1, %2, %3, %4) :
         (index, index, si64, si64, f64, si64) -> !daphne.Matrix<?x?xsi64>
    %8 = "daphne.constant"() {value = 1000 : si64} : () -> si64
    %9 = "daphne.constant"() {value = 1 : si64} : () -> si64
```

```
    %10 = "daphne.constant"() {value = 1 : si64} : () -> si64
    %11 = "daphne.seq"(%9, %0, %10) : (si64, si64, si64) -> !daphne.Matrix<?x?xsi64>
    %12 = "daphne.constant"() {value = 0x7FF0000000000000 : f64} : () -> f64
    %13 = "daphne.constant"() {value = 1 : si64} : () -> si64
    %14:3 = scf.while (%arg0 = %13, %arg1 = %12, %arg2 = %11) :
            (si64, f64, !daphne.Matrix<?x?xsi64>) -> (si64, f64, !daphne.Matrix<?x?xsi64>)
    {
      %15 = "daphne.constant"() {value = 0.000000e+00 : f64} : () -> f64
      %16 = "daphne.ewGt"(%arg1, %15) : (f64, f64) -> f64
      %17 = "daphne.cast"(%16) : (f64) -> si64
      %18 = "daphne.ewLe"(%arg0, %8) : (si64, si64) -> si64
      %19 = "daphne.ewAnd"(%17, %18) : (si64, si64) -> si64
      %20 = "daphne.cast"(%19) : (si64) -> i1
      scf.condition(%20) %arg0, %arg1, %arg2 : si64, f64, !daphne.Matrix<?x?xsi64>
    } do {
    ^bb0(%arg0: si64, %arg1: f64, %arg2: !daphne.Matrix<?x?xsi64>):  // no predecessors
      %15 = "daphne.transpose"(%arg2) :
            (!daphne.Matrix<?x?xsi64>) -> !daphne.Matrix<?x?xsi64>
      %16 = "daphne.ewMul"(%7, %15) :
           (!daphne.Matrix<?x?xsi64>, !daphne.Matrix<?x?xsi64>) -> !daphne.Matrix<?x?xsi64>
      %17 = "daphne.constant"() {value = 0 : si64} : () -> si64
      %18 = "daphne.maxRow"(%16) : (!daphne.Matrix<?x?xsi64>) -> !daphne.Matrix<?x?xsi64>
      %19 = "daphne.ewMax"(%18, %arg2) :
           (!daphne.Matrix<?x?xsi64>, !daphne.Matrix<?x?xsi64>) -> !daphne.Matrix<?x?xsi64>
      %20 = "daphne.ewNeq"(%19, %arg2) :
           (!daphne.Matrix<?x?xsi64>, !daphne.Matrix<?x?xsi64>) -> !daphne.Matrix<?x?xsi64>
      %21 = "daphne.sumAll"(%20) : (!daphne.Matrix<?x?xsi64>) -> si64
      %22 = "daphne.cast"(%21) : (si64) -> f64
      %23 = "daphne.constant"() {value = 1 : si64} : () -> si64
      %24 = "daphne.ewAdd"(%arg0, %23) : (si64, si64) -> si64
      scf.yield %24, %22, %19 : si64, f64, !daphne.Matrix<?x?xsi64>
    }
    "daphne.return"() : () -> ()
  }
}
```

After additional optimization passes for rewrites, shape inference, and lowering of the SCF dialect we obtain the following representation:

```
module  {
  func @main() {
    %0 = "daphne.call_kernel"() {callee = "_createDaphneContext__DaphneContext"} : () -> !daphne.DaphneContext
    %1 = "daphne.constant"() {value = 2000 : si64} : () -> si64
    %2 = "daphne.constant"() {value = 1 : si64} : () -> si64
    %3 = "daphne.constant"() {value = 1.000000e-03 : f64} : () -> f64
    %4 = "daphne.constant"() {value = -1 : si64} : () -> si64
    %5 = "daphne.call_kernel"(%1, %0) {callee = "_cast__size_t__int64_t"} : (si64, !daphne.DaphneContext) -> index
    %6 = "daphne.call_kernel"(%1, %0) {callee = "_cast__size_t__int64_t"} : (si64, !daphne.DaphneContext) -> index
    %7 = "daphne.call_kernel"(%5, %6, %2, %2, %3, %4, %0)
          {callee = "_randMatrix__DenseMatrix_int64_t__size_t__size_t__int64_t__int64_t__double__int64_t"} :
          (index, index, si64, si64, f64, si64, !daphne.DaphneContext) -> !daphne.Matrix<2000x2000xsi64>
    %8 = "daphne.constant"() {value = 1000 : si64} : () -> si64
    %9 = "daphne.call_kernel"(%2, %1, %2, %0) {callee = "_seq__DenseMatrix_int64_t__int64_t__int64_t__int64_t"} :
         (si64, si64, si64, !daphne.DaphneContext) -> !daphne.Matrix<2000x1xsi64>
    %10 = "daphne.constant"() {value = 0x7FF0000000000000 : f64} : () -> f64
    br ^bb1(%2, %10, %9 : si64, f64, !daphne.Matrix<2000x1xsi64>)
  ^bb1(%11: si64, %12: f64, %13: !daphne.Matrix<2000x1xsi64>):  // 2 preds: ^bb0, ^bb2
    %14 = "daphne.constant"() {value = 0.000000e+00 : f64} : () -> f64
    %15 = "daphne.call_kernel"(%12, %14, %0) {callee = "_ewGt__double__double__double"} :
          (f64, f64, !daphne.DaphneContext) -> f64
    %16 = "daphne.call_kernel"(%15, %0) {callee = "_cast__int64_t__double"} : (f64, !daphne.DaphneContext) -> si64
    %17 = "daphne.call_kernel"(%11, %8, %0) {callee = "_ewLe__int64_t__int64_t__int64_t"} :
          (si64, si64, !daphne.DaphneContext) -> si64
    %18 = "daphne.call_kernel"(%16, %17, %0) {callee = "_ewAnd__int64_t__int64_t__int64_t"} :
          (si64, si64, !daphne.DaphneContext) -> si64
    %19 = "daphne.call_kernel"(%18, %0) {callee = "_cast__bool__int64_t"} : (si64, !daphne.DaphneContext) -> i1
    cond_br %19, ^bb2(%11, %12, %13 : si64, f64, !daphne.Matrix<2000x1xsi64>), ^bb3
  ^bb2(%20: si64, %21: f64, %22: !daphne.Matrix<2000x1xsi64>):  // pred: ^bb1
    %23 = "daphne.call_kernel"(%22, %0) {callee = "_transpose__DenseMatrix_int64_t__DenseMatrix_int64_t"} :
          (!daphne.Matrix<2000x1xsi64>, !daphne.DaphneContext) -> !daphne.Matrix<1x2000xsi64>
    %24 = "daphne.call_kernel"(%7, %23, %0)
          {callee = "_ewMul__DenseMatrix_int64_t__DenseMatrix_int64_t__DenseMatrix_int64_t"} :
          (!daphne.Matrix<2000x2000xsi64>, !daphne.Matrix<1x2000xsi64>, !daphne.DaphneContext) ->
          !daphne.Matrix<2000x2000xsi64>
    %25 = "daphne.call_kernel"(%24, %0) {callee = "_maxRow__DenseMatrix_int64_t__DenseMatrix_int64_t"} :
```

```
        (!daphne.Matrix<2000x2000xsi64>, !daphne.DaphneContext) -> !daphne.Matrix<2000x1xsi64>
    %26 = "daphne.call_kernel"(%25, %22, %0) {callee =
        "_ewMax__DenseMatrix_int64_t__DenseMatrix_int64_t__DenseMatrix_int64_t"} : (!daphne.Matrix<2000x1xsi64>,
        !daphne.Matrix<2000x1xsi64>, !daphne.DaphneContext) -> !daphne.Matrix<2000x1xsi64>
    %27 = "daphne.call_kernel"(%26, %22, %0) {callee =
        "_ewNeq__DenseMatrix_int64_t__DenseMatrix_int64_t__DenseMatrix_int64_t"} : (!daphne.Matrix<2000x1xsi64>,
        !daphne.Matrix<2000x1xsi64>, !daphne.DaphneContext) -> !daphne.Matrix<2000x1xsi64>
    %28 = "daphne.call_kernel"(%27, %0) {callee = "_sumAll__int64_t__DenseMatrix_int64_t"} :
        (!daphne.Matrix<2000x1xsi64>, !daphne.DaphneContext) -> si64
    %29 = "daphne.call_kernel"(%28, %0) {callee = "_cast__double__int64_t"} : (si64, !daphne.DaphneContext) -> f64
    %30 = "daphne.call_kernel"(%20, %2, %0) {callee = "_ewAdd__int64_t__int64_t__int64_t"} :
        (si64, si64, !daphne.DaphneContext) -> si64
    br ^bb1(%30, %29, %26 : si64, f64, !daphne.Matrix<2000x1xsi64>)
  ^bb3:  // pred: ^bb1
    "daphne.call_kernel"(%0) {callee = "_destroyDaphneContext"} : (!daphne.DaphneContext) -> ()
    "daphne.return"() : () -> ()
  }
}
```

Finally, this representation is further lowered to the MLIR-LLVM dialect, including LLVM function calls to specific kernels such as `ewNeq` and `sumAll` for `sum(u != c)`, and ultimately compiled to hardware-specific instructions.

```
%77 = llvm.alloca %76 x !llvm.ptr<i1> : (i64) -> !llvm.ptr<ptr<i1>>
%78 = llvm.mlir.null : !llvm.ptr<i1>
llvm.store %78, %77 : !llvm.ptr<ptr<i1>>
%79 = llvm.call @_ewNeq__DenseMatrix_int64_t__.__DenseMatrix_int64_t(%77, %75, %55, %4) :
      (!llvm.ptr<ptr<i1>>, !llvm.ptr<i1>, !llvm.ptr<i1>, !llvm.ptr<i1>) -> !llvm.void
%80 = llvm.load %77 : !llvm.ptr<ptr<i1>>
%81 = llvm.mlir.constant(1 : i64) : i64
%82 = llvm.alloca %81 x i64 : (i64) -> !llvm.ptr<i64>
%83 = llvm.call @_sumAll__int64_t__DenseMatrix_int64_t(%82, %80, %4) :
      (!llvm.ptr<i64>, !llvm.ptr<i1>, !llvm.ptr<i1>) -> !llvm.void
%84 = llvm.load %82 : !llvm.ptr<i64>
```

4.2 Compilation Chain

Optimization Passes: The DAPHNE compilation is based on MLIR optimization passes for enrichment by inferred properties and lowering to executable runtime programs. This lowering descends from high-level, abstract operations and data types to multiple levels of operator specialization (e.g., local/distributed operations, device placement on CPUs/GPUs/FPGAs, choice of physical kernels for specific environments and devices), as well as data specialization (e.g., DenseMatrix/CSRMatrix representations). Optimization passes for rewrites and lowering can be interleaved and repeatedly executed. Important categories of optimization passes include (so far, only partially implemented):

- *MLIR Programming Language Rewrites* (common subexpression elimination, constant propagation, constant folding, branch removal, code motion/loop hoisting, function inlining / unrolling)
- *Type and Property Inference* (e.g., data and value types, shape/dimensions, schema, sparsity/cardinality, symmetry)
- *Inter-Procedural Analysis* (analysis of function call graphs, propagation of types, dimensions, properties)
- *Algebraic Simplification Rewrites* (e.g., many peephole optimizations for sequences/sub-DAGs of relational/linear algebra operations)
- *Operator Ordering* (e.g., join ordering/enumeration, matrix multiplication chain optimization, sum-product optimizations, data-flow-graph linearization)

- *Generation of Fused Operator Pipelines* (selection of fused operators in DAGs, vectorization/tiling, and splitting/merging strategies of inputs/results)
- *Memory Management* (update-in-place, reuse of allocations, garbage collection)
- *Execution Type Selection* (local vs distributed operations, w/ distribution primitives for distributed caching/partitioning)
- *Device Placement* (e.g., CPU/GPU/FPGA, multiple devices)
- *Physical Operator Selection* (e.g., different join/group-by operators, matrix multiplication operators, matrix-vector element-wise operations)

Additional Compiler Components: These aforementioned optimization passes are assembled in a heuristic manner according to known dependencies among rewrites and lowering passes. In addition, there are several compiler components that are orthogonal to the compilation chain and used by several passes, and even during runtime. First, many advanced rewrites and reordering optimization passes require cost estimation for subprograms. We aim to provide a central cost estimation component including cardinality and sparsity estimation (with different cost functions and summary statistics) for blocks of operations and entire sub-programs. The latter has to additionally reason about loops and branches and aggregate a hierarchy of regions and blocks. Second, data-dependent operators, user-defined functions, and conditional control flow can create unknown shapes and properties, which would result in robust but inefficient fallback plans. A dynamic recompilation component aims to adaptively recompile remaining sub-programs at natural or artificial block boundaries according to the actual sizes of intermediates. Third and finally, there is a wide variety of additional runtime strategies that would benefit from compiler assistance – examples include compiler-assisted compression and reuse, which both leverage workload characteristics (aggregated operation workload) to make more informed choices during runtime.

5 Extensibility

The overall design principle of an open and extensible system architecture, is of utmost importance to enable low effort exploratory experimentation and custom extensions for new data types, operations, and hardware. We aim to support extensibility in terms of configurations of internal behavior, but also in terms of extensions for custom operations (at script level and kernels), custom data formats (e.g., vendor-specific binary formats), new data types (e.g., compressed types), and various extensions of the compiler and runtime system (e.g., additional IR dialects, new optimization passes, and scheduling algorithms).

5.1 API and DSL Extensibility

At API, DSL, and configuration level, there are multiple aspects of extensibility, all of which require a discussion of different personas and deployments. DAPHNE aims for deployments with different distribution strategies (local, embedded, distributed collections, federated data, parameter servers, custom libraries, and ring/tree reduce), different distributed computing frameworks (DAPHNE standalone, embedded in DBMS and orchestration engines, Spark, Ray, and MPI), different resource managers (e.g., dedicated clusters, resource negotiators like YARN,

Mesos, and Kubernetes, and HPC Batch schedulers like Slurm), as well as different on-premise and cloud hardware resources. In this context, we see the personas of (1) internal/external developers, (2) users, and (3) infrastructure administrators.

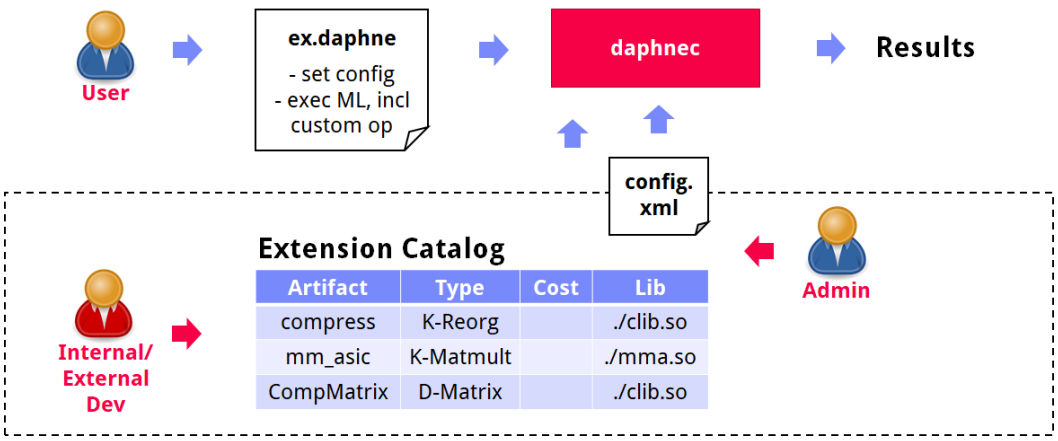

Figure 3: DAPHNE Configuration and Extensibility.

Extension Catalog: Figure 3 shows an example workflow involving an initial design of an extension catalog that allows registering dedicated artifacts in the form of shared libraries. The catalog also registers the type of extension (e.g., kernels for existing operations like matrix multiplication if possible, or data types), traits and properties, as well as cost functions provided by developers. Based on this metadata, theses extensions are represented in DaphneIR and thus included in various optimization passes such as shape inference, and operator selection. The concrete use of extensions can be further influenced both at script level (mostly by users or during experimentation) or configuration files that influence the entire deployment and thus, potentially many users (mostly by administrators but also in batch scheduling scripts).

Extensibility at Script Level: At DaphneDSL level, we provide means for obtaining and setting configurations parameters and topology information transiently in a programmatic manner (e.g., `getNumThreads()` and `setNumThreads(32)`). These configurations include available devices, degree of parallelism, memory budgets, but also selecting different garbage collection and scheduling algorithms. Depending on the place of invocation these transient configurations affect the default configuration for the current scope and child-scopes (e.g., called function). Script-level extensibility also includes dedicated built-in functions for affecting data representations, data placement, and operator placement:

```
X = sparse(Y);
X = compress(Y);
X = device(Y, "/GPU:0");
X = device(Y, ["/GPU:0","/GPU:1"], round_robin);
X = Y @_gpu Z; // matmult on GPU
```

All these decisions are made via basic built-in functions. This approach provides clear data-flow semantics with copy-on-write, and allows extensibility. For example, a developer might create a new compressed data type and operations. By registering and invoking a shared library

with a custom `compressXYZ()` operation, the data can be explicitly brought into this representation and subsequent function calls on this data object will then be dispatched to the new specialized operator implementations accordingly. All these script-level decisions lose data dependence but are user choices and will be treated as constraints. The optimizing compiler then handles remaining operations around these fixed operators, and helps lowering everything to execution plans as needed (multi-level specification).

Extensibility at Configuration Level: Similar to setting configuration parameters at script level, the same configurations can also be set globally at deployment level. We will provide JSON or XML configuration files in the installation directory, and allow users to use per-user configuration files (e.g., configured via bashrc or environment variables). These configuration files or environment variables will also enable automatic modifications in the context of batch schedulers (e.g., after resource allocation). Regarding extensibility, these configurations primarily focus on selecting alternative algorithms and influencing the extension catalog.

5.2 Compiler and Runtime Extensibility

Besides the extensibility features at script and configuration level (which also allow the introduction of new runtime kernels), we further aim to allow configurations and extensions of the optimizing DAPHNE compiler and DaphneIR.

Daphne IR: The DAPHNE MLIR dialect DaphneIR defines types, operations, and various traits (e.g., for schema, type, and shape inference) in so-called TableGen [TD21] records, but also reuses existing MLIR dialects like SCF (structured control flow). One developer-centric direction for extensibility is the extension of our DaphneIR dialect. Common use cases are adding new operations of an existing category (e.g., a new unary elementwise operation), adding a new category of operations (e.g., a specific quaternary operator), adding new traits (e.g., as help for new optimization passes). Additionally, developers might add existing or new MLIR dialects and integrate them with the rest of the system by changing the DAPHNE infrastructure internally. While some of these extensions can reuse most of the existing runtime operations, other require additional runtime kernels.

Compilation Chain: MLIR's approach of applying a sequence of optimization passes is already very modular and can reuse existing LLVM and MLIR passes (e.g., constant folding, common subexpression elimination, code motion/loop hoisting). Adding new optimization passes or re-composing existing optimization passes into custom compilation chains is a natural direction for compiler extensibility. For example, having registered a new data type or kernel, an additional optimization pass may apply them for a given IR program under certain conditions.

Sideways Entry in Multi-level Compilation: The normal invocation of DAPHNE through `daphnec` (by a user or through the Python API) takes a DaphneDSL script and then compiles and executes this script. In order to allow for debugging and understanding, an explain flag allows to print the DaphneIR at different states of compilation. Similar to the use of kernels at script level, which are treated as constraints, we will extend `daphnec` to take valid DaphneIR instead of DaphneDSL as program specification as well. This flexibility allows researchers to obtain the generated execution plan, modify the plan slightly (e.g., to force certain sequences

of local or distributed operations), and execute this plan through daphnec, which performs the remaining lowering and runs the final executable plan.

6 Related Work

Besides existing work on language abstractions in the areas of data management, high-performance computing, and ML systems, there are a couple of closely related projects and systems, aiming for system infrastructure similar to DAPHNE. In the following, we discuss this additional related work of systems and language specifications for IDA pipelines (partial extension from deliverable D2.1 [D2.1]), means of extensibility in data management and ML systems including algorithms, data formats, and the underlying hardware.

Systems for IDA Pipelines: The trend toward IDA pipelines is currently handled with a combination of existing systems including standalone and embedded DBMS like DuckDB [RM20], ML systems like TensorFlow [A+16] or PyTorch [P+19], data-parallel computation frameworks like Spark [ZC+12], Flink [CK+15], or Dask [R15] (often with collections of tiles of an overall matrix or frame), as well as a variety of specialized systems or libraries (e.g., for graph processing and time series analysis) and even custom application codes. Furthermore, ML systems are extended with features for basic data processing (e.g., from TensorFlow to TFX [BB+17]), DBMS are extended with ML capabilities (e.g., via UDFs or lambda functions) [KBY17], data-parallel frameworks aim to provide a unified environment [ZC+12], compilation frameworks like MLIR [LP+20] or CVM [MM+20] provide common compiler infrastructure and HPC techniques are increasingly adopted across these systems [BKS20]. However, these integrated systems often rely on separate libraries and data representations for query processing, ML, and HPC, which makes them difficult to apply in tightly integrated pipelines where, for example, ML-based data cleaning, query processing, and ML training is performed repeatedly in an interleaved manner.

Extensibility: Providing extensibility for functionality and performance has been investigated in different systems and at different abstraction levels. First, in the context of database management systems, there are great surveys of work on extensibility [CH90]. Abstract data types and user-defined functions/aggregates were introduced in PostgreSQL [S16] and are now widely used in practice. Such UDFs have also been used to integrate ML into DBMS [FK+12] and HPC OpenMP applications in DBMS [WPM+15], but UDFs are often treated as black boxes thus, not subject to optimization (unless parsed and explicitly included into the optimization process [HP+12]). Additional means of extensibility include query optimizer generators [GM+93], extensible cardinality estimation [JS+02], interfaces for new storage methods (aka storage managers, or storage engines) with well-defined interfaces for create/drop relation, insert, delete, update, and get()/getNext() operations, but also persistently stored modules like attachments or triggers. Second, several ML systems also provide means of extensibility. DNN frameworks like Caffee, PyTorch, and TensorFlow make it easy to add layers and optimizers. AutoML systems like MLBase defined catalogs for registering new ML algorithms with their cost functions [KT+13]. Additionally, some ML systems also focus on extensibility of ML system internals. Examples include TensorFlow distribution strategies [G19] for data exchange in mini-batch training, TVM code generation for new hardware backends

[CM+18], and the Flashlight library for extensibility by custom modules and kernels [F21]. DAPHNE takes inspiration from these existing works and aims to build an open and extensible infrastructure for IDA pipelines that combine data management, HPC, and ML systems.

7 Conclusions

To summarize, we described the overall DAPHNE language design specification, including DaphneDSL and DaphneLib, an overview of the DAPHNE compiler including DaphneIR, and plans for extensibility at different levels. In conclusion, a common language abstraction based on abstract frames and matrix operations (as well as a hierarchy of composite language abstractions) for specifying and executing integrated data analysis pipelines seems feasible. In this context, we follow the major design principles of coarse-grained frame and matrix operations for preserving semantics and simplifying parallelization, data independence to enable adaptation to data and deployment characteristics, and extensibility to allow for exploration and extensions for new environments. The language abstractions will be extended as we build out the DAPHNE prototype over the next years. Interesting research directions related to language abstractions include the design of the catalog for registering extensions, generic optimization passes that seamlessly leverage unknown extensions, constrained program optimization respecting user-provided decisions, and internal runtime abstractions for low-overhead access of custom data types to reuse non-specialized operators.

8 References

[A+16] Martín Abadi et al.: TensorFlow: A System for Large-Scale Machine Learning. OSDI 2016

[AX+15] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, Matei Zaharia: Spark SQL: Relational Data Processing in Spark. SIGMOD 2015

[BA+20] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqui, Sebastian Benjamin Wrede: SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. CIDR 2020

[BB+17] Denis Baylor et al: TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. KDD 2017

[BBY13] Botong Huang, Shivnath Babu, Jun Yang: Cumulon: optimizing statistical data analysis in the cloud. SIGMOD 2013

[BC+18] Jeff Bezanson, Jiahao Chen, Benjamin Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, Lionel Zoubritzky: Julia: dynamism and performance reconciled by design. Proc. ACM Program. Lang. 2(OOPSLA), 2018

[BD+16] Matthias Boehm, Michael Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick Reiss, Prithviraj Sen, Arvind Surve, Shirish Tatikonda: SystemML: Declarative Machine Learning on Spark. PVLDB 9(13), 2016

[BK+12] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman: Julia: A Fast Dynamic Language for Technical Computing. CoRR abs/1209.5145 (2012)

[BKS20] S. Blanas, P. Koutris, and A. Sidiropoulos. Topology-aware Parallel Data Processing: Models, Algorithms and Systems at Scale. In CIDR, 2020.

[BMM13] Carsten Binnig, Norman May, Tobias Mindnich: SQLScript: Efficiently Analyzing Big Enterprise Data in SAP HANA. BTW 2013

[BT+14] Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen, Yuanyuan Tian, Douglas Burdick, Shivakumar Vaithyanathan: Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. PVLDB 7(7), 2014

[C70] E. F. Codd: A Relational Model of Data for Large Shared Data Banks. Commun. ACM 13(6), 1970

[CB74] Donald D. Chamberlin, Raymond F. Boyce: SEQUEL: A Structured English Query Language. SIGMOD Workshop, Vol. 1 1974

[CG+05] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, Vivek Sarkar: X10: an object-oriented approach to non-uniform cluster computing. OOPSLA 2005

[CH90] Michael J. Carey, Laura M. Haas: Extensible Database Management Systems. SIGMOD Rec. 19(4), 1990

[CK+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, Kostas Tzoumas: Apache Flink™: Stream and Batch Processing in a Single Engine. IEEE Data Eng. Bull. 38(4), 2015

[CM+18] Tianqi Chen et al.: TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. OSDI 2018

[D2.1] DAPHNE: D2.1 Initial System Architecture, EU Project Deliverable, 08/2021

[DR18] Luna Dong, Theodoros Rekatsinas: Data Integration and Machine Learning: A Natural Synergy. PVLDB 11(12), 2018

[DSB+10] Sudipto Das, Yannis Sismanis, Kevin S. Beyer, Rainer Gemulla, Peter J. Haas, John McPherson: Ricardo: integrating R and Hadoop. SIGMOD 2010

[F21] Facebook: Flashlight: Fast and flexible machine learning in C++, 2021, https://github.com/flashlight/flashlight

[FK+12] Xixuan Feng, Arun Kumar, Benjamin Recht, Christopher Ré: Towards a unified architecture for in-RDBMS analytics. SIGMOD 2012

[G19] Google: Inside TensorFlow: tf.distribute.Strategy, 2019, https://www.youtube.com/watch?v=jKV53r9-H14

[GM+93] Goetz Graefe, William J. McKenna: The Volcano Optimizer Generator: Extensibility and Efficient Search. ICDE 1993

[GLW+11] Philipp Große, Wolfgang Lehner, Thomas Weichert, Franz Färber, Wen-Syan Li: Bridging Two Worlds with RICE Integrating R into the SAP In-Memory Computing Engine. PVLDB 4(12), 2011

[HM+20] Charles R. Harris et al: Array programming with NumPy. Nat. 585, 2020

[HP+12] Fabian Hueske, Mathias Peters, Matthias Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, Kostas Tzoumas: Opening the Black Boxes in Data Flow Optimization. PVLDB 5(11), 2012

[HR+12] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, Arun Kumar: The MADlib Analytics Library or MAD Skills, the SQL. PVLDB 5(12), 2012

[JS+02] Vanja Josifovski, Peter M. Schwarz, Laura M. Haas, Eileen Tien Lin: Garlic: a new flavor of federated query processing for DB2. SIGMOD 2002

[KBY17] Arun Kumar, Matthias Boehm, and Jun Yang: Data Management in Machine Learning: Challenges, Techniques, and Systems. In SIGMOD, 2017.

[KT+13] Tim Kraska, Ameet Talwalkar, John C. Duchi, Rean Griffith, Michael J. Franklin, Michael I. Jordan: MLbase: A Distributed Machine-learning System. CIDR 2013

[LP+20] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, Oleksandr Zinenko: MLIR: A Compiler Infrastructure for the End of Moore's Law. CoRR abs/2002.11054, 2020

[MBK00] Stefan Manegold, Peter A. Boncz, Martin L. Kersten: Optimizing database architecture for the new bottleneck: memory access. VLDB J. 9(3), 2000

[MDM19] Piero Molino, Yaroslav Dudin, Sai Sumanth Miryala: Ludwig: a type-based declarative deep learning toolbox. CoRR abs/1909.07930, 2019

[MG+11] Josh Milthorpe, V. Ganesh, Alistair P. Rendell, David Grove: X10 as a Parallel Language for Scientific Computation: Practice and Experience. IPDPS 2011

[MH+12] Floréal Morandat, Brandon Hill, Leo Osvald, Jan Vitek: Evaluating the Design of the R Language - Objects and Functions for Data Analysis. ECOOP 2012

[MM+20] Ingo Müller, Renato Marroquin, Dimitrios Koutsoukos, Mike Wawrzoniak, Sabir Akhadov, Gustavo Alonso: The collection Virtual Machine: an abstraction for multi-frontend multi-backend data analysis. DaMoN@SIGMOD 2020

[NS+20] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, Matteo Interlandi: A Tensor Compiler for Unified Machine Learning Prediction Serving. OSDI 2020

[P+11] Fabian Pedregosa et al.: Scikit-learn: Machine Learning in Python. JMLR 12, 2011

[P+19] Adam Paszke et al: PyTorch: An Imperative Style, High-Performance Deep Learning Library. NeurIPS 2019

[R15] Matthew Rocklin: Dask: Parallel Computation with Blocked algorithms and Task Scheduling, SCIPY 2015.

[Re20] Christopher Ré: Overton: A Data System for Monitoring and Improving Machine-Learned Products. CIDR 2020

[RM20] Mark Raasveldt, Hannes Mühleisen: Data Management for Data Science - Towards Embedded Analytics. CIDR 2020

[RMT+17] Viktor Rosenfeld, René Müller, Pinar Tözün, Fatma Özcan: Processing Java UDFs in a C++ environment. SoCC 2017

[RP+17] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César A. Galindo-Legaria, Conor Cunningham: Froid: Optimization of Imperative Programs in a Relational Database. PVLDB 11(4), 2017

[S16] Michael Stonebraker: The land sharks are on the squawk box. Commun. ACM 59(2), 2016

[SB21] Svetlana Sagadeeva, Matthias Boehm: SliceLine: Fast, Linear-Algebra-based Slice Finding for ML Model Debugging. SIGMOD 2021

[SW+76] Michael Stonebraker, Eugene Wong, Peter Kreps, Gerald Held: The Design and Implementation of INGRES. ACM Trans. Database Syst. 1(3), 1976

[SW+15] Jaeho Shin, Sen Wu, Feiran Wang, Christopher De Sa, Ce Zhang, Christopher Ré: Incremental Knowledge Base Construction Using DeepDive. PVLDB 8(11), 2015

[TD21] LLVM Team: TableGen Overview, https://llvm.org/docs/TableGen/, 2021

[WPM+15] Florian Wolf, Iraklis Psaroudakis, Norman May, Anastasia Ailamaki, Kai-Uwe Sattler: Extending database task schedulers for multi-threaded application code. SSDBM 2015

[ZC+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, Ion Stoica: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI 2012

Appendix A Prioritized List of Operations

Continuous experiments for quantifying and improving the DAPHNE prototype currently use a set of simple IDA pipelines. These pipelines include (1) query processing and linear regression, (2) ResNet-20 classification, (3) K-Means clustering, and (4) connected components graph processing. For supporting these algorithms, the list of prioritized operations is:

- *Matrix multiplications:* GEMM, SYRK, GEMV, conv2d
- *Element-wise operations:* M–r, M/r, M/c, M*r, M*s, M<=c, AXPY, r!=r, max(M,M), max(M,s) where r is a row vector, c is a column vector, and M is a matrix.
- *Aggregations:* colMeans(), colSds(), colSums(), rowSums(), sum(), rowMins(), rowMaxs()
- *Reorganizations:* slicing, indexing, transposition, cbind(), reshape()
- *Others:* solve(), seq(), selection, group-join, batchnorm