Not Black-Box Anymore! Enabling Analytics-Aware Optimizations in Teradata Vantage

Mohamed Eltabakh, Anantha Subramanian, Awny Al-Omari, Mohammed Al-Kateb, Sanjay Nair
Mahbub Hasan, Wellington Cabrera, Charles Zhang, Amit Kishore, Snigdha Prasad
Teradata Labs
CA, USA
firstname.lastname@teradata.com

ABSTRACT

Teradata Vantage is a platform for integrating a broad range of analytical functions and capabilities with the Teradata's SQL engine. One of the main challenges in optimizing the execution of these analytical functions is that many of them are not only black boxes, but also have polymorphic nature, i.e., their behavior and properties may change depending on the invocation context. In this paper, we first demonstrate the inherent complexity in optimizing polymorphic functions, and then present the Vantage's *Collaborative Optimizer*, which is a cross-platform optimizer designed for optimizing the analytical functions invoked from within the SQL engine. The Collaborative Optimizer is the industry-first effort towards enabling analytics-aware optimizations over polymorphic analytical functions. We present a novel markup language-based approach for expressing the functions' polymorphic properties via a set of well-defined instructions. The Collaborative Optimizer uses these instructions at query time to infer the corresponding properties, and then decide on the applicable optimizations. From several possible optimizations, we showcase two core optimizations, namely *"projection push"* and *"predicate push"*, which aim at optimizing the data movement to and from the analytical functions. The experiments using the Teradata-MLE analytical system demonstrate the expressiveness power and flexibility of the proposed markup language. Moreover, benchmark and real-world customer queries show the significant performance gain that the Collaborative Optimizer brings to the Vantage system.

PVLDB Reference Format:
Mohamed Eltabakh, Anantha Subramanian, Awny Al-Omari, Mohammed Al-Kateb, Sanjay Nair and Mahbub Hasan, Wellington Cabrera, Charles Zhang, Amit Kishore, Snigdha Prasad. Not Black-Box Anymore! Enabling Analytics-Aware Optimizations in Teradata Vantage . PVLDB, 14(12): 2959-2971, 2021.
doi:10.14778/3476311.3476375

1 INTRODUCTION

Analytics and SQL-style processing are two integral components for most modern applications. Together, the two types of processing (SQL and analytics) provide an enriched environment with a broad spectrum of operations ranging from data transformations and filtering, aggregations, joins of multiple datasets, to data mining, machine learning, and deep analytics. Most of Teradata's customers strive for such type of integrated processing especially at the terabyte scale of big data.

In Teradata, the *Teradata Vantage* is an ecosystem that extends the Teradata's SQL engine with a broad range of analytical functions and capabilities. The analytical functions can be either *remote* or *native*. In the *remote* approach, the analytical functions run by separate dedicated analytical systems, e.g., Teradata-MLE [26][1], Spark [3], TensorFlow [31], etc., which may or may not be sharing the same physical cluster with the database system. The users' permanent data is stored in the database engine, and the data moves to and from the remote systems only as needed at query time. In contrast, in the *native* approach, the analytical functions along with their executable code if any, e.g., jar files, are embedded in and natively run within the database system.

The Vantage system enables treating the analytical functions as first-class citizens within the system, where end-users can seamlessly invoke the analytical functions from the SQL queries (see the invocation of `SESSIONIZE()` function in Figure 1), and build analytical pipelines of nested or joined functions intermixed with other relational tables or SQL sub-queries. Moreover, from the system's point of view, the analytical functions are fully integrated within the SQL query tree such that relational tables flow naturally from the SQL operators to the analytical functions and vice versa, and more importantly, the analytical functions are made eligible for and subject to various types of SQL-related optimizations.

In this paper, we focus on the challenge of: *How to optimize the execution of the SQL queries involving polymorphic analytical functions*? Treating the functions as black boxes is very inefficient and introduces unnecessary overheads both in resources and execution time. In contrast, capturing the functions' properties for the purpose of query optimization is very challenging not only because functions are too many and very diverse, but also because of their polymorphic nature, which means that a function's behavior and properties may dynamically change based on the query and invocation context [9, 30]. The example illustrated in Figure 1 highlights few questions central to the rest of this paper. For example, discovering the output schema that the function will generate is an essential step in query compilation, otherwise the rest of the query plan cannot be decided (*Q1* in the fig.). However, in polymorphic functions, the output schema may totally depend on the schemas of the input tables as well as the parameters passed to the function.

* The author Mohamed Eltabakh is a faculty member at WPI (meltabakh@wpi.edu).

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476375

[1] The Teradata-MLE is previously known as the *Aster Analytics* system [4].

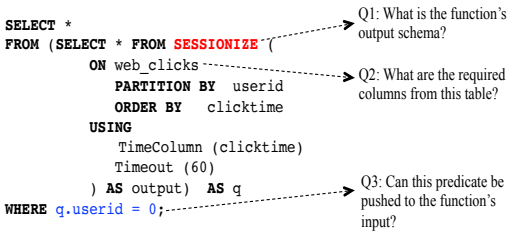


Figure 1: Example Query in Teradata Vantage.

Similarity, deciding on which columns from the input table(s) are actually needed by the function (*Q2* in the fig.) may depend on the passed parameters as well as the types of the other input tables involved in the invocation. Moreover, deciding on whether or not a post-function predicate can be pushed to the function's input entirely depends on the function's internal logic and behavior (*Q3* in the fig.).

In this paper, we present the *Vantage's Collaborative Optimizer* [8], which is a cross-platform optimizer that resides within the SQL engine. The Collaborative Optimizer role is to optimize the execution of the analytical functions invoked from within the SQL engine, and to address questions like those highlighted in Figure 1. The proposed optimizer is equally applicable to both native and remote functions alike. The Collaborative Optimizer approach is novel and unique compared to all other existing techniques proposed in literature, e.g., [5, 11, 15, 17] or adopted in commercial systems, e.g., [6, 13, 20, 23], which only focus on the integration aspect between the SQL and analytical processing. None of existing systems address the challenges of optimizing the analytical function execution, especially under the polymorphic execution model.

The integration aspect alone is a value-added feature to customers, yet the real value proposition lies in the ability to perform *analytics-aware* optimizations in the forthcoming next-generation processing platforms. The scope of this paper focuses on the following three key properties of analytical functions:

Output Schema Property: This property concerns the inference of the output schema that the analytical function will generate, which is essential for query compilation. Although the property does not help triggering runtime optimizations, it is more efficient compared to the existing mechanism highlighted in Section 3.

Input Schema Property: This property concerns the inference of the minimal set of columns required by the function from its input table(s) while still guaranteeing the generation of the same output. This property enables a core optimization, referred to as the *"projection push"* optimization, which entails the early elimination of unneeded columns before passing the input tables to the function.

Predicate Push Property: This property concerns the feasibility of pushing post-function predicates, i.e., predicates on the function's output, to the function's input table(s) while still guaranteeing the generation of the same output (e.g., *Q3* in Figure 1). This property enables another core optimization, referred to as the *"predicate push"* optimization, which entails the early elimination of unneeded rows before passing the input tables to the function.

To enable the inference of these properties, we design a novel engine-independent markup language (protocol) for expressing the functions' polymorphic properties via a set of well-defined instructions. These instructions are internally maintained as a function metadata attribute, referred to as the *function descriptor*. The instructions are designed to have a good expressiveness power to cover a wide range of functions. The Collaborative Optimizer retrieves the desired descriptors at query time, interpret their instructions over the invocation context to infer the properties, and then apply the applicable optimizations, e.g., *predicate* and *projection* push.

Although the optimizations of *predicate* and *projection* push are very primitive in the context of database systems over the well-defined SQL operators, it is far more challenging to enable these optimizations over analytical functions. This is because the analytical functions are too many, very diverse, involve complex logic, and many of them are polymeric. Typically, end-users may have limited knowledge on both the SQL syntax and the functions' internal properties, and hence they are only expected to plug in the names of the input tables and fill in the parameters' values. Then, the system should be responsible for enabling all applicable optimizations transparently, which is the job of the Collaborative Optimizer.

The rest of this paper is organized as follows. In Section 2, we present preliminaries and background. In Sections 3 and 4, we introduce the function descriptors, and the details of the markup language, respectively. The exploitation of the descriptors in query re-writing and optimization is presented in Section 5, and the related work is presented in Section 6. Finally, the experimental evaluation and the conclusion remarks are included in Sections 7 and 8.

2 PRELIMINARIES

For the sake of this paper, we assume that end-users interact with the Vantage system by submitting SQL queries involving invocations to analytical functions as depicted in Figure 1. Each remote analytical system must be registered in the Teradata Vantage to facilitate the communication with that system through the Vantage's network layer. In addition to the system-level registration, each individual remote function needs to be also registered within the SQL engine through a mechanism (SQL command) called `CREATE FUNCTION MAPPING`. This mechanism simply links the remote function in a specific system to a unique internal user-defined alias, which is later used for function invocation.

In the following subsections, we briefly overview the functions' invocation interface within a SQL query, which is a SQL construct called *Table Operator*, and the polymorphism property.

2.1 Table Operator and Execution Model

In Teradata Vantage, table operators [1] are the building blocks for implementing and invoking the analytical functions from a SQL query. They receive as inputs one or more data tables and a set of parameters, and produce as output a table that can be consumed by a parent SQL query. The basic invocation syntax of a table operator from a SQL query is as illustrated in the examples of `SESSIONIZE()` (Figure 1), `UNPIVOT()` (Figure 2(a)), and `DTW()` (Figure 3(a)). In general, table operators accept two types of inputs:

(1) One or more data tables, each is specified in a separate `ON` clause. Each `ON` clause may be optionally qualified with the sub-clauses of `PARTITION BY expressions>` and `ORDER BY <expressions>`,

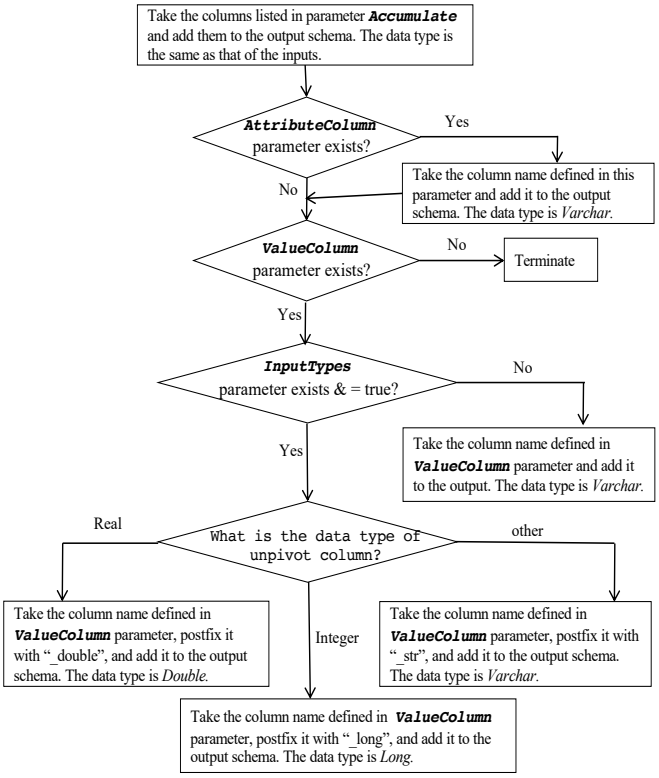
```
SELECT * FROM Unpivot (
  ON (SELECT * FROM input_timeseries_table)
  USING
    Unpivot ({ 'unpivot_column' | 'unpivot_range' }[,…])
    Accumulate ({ 'accumulate_column' | 'accumulate_column_range' }[,…])
    [ InputTypes ({'true'|'false'}) ]
    [ AttributeColumn ('attribute_column') ]
    [ ValueColumn ('value_column') ]
);
```

(a) The Unpivot Function Syntax

Column Name*	Data Type	Description
accumulate_column	Same as in input table	Columns specified in Accumulate parameter. Copied from the input table.
attribute_column	VARCHAR	Unpivoted attribute.
value_column	VARCHAR	Appears when InputTypes('false'). Contains the unpivoted value of the corresponding attribute. Numeric values are cast to VARCHAR.
value_column_double	DOUBLE	Appears when InputTypes('true') and an unpivot column has a real data type. Contains the unpivoted value of the corresponding attribute if the value is real; NULL otherwise.
value_column_long	LONG	Appears when InputTypes('true') and an unpivot column has an integer data type. Contains the unpivoted value of the corresponding attribute if the value is integer; NULL otherwise.
value_column_str	VARCHAR	Appears when InputTypes('true') and an unpivot column has a data type other than real or integer. Contains the unpivoted value of the corresponding attribute.

* The black italic part in the column names indicates that it is dynamic string that depends on the inputs. Whereas, the blue non-italic part indicates that it is a fixed string.

(b) The Specifications of the Function's Output Schema (from the Manual)



(c) The Flowchart for Constructing the Function's Output Schema

Figure 2: Unpivot Function from Teradata-MLE.

or labeled as a dimension table using keyword `DIMENSION`. Dimension tables are typically broadcasted to all worker machines (called AMPs in Teradata). The ON clause tables can be either base tables referenced by name, or derived tables generated from any arbitrary SQL query, which may involve nested table operators.

(2) A set of parameters defined in the `USING` clause, each parameter is in the form of name-value pair, where each value is either a single string or multiple comma-separated strings. As defined in the manual of each function, some parameters may be mandatory while others are optional.

For example, referring to the `SESSIONIZE()` function in Figure 1, the function receives one input table, i.e., *web_clicks*, and this table is partitioned by the *userid* column and each partition gets sorted by the *clicktime* column. The function also receives two input parameters, namely *TimeColumn*, which specifies the name of the column carrying the clicks timestamps, and the *Timeout* parameter, which specifies the ideal time between consecutive clicks to break sessions. If the `PARTITION BY` clause is not present for a non-dimension input table, e.g., the input to the `Unpivot()` function in Figure 2(a), then the table is randomly partitioned. Finally, the example of the `DTW()` function in Figure 3(a), illustrates passing three tables to the function, where one of them is a dimension table, as well as a set of parameters, where the first four parameters are mandatory, and the last three, i.e., the ones in-between square brackets, are optional.

Function Execution Model: The execution model of table operators is very similar to the map-reduce model in Hadoop [32]. The processing of a table operator can be viewed as three phases; *record-level manipulation*, *data partitioning*, and finally *partition-level manipulation*. For example, referring to the `SESSIONIZE()` function in Figure 1, since the ON clause directly refers to a base table name, i.e., *web_clicks*, then there is no *record-level manipulation*, otherwise a full-fledged SELECT statement could have been used to perform any of such manipulations. Then, the input table is partitioned by the *userid* column and each partition gets sorted by the *clicktime* column. After that, each partition is sent to a worker node (called "AMP" in the Teradata distributed system). These AMPs will also receive any specified parameters and any dimension tables as broadcasted tables.

In the case the table operator is implementing a native analytical function, the code and logic of the table operator that gets executed over each partition is defined in an external language, e.g., using Java/C/C++/Python code. The external function is linked to the table operator during its creation time using CREATE FUNCTION command. After this logic is applied over each partition, the function produces its output in the form of a relational table to continue the execution of the outer (parent) SQL query. In the case the analytical function is remote and executes in an external engine, then the created table operator is just an interface to the remote function. In this case, the SQL engine prepares all inputs, e.g., derived tables and parameters, applies any specific partitioning or ordering defined in the ON clauses, and then passes the data to the remote system. Finally, the function's output is sent back in the form of a relational table to continue the execution of the outer SQL query.

```
SELECT * FROM DTW (
  ON input_table AS input_table
     PARTITION BY i_partition_column [,...]
     ORDER BY i_ordering_column [,...]
  ON template_table AS template_table DIMENSION
     ORDER BY t_ordering_column [,...]
  ON mapping_table AS mapping_table
     PARTITION BY m_partition_column [,...]
  USING
      InputColumns ('i_value', 'i_timestamp')
      TemplateColumns ('t_value', 't_timestamp')
      TimeseriesID ('timeseriesid' [,...])
      TemplateID ('templateid' [,...])
      [ Radius ('radius') ]
      [ DistMethod ('distance_metric') ]
      [ WarpPath ({'true'|'false'})]
);
```

* The blue aliases are mandatory aliases such that the function can identify the different input tables (with mandatory aliases, the order is not important).

(a) The DTW Function Syntax

Input table (ON Clause)	Mandatory Columns (Needed by the function)
input_table	The column(s) defined in its ***PARTITION BY***
	The column(s) defined in its ***ORDER BY***
	The columns defined in parameter ***InputColumns***
template_table	The column(s) defined in its ***ORDER BY***
	The columns defined in parameter ***TemplateColumns***
...	...

(b) The Specifications of the input Schemas (from the Manual)

Figure 3: Dynamic Time Warping Function from Teradata-MLE.

2.2 Functions Polymorphism

Polymorphic functions, which are part of SQL:2016 [19], are functions that accept different types of inputs, their behavior and processing may change based on these inputs, and their outputs may also differ depending on the inputs they receive. Most analytical functions are inherently polymorphic. In the following examples, we demonstrate the polymorphism property using functions from the Teradata-MLE system, which offers approximately 180 analytical functions ranging from clustering, classification, predication, sentiment analysis, statistical methods, among others [2].

Example 1: Output Schema of `Unpivot` Function

The `Unpivot()` *function pivots data that is stored in columns into rows. The invocation syntax of the function and the specifications of its output schema from the function's manual are depicted in Figures 2(a) and (b). The function accepts one input table (one ON clause), two mandatory parameters, i.e.,* `Unpivot` *and* `Accumulate`*, and three optional parameters, i.e.,* `InputTypes`*,* `AttributeColumn`*, and* `ValueColumn`*. For the sake of clarity, we translate the output schema specifications in Figure 2(b) to the flowchart presented in Figure 2(c). Few observations from the flowchart include: (1) the output column names are dynamic and they are listed in some parameters, e.g.,* `Accumulate`*, and these names are referencing columns in the input table, (2) the data types of output columns inherit the corresponding types from the input table, (3) there are branching and conditional actions based on the presence or absence of some parameters as well as the data types of some columns, (4) the listing of the column names within a parameter can be explicit or by column positions and ranges, and (5) there are manipulation operations on the column names to, for example, postfix the names of some columns with fixed string (see the blue-marked string {"_double", "_long", "_str"} in Figure 2(b)).*

Example 2: Input Schema of `DTW` Function

Dynamic Time Warping (DTW) is a similarity measure algorithm over time series data. The invocation syntax of the function is depicted in Figure 3(a), where it takes three mandatory input tables (three ON clauses), four mandatory parameters, and three optional parameters. The function accepts any schema and any number of columns in each of the three ON clauses. However, the columns of interest to the function must be defined inside the parameters. The table in Figure 3(b) shows the required columns needed by the function for each input ON clause. Few observations from Figure 3 include: (1) Each ON clause has a pre-defined alias, which is indicated in blue in Figure 3 (a). Given these pre-defined aliases, the order among the ON clauses is arbitrary. For other functions, there might not be pre-defined aliases, and in these cases, the order of the ON clauses is strict. (2) The mandatory columns needed by the function from each ON clauses are different as illustrated in Figure 3(b). For example, the table with alias "input_table" *may have 100s of columns, but the relevant ones for the function are only those defined in the* `Partition By` *and* `Order By` *sub-clauses as well as the names listed in parameter* `InputColumns`*. (3) The mandatory columns of a given ON clauses can be either coming from its sub-clauses or parameters, and they cannot reference other ON clauses (as exemplified in Figure 3(b)).*

3 FUNCTION DESCRIPTOR LIFECYCLE

The analytical functions in Teradata Vantage are made eligible for optimizations through a metadata object, referred to as the *"function descriptor"*. The descriptor of a function is a JSON-based document that captures specific properties of interest to the query optimizer to help generating better execution plans, e.g., the *"outputSchema"*, *"inputSchema"*, and *"predicatePush"* properties. Each of these properties is an array of other JSON documents containing instructions for inferring the property's value at query time (Sections 4 and 5). In this section, we describe the lifecycle of a descriptor, which includes:

(1) *Creation.* In the first release of the Collaborative Optimizer feature, the descriptors are solely created by the Teradata system engineers who are implementing and integrating the function with the Vantage system. Therefore, the descriptors are shipped as part of the entire system. In the experiment section, we provide more details on the descriptors' creation process.

(2) *Loading*: During the installation of the function in the Vantage system, the descriptors are uploaded to the database system and stored in a dictionary table, which maintains one entry per installed function. For performance reasons, we opt to store the descriptors in BSON binary format, which allows for better storage utilization and faster retrieval and parsing at query time.

(3) *Retrieval and Parsing*: Given a query involving one or more analytical functions, the Collaborative Optimizer retrieves the corresponding descriptors from the dictionary table at query compilation time, parses the JSON fields of a descriptor, and then builds a main memory data structure containing all the details of the stored instructions. This structure is cached for the query lifetime such that if a query invokes the same function multiple times, there will be a single retrieval from the dictionary table.

(4) *Interpretation and Optimizations*: The last phase involves applying the descriptor's instructions over a specific function invocation within the query. This step is context dependent and results in inferring the function's properties given that context, e.g., the required columns that the function needs given the input schemas. Finally, the Collaborative Optimizer triggers the applicable re-write optimizations by, for example, projecting out certain columns or pushing a post-function predicate to the function's inputs.

4 MARKUP LANGUAGE

4.1 Design Principles and Language Coverage

The proposed markup language is the language in which the function descriptors are expressed, and it consists of a set of instructions in JSON format. The key design principles guiding the design of the markup language are:

Interoperability: The language is engine-independent and is designed to facilitate the communication between heterogeneous analytical engines, including built-in analytical functions within a database system, and the Collaborative Optimizer.

Expressiveness: The language is expressive enough to cover a wide range of functions. We analyzed dozens of functions and extracted polymorphic patterns regarding what needs to be referenced within a function's invocation body and how it is referenced. The language's instructions are designed to cover these patterns.

Extensibility: As needed, additional instructions can be added to the language to expand its coverage and expressiveness power. Moreover, the two optimization types explored in this paper, namely *projection push* and *predicate push*, are only two examples of many other optimizations. The language can be extended to capture function properties for enabling cardinality estimation, i.e., the expected output size based on a given input size and other invocation parameters, and join-related optimizations, i.e., switching the order between a join operator and an analytical function based on the function's semantics.

Transparency: The language and its detailed instructions can be viewed as part of the engine's backend, which is entirely transparent to end-users who submit their analytical queries to the system. Even more, for the system admins and engineers who are expected to generate the functions' descriptors, they typically express the function's properties through a user-friendly interface, which automatically generates the descriptor. Therefore, the learning curve to leverage the markup language is minimal.

Coverage: The language is not intended to cover all possible functions and scenarios as that may excessively increase the language complexity. In general, the markup language is designed to cover the class of functions exhibiting ***"Invocation-dependent polymorphism"***, i.e., functions whose polymorphic characteristics can be decided solely based on the invocation content rather than the input data content passed to the function. Nevertheless, other function classes that exhibit ***"data-dependent polymorphism"***, i.e., depend on the input content or intermediate data state, or exhibit ***"cluster-dependent polymorphism"***, i.e., depend on the cluster configuration are not currently supported.

In the rest of this section, we present the three main instructions of the markup language, namely `ADD`, `CASE`, and `LOOP`.

4.2 Building Block `ADD` Instruction

The ADD instruction is used to add column information to the target list, either the *output* or *input* schema list. The main design elements of the ADD instruction are: (1) Ability to reference the different fragments within a function's invocation from which the columns will be inherited. This includes the input clause(s) and their PARTITION BY and ORDER BY clauses, and the input parameters, (2) Ability to add columns newly introduced by the function and in this case defining their data types, (3) Ability to apply some manipulations (if needed) on these columns, e.g., concatenating multiple column names into one or prefixing the column names with sequential numbers, etc., and (4) Ability to control the position in which the column names are augmented. The order of columns matters especially for the input schema. The design of the ADD instruction illustrated in Figure 4(a) captures these requirements.

The `ADD` instruction involves three levels (types) of nested JSON documents as illustrated in Figure 4(a). For ease of reference, we refer to them as `ADD-D1`, `ADD-D2`, and `ADD-D3`.

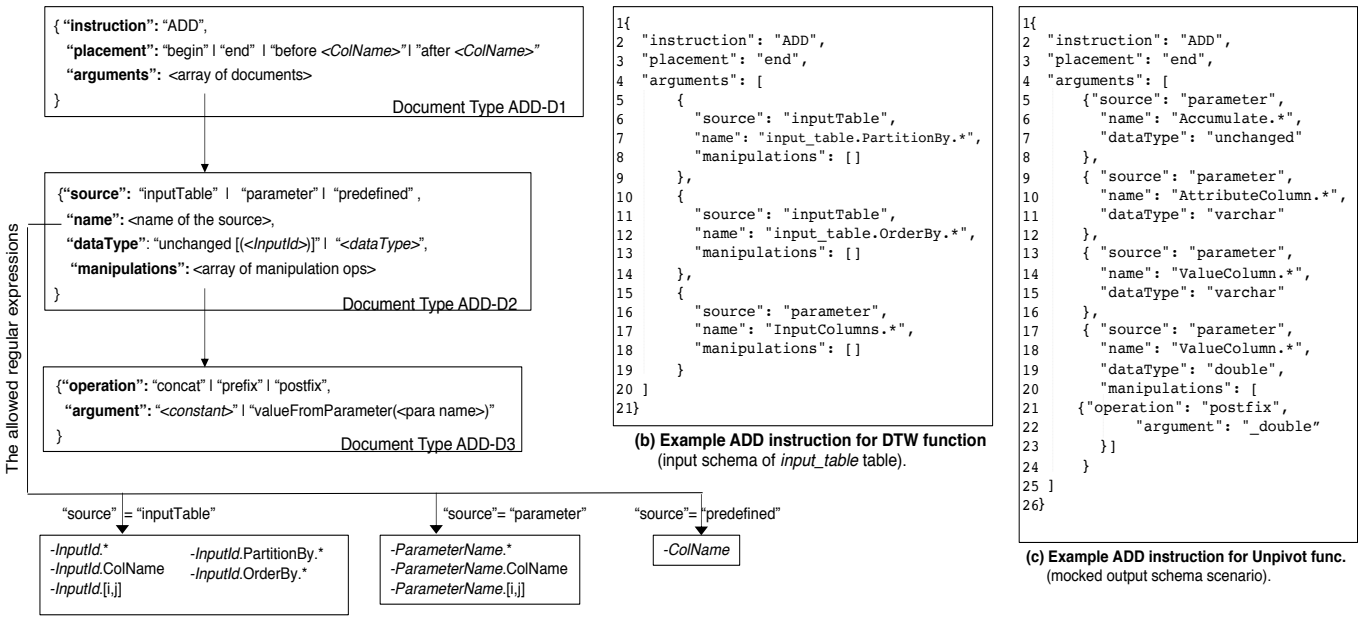
• `ADD-D2."source"`: The added columns come from one of three possible sources as follows: (1) *"inputTable"*, which is one of the ON clauses in the invocation body, (2) *"parameter"*, which is one of the parameters in the USING clause, or (3) *"predefined"*, which are columns that do not depend on the inputs, e.g., columns that the function adds to the output regardless of the input.

• `ADD-D2."name"`: The exact columns' information from the specified source are defined in the *"name"* field. The value of this field is one of the simple regular expressions depicted at the bottom of Figure 4(a). For example, in the case the *"source" = "predefined"*, then the name of the column is specified in the *"name"* field. In contrast, in the case the *"source" = "inputTable"*, then there are six regular expressions possibilities, which for example, allow referencing all columns from a specific input, e.g., *"inputId.*"*, the columns in the Partition By clause, e.g., *"inputId.PartitionBy.*"*, specific column range, e.g., *"inputId.[i,j]"*, among few other options. The *"inputId"* is a unique identifier specific to each ON clause in the invocation body. We will discuss later in this section the two possible referencing mechanisms for the ON clauses, namely *position-based* or *alias-based* referencing.

• `ADD-D2."manipulations"`: This field is optional and contains an array of possible manipulation operations that can be applied over the column names before inserting them into the list. The current supported operations are highlighted in document type `ADD-D3`, which include concatenating the column names together to form one column name (*"concat"* operation), and prefixing (or postfixing) each of the column names with a specific string (*"prefix"* (or *"postfix"*) operations, respectively).

Example 3: Applying the `ADD` Instruction to `DTW()`

Referring to the DTW function in Figure 3, lets define the input schema of one of its tables, say the one with alias "input_table". *We need one* `ADD` *instruction with three arguments as illustrated in Figure 4(b). Notice that the first two documents under* "arguments" *array indicate that the columns will come from the Partition By and Order By sub-clauses of the ON clause with alias* "input_table". *Whereas the* 3^{rd} *document indicates that the rest of the needed columns by the function will come from parameter* "inputColumns". *It is also worth*



(a) Specifications of the ADD Instruction in the Markup Language.

(b) Example ADD instruction for DTW function (input schema of *input_table* table).

(c) Example ADD instruction for Unpivot func. (mocked output schema scenario).

Figure 4: Specifications and Examples of the ADD Instruction.

noting that the "manipulations" *field can be set to an empty array as in the figure, or totally omitted. Each of the other two inputs, namely* "template_table" *and* "mapping_table" *will have very similar* ADD *instructions to define minimal columns from their schemas needed by the function.*

Example 4: Applying the ADD Instruction to `Unpivot()`

Referring to the Unpivot function in Figure 2, lets assume for the sake of simplicity that the output schema specifications of this function contain only the first four rows in Figure 2(b) without any conditions involved. Later, the Unpivot function will be revisited after introducing the CASE *instruction. Under this simplifying scenario, the first four rows can be translated to the* ADD *instruction in Figure 4(c).*

Input Table Referencing: An analytical function may have multiple input tables specified in its invocation, which is expressed as multiple ON clauses in the invocation body (see Figure 3(a) for an example). The markup language supports two referencing mechanisms that can be used depending on the specifications of each function.

Position-Based Referencing, in which a reference to a specific input is achieved by the position of its ON clause relative to the other ON clauses, i.e., *"input1"* and *"input2"* are keywords that reference the 1^{st} and 2^{nd} ON clauses, respectively. This referencing scheme is applicable only if the order of the inputs in the function's invocation is fixed and there are no optional ON clauses.

Alias-Based Referencing, in which a reference to a specific input is achieved by its alias. For example, referring to the `DTW()` function in Figure 3(a), the main and template tables to the function can be referenced using their aliases, i.e., *"input_table"* and *"template_table"*, respectively. This referencing scheme is applicable only if the function's manual mandates specific aliases to be given to the inputs. In this case, the relative order among the ON clauses in not important, and hence the instructions must use the table aliases (see Figure 4(b) for an example).

4.3 Building Block CASE Instruction

The `CASE` instruction is used in the scenarios where a branching or conditional action is needed, e.g., the conditions in the `Unpivot()` flowchart in Figure 2(b). The main design elements of the CASE instruction are: (1) Ability to reference and check parameters, either checking their presence or absence or checking the value in a given parameter, and (2) Ability to specify a list of actions under each branch, which includes defining ADD and LOOP instrcutions. The markup language does not currently support nested `CASE` instructions as we did not find a strong need for such complexity based on the observed functions.

The `CASE` instruction consists of three levels of documents as illustrated in Figure 5(a), referred to as `CASE-D1`, `CASE-D2`, and `CASE-D3`. The instruction is designed in a very similar way to the CASE statement in programming languages. For example, the instruction consists of an array of conditions (branches), and a set of actions associated with each branch. At most one branch can evaluate to True and get executed, and then all subsequent branches are skipped. It is possible that none of the branches evaluates to True, and as a result none of actions is executed. In the following, we highlight the description of some fields in the instruction.

• `CASE-D2."condition"`: This is an array of conjunctive conditions, i.e., an array of documents of type `CASE-D3`, that together represent the condition part of one branch. If these conditions evaluate to True, then the action(s) associated with this branch, which are specified in the `CASE-D2."action"` field, get executed. The actions

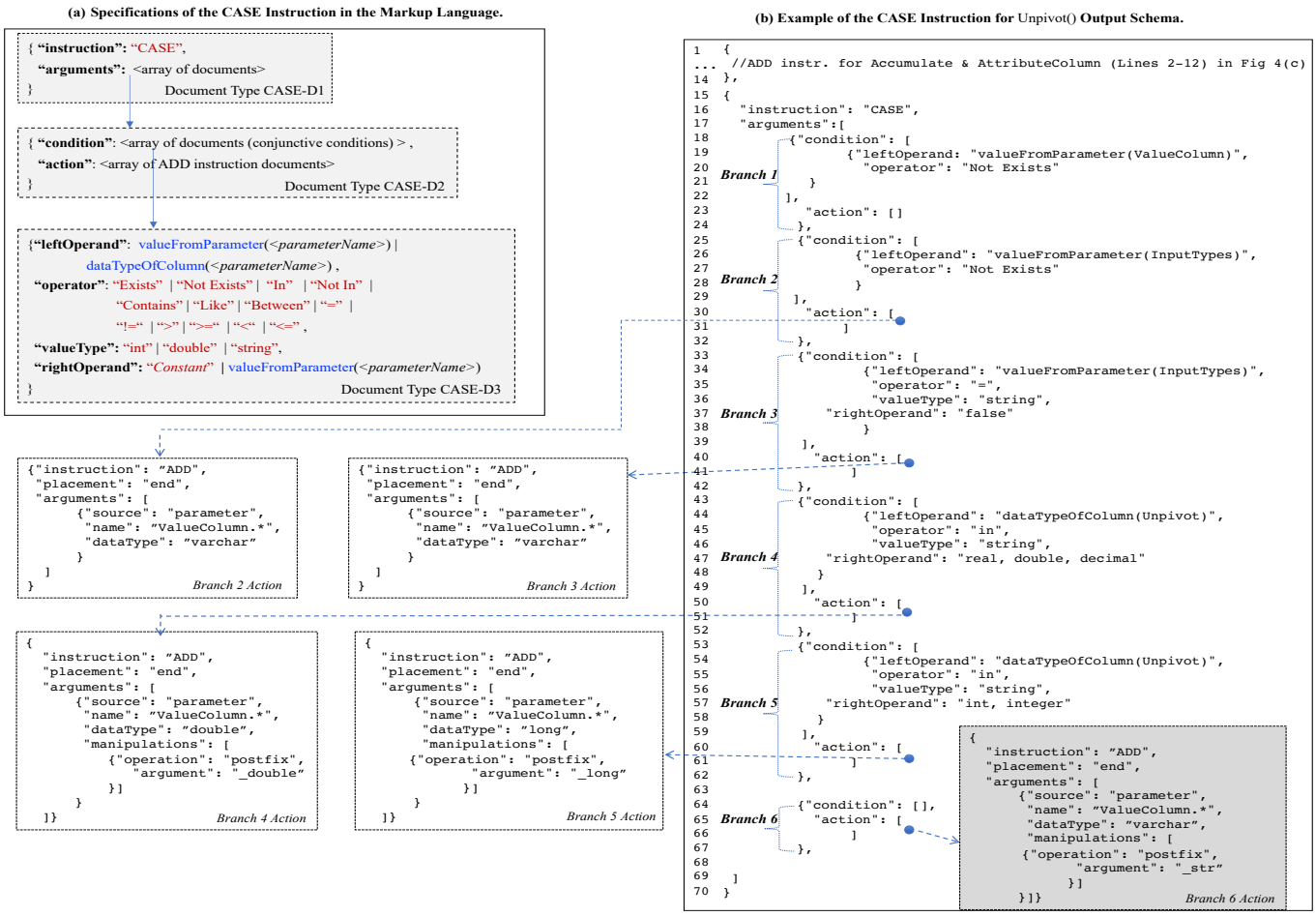


Figure 5: Specifications and Examples of the CASE Instruction.

in CASE-D2."action" are simply an array of the ADD instruction introduced in Section 4.2.

• CASE-D3: This is a document for expressing one condition. As illustrated in Figure 5(a), the document includes specifications for: the left-side and right-side operands, the comparison operator, and the data type used for the comparison. The left-side operand is either a value retrieved from an invocation parameter in the USING clause, i.e, the *"valueFromParameter(<parameterName>)"* syntax, or a data type of a column name specified in a parameter, i.e, the *"dataTypeOfColumn(<parameterName>)"* syntax. In contrast, the right-side operand is either a constant value or a value retrieved from a parameter.

Example 5: Applying the CASE Instruction to Unpivot()

We can now re-visit Example 4, and consider the complete output schema flowchart of the Unpivot() *function depicted in Figure 2(b). The flowchart can be expressed using the* ADD *and* CASE *instructions as illustrated in Figure 5(b). We will use this example to highlight few additional features of the markup language. Lines 2-12 express a single* ADD *instruction to inserting the column names in parameters* Accumulate *and* AttributeColumn *to the output schema (identical to those in Figure 4(c)).*

Following the ADD *instruction, a* CASE *instruction is expressed to insert the remaining columns (Lines 15-70). As illustrated, there are six main branches, each including a condition-action elements. Branches are evaluated in sequence, and once a branch's condition is matched, the subsequent branches are not considered. The action element of branches 4, 5, and 6 involves a* "manipulation" *operation to postfix the column name specified in parameter* ValueColumn *with a constant string.*

4.4 Building Block LOOP Instruction

The following example shows a type of functions for which the number (and possibly the name) of the inserted columns to the target schema, e.g., the output schema, depends on a value of a parameter. For these functions, the ADD and CASE instructions are not sufficient to express their polymorphic properties, and there is a need for the LOOP instruction introduced in this section.

Example 6: Output Schema of PCAPlot Function

The PCAPlot() *function calculates a set of principal components, and each principal component is a linear combination of the set of original predictors. The invocation syntax and the output schema specifications are illustrated in Figures 6(a) and (b), respectively. More*

```
SELECT * FROM PCAPlot (
   ON input_table AS inputtable PARTITION BY ANY
   ON pca_table AS pca_table DIMENSION
   USING
      Components ('num_components')
     [Accumulate({ 'accumulate_column'}[,...]) ]
);
```

(a) The PCAPlot Function Syntax

```
SELECT *
FROM PCAPlot ( ON ... ON ...
   USING Components (3)
      Accumulate('pid, strokes'));
```

{pid, strokes,principal_component_1, principal_component_2, principal_component_3}

(c) Example of an output schema

Column Name	Data Type	Description
accumulate_column	Same as in input table	Columns specified in Accumulate parameter. Copied from the input table.
principal_component_*i*	double	The principal component columns. The number of columns depends on the value of parameter *Component*. The postfix "*i*" is the component number ranging from 1 to the parameter value.

(b) The Specifications of the Function's Output Schema (from the Manual)

Figure 6: Principal Component Analysis Func. from Teradata-MLE.

specifically, the output schema consists of the column names specified in the Accumulate *parameter plus a set of columns whose number matches the value specified in the* Components *parameter and their names start with string "principal_component_" followed by a sequential number. Figure 6(c) shows an example of invocation parameters and the corresponding output schema.*

The LOOP instruction adds iteration capabilities to the markup language for expressing the type of functions highlighted in Example 6. The main design elements of the LOOP instruction are: (1) Ability to specify the number of iterations, which may come from a predefined constant, a parameter's value, or an enumerated list of values within a parameter, (2) Ability to reference the iteration index, which either gets modified in a static manner, e.g., gets incremented in each iteration, or hops over enumerated list, and (3) Ability to integrate the iteration index into the specifications of the column names, i.e., within the ADD instruction.

The LOOP instruction consists of a single JSON document, referred to as LOOP-D1, as depicted in Figure 7(a). The example in Figure 7(b) illustrates the usage of the LOOP instruction to define the output schema of the PCAPlot() function.

5 QUERY OPTIMIZATION

In this section, we zoom out from the markup language instructions to the top-level function descriptor, which is one JSON file per function. The structure of the descriptor is highlighted in Figure 8(a), where it contains one JSON element for each of the main properties. In this section, we focus on two semantic properties, namely *inputSchema* and *predicatePush*, which enable the two core optimizations of *projection push* (Section 5.1) and *predicate push* (Section 5.2) , respectively. These two optimizations are examples of *rule-based* optimizations since they are guaranteed to improve performance and there is no potential benefit from carrying and processing unnecessary data. Nevertheless, other optimization types, including cost-based optimizations, are also applicable but beyond the scope of this paper.

5.1 Input Schema Specifications

Document Structure: Since functions may have multiple input tables (see the DTW() function in Figure 3(a)), the *"inputSchema"*

{ **"instruction"**: "LOOP",
"numIterations": "*<constant>*" | "valueFromParameter(<parName>)" | "EnumFromParameter(<parName>)",
"startIndex": "0" | "1",
"arguments": <array of ADD or CASE instruction documents>
}
Document Type LOOP-D1

(a) Specifications of the LOOP Instruction in the Markup Language.
Two reserved variables inside the *LOOP* instruction:
"*iteration.index*": returns the iteration number.
"*iteration.value*": Only valid when "EnumFromParameter()" is used. In each iteration, it carries the next value from the comma-separated values specified in "numIterations" field.

```
{
  "instruction": "ADD",
  "placement": "end",
  "arguments": [
      {"source": "parameter",
        "name": "Accumulate.*",
        "dataType": "unchanged"
      }]
},
{
  "instruction": "LOOP",
  "numIterations": "valueFromParameter(Components)",
  "startIndex": "1",
  "arguments": [
        {"instruction": "ADD",
         "placement": "end",
         "arguments": [
               {"source": "predefined",
                "name": "principal_component_",
                "dataType": "double",
                "manipulations": [
                       {"operation": "postfix",
                        "argument": "iteration.index"
                  }]
         }]
   }]
}
```

(b) PCAPlot() Instructions

Figure 7: Specifications and Examples of the LOOP Instruction.

JSON element in the descriptor contains an array of documents, each describes the minimal columns required for a specific ON clause. The structure of a single *inputSchema* document is depicted in Figure 8(b). The document contains two straightforward fields *"inputId"* and *"instructions"*. The former references a specific ON clause either by its position or by its alias depending on the function (refer to the last paragraph in Section 4.2 for the description of the *alias-based* vs. *position-based* referencing mechanisms). The latter field contains an array of instructions (ADD, CASE, LOOP) that specify how to infer the minimal input schema at query time.

The document contains an additional field, namely *"surplus"*, which is important in deciding the projection push optimization and the elimination of the unneeded input columns. This field specifies the behavior of the function with respect to the additional columns beyond the mandatory ones if passed to the function. The allowed values are:

- *"surplus" = "notAllowed"*: This value indicates that it is not allowed to send additional columns to the function beyond what it needs, otherwise the function fails. For functions conforming to this behavior, the elimination of the unneeded columns can be viewed as a value-added functionality since the end-users do not need to exactly know the columns needed by the function. Instead, the Collaborative Optimizer projects out the extra columns, which leads to a successful function execution.
- *"surplus" = "ignored"*: This value indicates that if additional columns are sent to the function beyond the needed ones, these columns are ignored (dropped) by the function. Clearly, applying

the projection push optimization to early eliminate these columns is very critical for better performance.

- *"surplus" = "propagatedBack"*: This value indicates that if additional columns are sent to the function beyond the needed ones, these columns are carried (copied) from the input to the output. These extra columns are called *"pass-through"* columns. Functions that conform to this behavior are typically row-based functions that operate on each row independently.

Exploitation in Query Optimization: The goal from the input schema specification is to identify the minimal columns required by the function for each ON clause and the elimination of any unnecessary columns. For the cases of *"surplus" = "notAllowed"* and *"surplus" = "ignored"*, the projection push optimization is guaranteed that passing any extra columns to the function will not bring any benefit to the entire SQL query, and hence it is applied as a rule-based optimization. The following example demonstrates this case.

Example 7: Case ("surplus" = "ignored"**)**

Consider the following SQL query involving the DTW() *function. Recall that the function's properties are highlighted in Figure 3.*

```
SELECT * FROM DTW (
  ON timeSeriesPrimary AS input_table
     PARTITION BY ts_id  ORDER BY timestamp1
  ON ... ON ...
  USING
     InputColumns ('temperature', 'timestamp1')
     ...
);
```

Now, lets assume that table timeSeriesPrimary *has the following schema:*

```
timeSeriesPrimary(ts_id, timestamp1, temperature,
                  metaCol_1, metaCol_2, …, metaCol_k)
```

Focusing on the input schema specifications of the 1^{st} *ON clause for which the instructions are presented in Figure 4(b). The re-written query is thus:*

```
SELECT * FROM DTW (
  ON (SELECT ts_id, timestamp1, temperature
      FROM timeSeriesPrimary) AS input_table
     PARTITION BY ts_id
     ORDER BY timestamp1
  ON ...
  ON ...
  USING
     InputColumns ('temperature', 'timestamp1')
     ...
);
```

5.2 Predicate Push Specifications

Document Structure: The "predicatePush" property in the descriptor (refer to Figure 8(a)) specifies some properties that help deciding whether or not a post-function predicate can be safely pushed to the function's input. A predicate can be pushed to the function's inputs on either a single ON clause or multiple ON clauses. For example, a function may have three ON clause inputs, the first two specify primary tables that are co-grouped, i.e., partitioned in the same way and have the same schema, while the 3^{rd} ON clause is a dimension broadcast table. It can be the case that a predicate p on the function's output can be pushed to only the first two ON clauses.

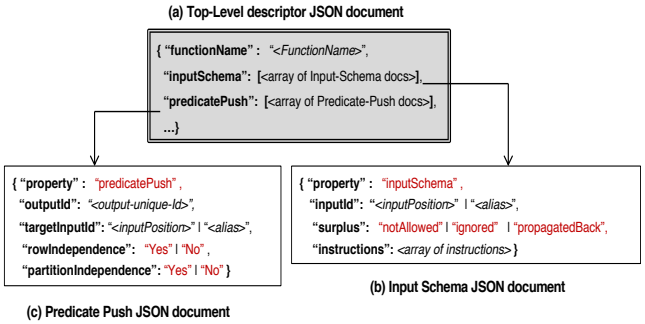


Figure 8: Function Descriptor Structure and its Sub-Documents.

The two key fields in the document for capturing the function's semantics regarding predicate pushing are:

- *"rowIndependence"*: This property is set to *"Yes"* only if the function satisfies the following three conditions, otherwise the property is set to *"No"*: (1) The function operates on each row independent of the other rows, and (2) The function does not alter the values of the input rows although subset of the columns may be dropped, and (3) The function does not alter the input column names.
- *"partitionIndependence"*: This property is set to *"Yes"* only if the function satisfies the following three conditions, otherwise the property is set to *"No"*: (1) The function operates on each partition independent of the other partitions, and (2) The function does not alter the values of the partitioning columns rows although subset of the columns may be dropped, and (3) The function does not alter the names of the partitioning columns.

Exploitation in Query Optimization: The *predicate push* optimization can significantly reduce the data transfer overhead as well as the function's execution time. Moreover, it can enable the generation of more efficient query plans by possibly leveraging available access paths. In the case where "rowIndependence = Yes", a post-function predicate on any of the input columns can be pushed as a pre-function predicate and gets evaluated on the function's input(s). Similarly, if "partitionIndependence = Yes", then a post-function predicate on any of the partitioning columns of the specified ON clause can be pushed as a pre-function predicate to that ON clause.

Example 8: *Consider the SQL query presented in Section 1 in Figure 1 involving the* SESSIONIZE() *function. The predicate push specifications for the function (the left-hand side) and the function invocation re-writing (the right-hand side) are as follows:*

```
...
{ "property" :"predicatePush",
  "outputId": "standard",
  "targetInputId": "1",
  "rowIndependence": "No" ,
  "partitionIndependence": "Yes"
}
```

```
SELECT * FROM (SELECT *
     FROM SESSIONIZE (
       ON (SELECT * FROM web_clicks
           WHERE userid = 0)
        PARTITION BY  userid
        ORDER BY   clicktime
        USING ...) AS output)  AS q;
```

The Collaborative Optimizer will recognize that the post-function predicate is on the same partitioning column of the 1^{st} *ON clause, which is the same ON clause referenced in the JSON document. Therefore, the depicted re-writing becomes applicable.*

6 RELATED WORK

Several projects have been proposed in literature or adopted in commercial systems for enabling seamless integration between the SQL engines and analytical functions and systems [11, 16–18, 20, 23, 25, 33]. However, to the best of our knowledge, there is no previous effort on addressing the challenges of optimizing the analytical functions within such integrated ecosystems, especially under the complex model of polymorphism. In that sense, the Collaborative Optimizer is fundamentally distinct from existing work. The support for polymorphic functions is reported in several systems such as Oracle [7] and SAP databases [28]. Nevertheless, optimizing these functions is not previously proposed.

The MLog project [17] introduces a set of new well-defined operators, e.g., slicing, linear algebra, and matrix manipulation operators, to operator on relations (aka *"tensors"*). They propose some optimizations over these operators. Nevertheless, any function not expressed in these operators is treated as black box, and hence not subject to optimizations. In contrast, the work in [5, 11] proposes a scalable in-database analytics library, called *MADlib*, implementing a handful of analytical functions using extended SQL and Python scripts. MADlib is adopted by some commercial DBMSs such as Greenplum [13]. Other DBMSs such as Snowflake [6] runs its analytics through integration with external analytical engines, e.g., R, SAS, alteryx, among others. However, these existing projects focus only on the integration aspect and they do not offer analytics-aware optimizations as proposed by the Collaborative Optimizer approach.

The work in [14] proposes a UDF model for capturing specific UDF semantics as a composition of local map or reduce functions, and then leverages this model for re-using previous results. The work in [14] is engine specific and targets Map-Reduce frameworks like apache Hive and Pig, whereas the Collaborative Optimizer markup language is engine-independent. Second, the work in [14] cannot model polymorphic functions unless the each possible invocation signature is modeled as a separate function, whereas the Collaborative Optimizer markup language allows expressing the polymorphism programmatically (using instructions), which gets interpreted at run-time to derive the function's properties. Third, the two approaches target distinct optimizations, i.e., the Collaborative Optimizer optimizes a function's input, whereas the work in [14] optimizes the usability of the function's output.

More recent work such as the *Froid* system [27] proposes optimizing imperative UDFs in relational database systems by automatically transforming the UDF logic into relational algebraic expressions and embeds them into the calling SQL query. Froid system, however, focuses only on scaler open-box functions expressed in languages such as Transact-SQL (T-SQL). In contrast, the Collaborative Optimizer focuses on complex black-box and possibly non-scaler functions. Therefore, the Collaborative Optimizer and Froid are two complementary approaches that can co-exist in the same database system.

Some analytical engines, especially those executing polymorphic functions, use either mocked function execution or API interfaces to communicate with the functions at runtime to learn their properties [24]. These approaches are considered and evaluated by our team, yet we opt for the markup language and descriptor approach for several reasons including: (1) The API and mocked execution approaches are applicable to proprietary analytical engines, e.g., Teradata MLE in our case, but it does not easily extend to third-party external engines, e.g., Spark, Tensorflow, etc., without internal code changes, which is often not feasible. In contrast, the adopted approach is engine-independent. (2) The API and mocked execution approaches involve higher performance penalty due to the remote calls, which adds up especially under highly concurrent workloads and multiple functions per query. (3) The descriptor approach does not require any execution of the function runtime, which in the other approaches, needs to occur by either running the function code in the optimizer runtime, which is not preferred, or providing a higher-overhead service that runs the function code via a generic API e.g. REST.

In some systems, e.g., SCOPE system [33] and SAP HANA [10], the authors allow annotating the function's code to reveal some of the function's properties to the optimizer, e.g., partitioning properties and pass-through columns from inputs to outputs, which can be then used for optimizations. However, the code-annotated approach tightly couples the semantic properties to the code, which inherits the critical limitation mentioned above, i.e., this approach is only applicable to proprietary functions and does not extend to third-party engines. Moreover, the annotation approach in [10, 33], with its limited expressiveness, is not applicable to polymorphic functions. The work in [12] adopts static code analysis to discover dependencies (or conflicts) across five pre-defined operator types (*map*, *reduce*, *cross*, *match*, and *cogroup*). However, static code analysis does not extend to functions not adhering to these operator types. Moreover, its conflict graph targets operator-reordering optimization, but cannot be applied to the proposed optimizations addressed in this paper.

Finally, some systems use *plan directives* (or *optimizer hints*) for directing the optimizer towards selecting (or disabling) a specific execution plan for a given query [21, 22, 29]. Directives, which only target the standard SQL operators, are typically used by expert users either for debugging purposes or enforcing specific execution plan. Unlike directives, a function descriptor only provides facts about its function (in other words, it is part of the function), and it is up to the Collaborative Optimizer to leverage these properties as it decides. For example, a descriptor may indicate that a join on the function's output can be pushed to its inputs, but the Collaborative Optimizer may decide not to apply the push based on the cost estimates.

7 EXPERIMENTS

In this section, we provide some insight on the performance benefits that can be achieved by the Collaborative Optimizer.

Testing Environment: The cluster used in the experiments, referred to as *Vantage Cluster*, consists of 5 machines. Two machines run the SQL engine version 17.00, each has the hardware configuration of 2 CPUs, 223 GB of hard drive storage, and 20 GB main memory. Another two machines run the MLE analytics engines, each consists of 6 CPUs, 250 GB of hard drive storage, and 28 GB main memory. The 5^{th} machine, referred to as the *queen node*, is the master node in the MLE engine and it is responsible for the

execution of the analytical functions in a distributed way over the MLE machines.

Functions: We present results from six Teradata MLE analytical functions [26], namely `ExtractSentiment()`, `Sessionize()`, `GLM()`, `ForestDrive()`, `NaiveBayes()`, and `Attribution()`. They cover the different combinations of leveraging either one or both of the projection and predicate push optimizations.

Dataset Setup and Impacted Execution Phases: In general, the execution phases of an analytical query can be divided into the following phases: (1) I/O phase for reading the input tables from teradata distributed file system, (2) Export phase in which the data is transferred to the analytical function, (3) Function-Execution phase, and (4) Import phase in which the data is transferred from the analytical function back to the database system. The 2^{nd} and 4^{th} phases are only applicable in a remote setup. Depending on the database organization and whether the analytical functions are built-in or remote, we expect to see the optimization benefits shifting across these four phases.

In our experimental evaluation, the analytical functions are remote as highlighted above. Moreover, the input tables to the analytical functions are row-oriented with no specific indexing or partitioning in place. This implies that the expected benefits from the targeted optimizations will mostly reflect on the data transfer phases (the export and import phases) and the function execution phase. The I/O phase will not be impacted by the predicate and projection push optimizations since all data (rows and columns) will be read anyway from disk. Nevertheless, it is important to highlight that under different setup, the benefit's distribution across phases might take a different shape. For example, if functions are built-in within the database, the tables are columnar, and predicate-related partitioning is present, then the export and import phases will be eliminated, while most of the benefit will shift to the I/O and function-execution phases.

Descriptors Creation and Storage: As mentioned in Section 3, the descriptors are created by the Teradata system engineers who are implementing and integrating the function with the Vantage system. We currently have the descriptors created for 20+ functions from the Teradata MLE analytical engine. In Figure 9, we illustrate the size of these descriptors inside the database catalog table. Recall that the descriptors are stored in BSON format, which provides around 30% storage reduction compared to the JSON text format. In the figure, we also include the length of each descriptor, which represents the number of lines in the JSON file.

Query Processing and Effect of Triggered Optimizations: In Figures 10 and 11, we illustrate the performance gain in terms of the wall-clock query execution time from the different functions. Figure 10 presents the break up of the execution time into the different phases, which are: the *export*, *execution*, and *import* phases (the I/O phase is ignored under the current setup). Moreover, Figures 11(a)-(f) present the performance of each function under varying dataset sizes, predicate selectivity, and projection list sizes.

`Sessionize()` Function: For `Sessionize()`, the query before re-writing is as presented in Figure 1. The descriptor only enables the *predicate push* optimization on this function, where the query can be re-written as in Example 8. The input table *"web_clicks"* consists of 10 million records, where each user has on average 20 records. The results in Figure 10 are obtained under predicate selectivity of 0.0002% (which is an equality predicate on the *userId* column). The break up over the different phases show that all phases encounter big savings due to exporting less data (reduction from 238MB to 5KB), the execution of the function on very small amount of data, and consequently the function's output is also very small. In Figure 11(a), we execute the same query but under varying predicate selectivity, where the x-axis indicates the percentage of records to qualify the predicate.

`ExtractSentiment()` Function: The query before re-write is as follows (the left-hand side):

```
SELECT * FROM ExtractSentiment(
    ON product_review
    ON dictionary  AS DIMENSION
    USING
      TextColumn('feedback')
      Level('document')
      Accumulate('[1:2]') )AS dt
WHERE categoryId > 95;
```

```
SELECT * FROM ExtractSentiment(
    ON (SELECT Id, categoryId,
               feedback
        FROM product_review
        WHERE categoryId > 95)
    ON dictionary  AS DIMENSION
    USING
     TextColumn('feedback')
     Level('document')
    Accumulate('[1:2]') )AS dt;
```

Both the *predicate push* and *projection push* optimizations are applicable over `ExtractSentiment()`, which results on the re-written query in the right-hand side. The function's required columns are only the columns specified within parameters *TextColumn* and *Accumulate*. Parameter *TextColumn* specifies the column containing the text reviews to be analyzed for sentiment extraction, whereas parameter *Accumulate* specified a set of pass-through columns (referencing {Id, categoryId} by position) that function will just pass from its input to its output. The product_review table consists of 500,000 records and its schema contains 660 columns, which conforms with the schema from real customer workloads. All columns are of *numeric* data type except for the column containing the products' reviews is of type *varchar(20000)*.

The break-up results in Figure 10 are obtained under predicate selectivity of 5% and the projection of only three columns out of the 660 columns, which include the *feedback* column and the column range specified in parameter *Accumulate*. The results show a very similar behavior to that of the `Sessionize()` function. The savings are even bigger due to the combination of the predicate and projection push optimizations. In Figure 11(b), we execute the same query but under a varying range of the projected columns, i.e., expanding the column range specified in parameter *Accumulate*. As expected, the more columns to be sent to the function, the less the effect of the *projection push* optimization, but the *predicate push* optimization still yields considerable savings.

`ForestDrive()` Function: The `ForestDrive()` function takes as input a training dataset and generates a predictive model that feeds subsequent functions. This is a real-world customer query, where the training dataset consists of around 280 columns of a mix data types between numeric and string data types. From these columns, the function needs around 140 columns. Based of the function's descriptor only the *projection push* optimization is applicable. In Figure 10, we present the results under a dataset of 10 million records. The figure shows that unlike the other two functions, `ForestDrive()` is a heavy time-consuming function, which is usually the case for modeling function, and hence, the function's execution phase dominates the other three phases. As the figure shows, the Collaborative

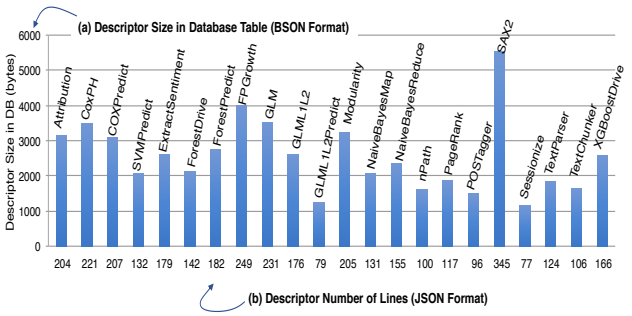


Figure 9: Descriptors Size.

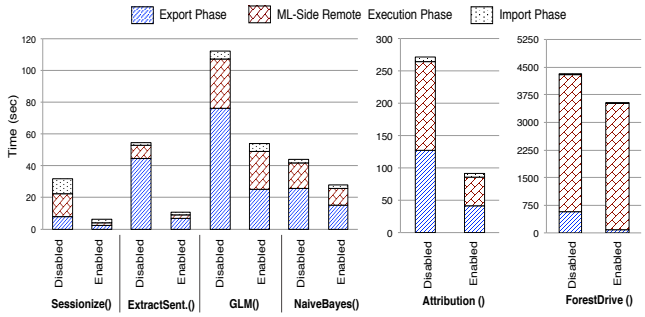


Figure 10: Execution Phases of Remote Functions.

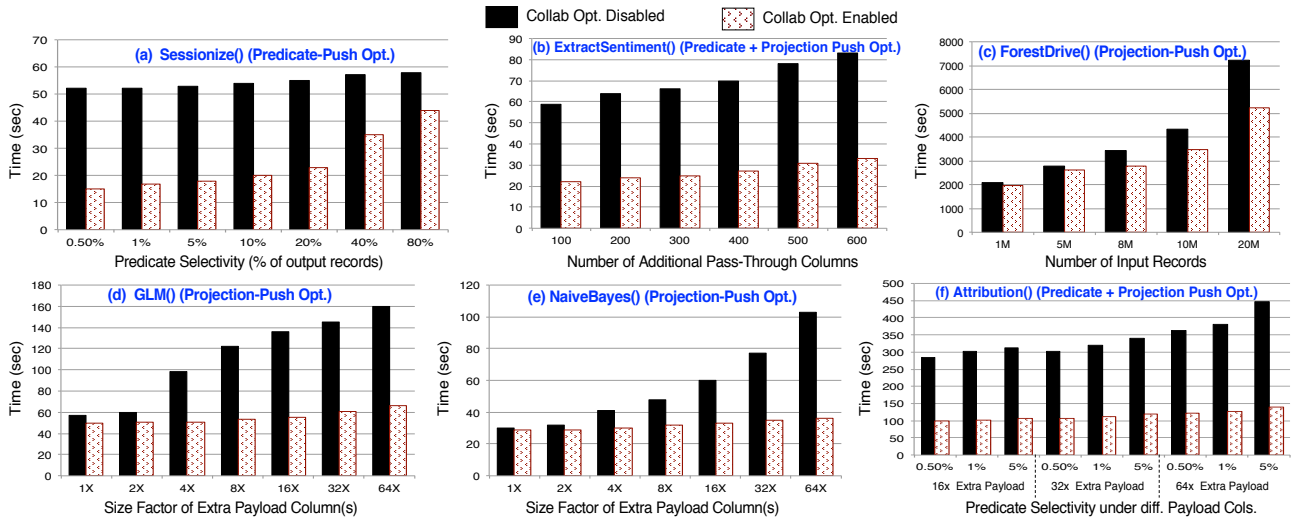


Figure 11: Evaluation of Analytical Functions under Different Parameters (dataset size, predicate selectivity, and projection sizes).

Optimizer achieves around 20% savings due to the early elimination of unneeded columns. In Figure 11(c), we execute the same query under different datasets sizes. As illustrated, the larger the input dataset, the more effective the Collaborative Optimizer in speeding up the execution. It is also worth highlighting that as the dataset gets larger, the savings are not only due to the reduction in the transferred data, but also because it reduces the chances that the analytical engine will face memory issues such as thrashing and buffer overflow.

`GLM()` & `NaiveBayes()` Functions: `GLM()` (Generalized Linear Model) and `NaiveBayes()` classifier are two additional examples where the projection push optimization is applicable. The detailed function invocation and input dataset schemas for both functions can be found in [26]. Each function receives a single input table, which consists of one million records, and each record consists of five primary columns of approximately 250 bytes altogether. These are the columns needed by the function. In Figure 11(d) and (e), we vary the number of the extra payload columns from 1X to 64X, each of size 250 bytes. As can be observed, without the Collaborative Optimizer, the system blindly pays the cost of transferring these un-needed columns.

`Attribution()` Function: This function calculates attributions with a wide range of distribution models, often used in web page analysis. The function takes 6 input tables, and the projection and predicate push specifications can be defined on each of the input tables [26]. Nevertheless, in our experiment, we focus on only applying the optimizations to the primary (large) table, which consisted of 10 million records. Figure 11(f) depicts the optimizations' savings under different predicate selectivity and projection list elimination. The zoom-in analysis for the different phases presented in Figure 10 is under predicate selectivity equals to 0.5% and 16X payload columns.

8 CONCLUSION

We presented the Teradata Collaborative Optimizer, an analytics-aware optimizer that treats the analytical functions as first-class citizens by exposing them to various types of optimizations in the SQL engine. The new optimizer is based on a novel markup language that is carefully designed to support the complex polymorphic nature of the analytical functions while still preserving the desired characteristics of simplicity, expressibility, and extensibility. Few enabled optimization types have been presented in the paper including the *projection push* and *predicate push*. Nevertheless, the Collaborative Optimizer infrastructure supports optimizations beyond these types, which will be the focus of future work.

REFERENCES

[1] 2017. SQL Data Definition Language, Release 16.20, B035-1184-162K. In *Teradata Documentation Library*.

[2] 2017. Teradata Aster Analytics Foundation User Guide Release 7.00.0. https://manualzz.com/doc/46991277/teradata-aster-analytics-/foundation-user-guide-update-2.

[3] Apache Spark. [n.d.]. https://spark.apache.org.

[4] Aster Data. [n.d.]. *http://www.asterdata.com* ([n. d.]).

[5] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. 2009. MAD Skills: New Analysis Practices for Big Data. *Proc. VLDB Endow.* 2, 2 (2009), 1481–1492.

[6] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, and et al. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. 215–226.

[7] Database PL/SQL Language Reference, Oracle Database Documentation, Release 18, Chapter 12. [n.d.].

[8] Mohamed Eltabakh, Awny AlOmari, Sanjay Nair, Mohammed Al-Kateb, Hasan Mahbub, Anantha Subramanian, Robert Wehrmeister, and Kashif Siddiqui. 2018. Enabling Cross-Platform Query Optimization via Expressive Markup Language, 12/10/18, No. 62/777,304. US Provisional Patent.

[9] Eric Friedman, Peter Pawlowski, and John Cieslewicz. 2009. SQL/MapReduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proc. VLDB Endow.* 2, 2 (2009), 1402–1413.

[10] Philipp Große, Norman May, and Wolfgang Lehner. 2014. A study of partitioning and parallel UDF execution with the SAP HANA database. *ACM International Conference Proceeding Series* (06 2014). https://doi.org/10.1145/2618243.2618274

[11] Joseph M. Hellerstein and et. al. 2012. The MADlib Analytics Library: Or MAD Skills, the SQL. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 1700–1711.

[12] Fabian Hueske, Mathias Peters, Matthias J. Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. 2012. Opening the Black Boxes in Data Flow Optimization. *Proc. VLDB Endow.* 5, 11 (2012), 1256?1267.

[13] Introduction to Greenplum In-Database Analytics. [n.d.]. https://greenplum.org/gpdb-sandbox-tutorials/introduction-greenplum-database-analytics/.

[14] Jeff LeFevre, Jagan Sankaranarayanan, Hakan Hacigumus, Junichi Tatemura, Neoklis Polyzotis, and Michael J. Carey. 2014. Opportunistic Physical Design for Big Data Analytics. In *ACM SIGMOD*. 851?862.

[15] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. 2015. Efficient Processing of Window Functions in Analytical SQL Queries. *Proc. VLDB Endow.* 8, 10 (June 2015), 1058–1069. https://doi.org/10.14778/2794367.2794375

[16] Kun Li, Daisy Wang, Alin Dobra, and Christopher Dudley. 2015. UDA-GIST: An In-database Framework to Unify Data-Parallel and State-Parallel Analytics. *VLDB* (2015), 557–568.

[17] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. 2017. MLog: Towards Declarative In-database Machine Learning. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1933–1936. https://doi.org/10.14778/3137765.3137812

[18] ShirishTatikonda MatthiasBoehm, PrithvirajSen BertholdReinwald, DouglasR YuanyuanTian, and ShivakumarVaithyanathan Burdick. 2014. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *Proceedings of the VLDB Endowment* 7, 7 (2014).

[19] Jan Michels, Keith Hare, Krishna Kulkarni, Calisto Zuzarte, Zhen Hua Liu, Beda Hammerschmidt, and Fred Zemke. 2018. The New and Improved SQL: 2016 Standard. *SIGMOD Rec.* 47, 2 (Dec. 2018), 51?60.

[20] Microsoft Analytics Platform System. [n.d.]. www.microsoft.com/en-us/server-cloud/products/analytics-platform-system.

[21] MySQL 5.7 Reference Manual. [n.d.]. https://dev.mysql.com/doc/refman/5.7/en/optimizer-hints.html.

[22] Benjamin Nevarez. 2010. Inside the SQL Server Query Optimizer. In *Simple Talk Publishing*.

[23] Oracle Analytics Cloud. [n.d.]. https://www.oracle.com/solutions/business-analytics/analytics-cloud.html.

[24] A. Pandit, D. Kondo, D. Simmen, A. Norwood, and T. Bai. 2015. Accelerating Big Data analytics with Collaborative Planning in Teradata Aster 6. In *2015 IEEE 31st International Conference on Data Engineering*. 1304–1315.

[25] Linnea Passing, Manuel Then, Nina Hubig, Harald Lang, Michael Schreier, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. 2017. SQL- and Operator-centric Data Analytics in Relational Main-Memory Databases. In *EDBT*.

[26] Srl Raghavan. 2014. Teradata Aster Discovery Portfolio. In *Teradata Documentation Library*.

[27] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of Imperative Programs in a Relational Database. *Proc. VLDB Endow.* 11, 4 (Dec. 2017), 432?444. https://doi.org/10.1145/3186728.3164140

[28] SAP IQ Administration: User-Defined Functions 16.1 SP04 PL02-PL06, Polymorphic Table Functions (PTFs). [n.d.].

[29] SQL Plan Directives in Oracle Database 12c Release 1 (12.1). [n.d.]. https://oracle-base.com/articles/12c/sql-plan-directives-12cr1.

[30] Xin Tang, Robert M. Wehrmeister, James Shau, and et al. 2016. SQL-SA for big data discovery polymorphic and parallelizable SQL user-defined scalar and aggregate infrastructure in Teradata Aster 6.20. In *32nd IEEE International Conference on Data Engineering, ICDE 2016*. 1182–1193.

[31] TensorFlow. [n.d.]. https://www.tensorflow.org.

[32] The Apache Software Foundation. [n.d.]. Hadoop. http://hadoop.apache.org.

[33] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Åke Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: parallel databases meet MapReduce. *The VLDB Journal* 21 (2012), 611–636.