TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik - Institut für Systemarchitektur

TECHNISCHE BERICHTE
TECHNICAL REPORTS
ISSN 1430-211X

TUD-FI14-03 Mai 2014

Philipp Große; Norman May; Wolfgang Lehner
Institut für Systemarchitektur, TU Dresden

A Study of Partitioning and Parallel UDF Execution with the SAP HANA Database

Technische Universität Dresden
Institut für Systemarchitektur
Professur für Datenbanken
Fakultät Informatik, 01062 Dresden, Germany
http://wwwdb.inf.tu-dresden.de/

A Study of Partitioning and Parallel UDF Execution with the SAP HANA Database

Philipp Große
Technische Universität Dresden
Database Technology Group
Dresden, Germany
philipp.grosse@sap.com

Norman May
SAP AG
Dietmer-Hopp-Allee 16
Walldorf, Germany
norman.may@sap.com

Wolfgang Lehner
Technische Universität Dresden
Database Technology Group
Dresden, Germany
wolfgang.lehner@tu-dresden.de

ABSTRACT

Large-scale data analysis relies on custom code both for preparing the data for analysis as well as for the core analysis algorithms. The map-reduce framework offers a simple model to parallelize custom code, but it does not integrate well with relational databases. Likewise, the literature on optimizing queries in relational databases has largely ignored user-defined functions (UDFs). In this paper, we discuss annotations for user-defined functions that facilitate optimizations that both consider relational operators and UDFs. We believe this to be the superior approach compared to just linking map-reduce evaluation to a relational database because it enables a broader range of optimizations. In this paper we focus on optimizations that enable the parallel execution of relational operators and UDFs for a number of typical patterns. A study on real-world data investigates the opportunities for parallelization of complex data flows containing both relational operators and UDFs.

1. INTRODUCTION

A lot of valuable information is stored in relational databases today. However, analyzing these large data sets is often limited by the expressiveness of SQL. For example data collected for radio astronomy may require data preprocessing including data cleansing and iterative algorithms for deriving good parameters or for analyzing the data. Business applications may analyze customer data for segmentation and classification of customers to derive targeted product offers. In our case study, we want to predict the relevant test cases given a code change which also requires the preprocessing steps of the test data, non-trivial data analysis and iterative computation of connected components. In all of the aforementioned cases there is a lot data with a rigid schema and relational databases are a good candidate to store this kind of data.

However, the examples also indicate that it is often necessary to implement core parts of the analysis with user-defined functions (UDFs), e.g. the data preparation or iterative algorithms. As optimizers of databases today have limited capabilities to optimize complex queries using UDFs, they are often only used as storage containers, but not considered for the execution of these complex analytical tasks.

In the recent years, the map-reduce (MR) framework has become popular for large-scale data analysis because it offers a simple model to implement custom code [9]. The MR framework also promises good scalability because the MR runtime handles the parallel execution of independent tasks. However, MR does not integrate well with relational databases where a significant amount of relevant data is stored. Database vendors attempt to remedy this situation by implementing adapters to MR, but this limits the ability for optimizations across relational and custom logic.

These observations lead us to the following requirements for large-scale data analysis. First, it is desirable to use a declarative language as much as possible. The expected benefits of declarative languages are better productivity and more opportunities to optimize the resulting code. Second, the (family of) languages used to implement large-scale analysis tasks must be expressive enough. For example, iteration and state maintenance are typically required for analysis tasks. Third, the performance of the analysis tasks must be good. This means that it must be possible to optimize the code including the custom logic expressed in UDFs even if treating the UDF code itself as blackbox. Considering the size of the data, the optimizations must consider a scale out by parallelizing code and exploit parallelization even across multiple nodes in the database.

Today, large scale data analysis is mainly approached from two directions. On the one side, SQL is used in data warehouse applications for the repeated analysis of well-structured data. In this domain, developers benefit from the high-level declarative query language, which is easy to optimize and makes them more productive. However, ad-hoc modes for analysis using complex algorithms are limited by the expressiveness of SQL queries. On the other side, map-reduce offers a lot of freedom to implement specialized algorithms [9]. This comes at the cost of manual implementation, testing and tuning. Also, the effectiveness of map-reduce was often questioned, e.g. because the basic map-reduce framework does not consider schema information and because intermediate results are stored on disk. As a middle ground, data-oriented workflow engines seem to evolve, and we see our work mostly

.

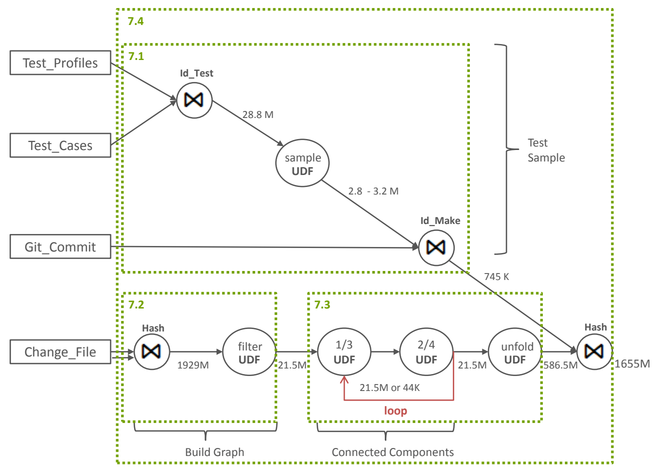


Figure 1: Data preprocessing with UDFs and Joins

related to this stream of work, e.g. [4, 3, 10].

We believe that the workflow-oriented approach is the most promising one because it brings the power of parallelizable data-oriented workflows to database technology. The contributions in this paper are summarized as follows:

- We introduce a set of UDF annotations describing UDF behavior.
- We show how parallel UDF execution can be combined with relational database operations.
- We discuss plan rewrites and a rewrite strategy to transform an initial sequential plan to a parallelized one.
- We show that plan optimization for parallel execution can also be combined with iterative loops.

In the following Section 2 we introduce the preprocessing of an application that predicts the relevant tests for code changes in a large development project based on past test failures. We will use this example later to demonstrate the effectiveness of our optimization and execution strategy. After that we survey work related to ours (Sec. 3). Section 4 introduces the UDF annotations; we base our plan optimizations upon. In Section 5 we present our translation process from workflow definition to basic execution plan. In Section 6 we present the rewrites and the rewrite strategy and discuss in Section 7 the effectiveness of this strategy based on the use case introduced in Section 2.

2. EXAMPLE

To motivate our work and to illustrate the benefit of an integrated optimization and execution of relational operators and user-defined functions we introduce the following use case. In the test environment of our system we faced exceedingly long turn-around times for testing after changes were pushed to our Git repository via Gerrit[1]. Similar to the ideas of JUnitMax[2], we only wanted to execute the tests that are affected by a code change starting with the most relevant ones.

[1]see `http://code.google.com/p/gerrit/`
[2]see `http://junitmax.com`

Table	Description	Cardinality
`Test_Profiles`	the test configurations, primary key `Id_Test`, `Id_Make` for the test execution	4.8M
`Test_Cases`	actual regression tests, primary key `ID`, foreign key `Id_TEST` references `Test_Profiles`, test result in `Status`	28.8M
`Git_Commit`	one submitted change, primary key `Hash`, the submission time `Commit_Date`, make id `Id_Make`	256K
`Changed_File`	the modified source files `Id_File` and the corresponding Git hash `Hash`	2M

Table 1: Tables of example use case

To tackle this problem we train a classification model based on the history of test executions stored in the test database. The classification model shall assign each regression test a probability of failure given the changed files identified by their file-ID. This allows us to define an ordering of the regression tests starting with the regression test with the highest probability of failure and also to exclude tests from the regression test run.

Figure 1 shows a simplified version of the data preprocessing done for the classifier training, and the schema of our test database is shown in table 1. The model combines relational processing and user-defined functions in a single DAG-structured query execution plan. Such plans are quite common in scientific workflows and large-scale data analysis. But as discussed in the introduction, current systems either shine in processing relational operators or UDFs, but rarely in both. The process of creating the training data is illustrated in Figure 1 and can roughly be separated into two parts.

The first part collects information on the test case behavior joining the tables `Test_Profiles` and `Test_Cases` and the first UDF function `sample`. The `sample` UDF creates a sample[3] of the regression tests at different rates depending on the outcome of the test. Because successful test executions are much more common than failures, we handle the data skew on the outcome of tests by down-sampling successful test cases and keeping all failing tests.

The second part of the process is more complex. Instead of looking at the impact of a single source file modification from the table `Changed_File` for our classifier, we group files that are commonly changed together. We conjecture that a strong relationship between these groups of files and sets of relevant tests exists. For example, file groups may relate to a common system component in the source code. To identify those file groups we are looking for files that were often modified together in one Git commit. The second part of the process therefore starts with the self join of the base table `Changed_File` followed by the `filter` UDF. The UDF groups the join result by pairs of `Id_File` of common Git commits and aggregates the `Hash` values of the commits into a list of `Hashes`. After that pairs of files are discarded that are below a given threshold of co-occurrence in a commit. It may seem that these two operations could be performed by regular relational operators, but the text-based "folding" of

[3]similar in function to the sample clause from the SQL Standard 2003 [16]

strings into a CLOB (or set-valued attribute) is not defined in the SQL standard. Inspired by the built-in aggregation function GROUP_CONCAT provided by MySQL we realize this grouping with a user-defined aggregate. The final UDF `unfold` in this part of the process unfolds the CLOB column again for further processing. After the second UDF we have identified common file pairs. But to identify all groups of connected files further UDFs are required. We implemented the computation of the connected components with an iterative algorithm [15] using two UDFs `UDF1` and `UDF2` with a data-driven break condition.

The details of all involved UDFs are discussed in Section 7. For now it is only important to note that the input of each UDF can be partitioned and the UDFs can therefore be executed in parallel—similar to relational operators. It is our goal to extend optimizers to exploit these opportunities for better performance. Unfortunately, it is difficult to analyze the UDF code and detect if it can and should be parallelized. To solve this problem, we propose annotations in Section 4 to declare opportunities for parallelization but also to annotate how the output cardinality of the UDF relates to its input.

3. RELATED WORK

User-Defined Functions in Relational Databases.

If complex algorithms are implemented inside a relational database, one usually needs to rely on user-defined functions (UDFs) to implement non-standard operations like data cleansing, specific sampling algorithms or machine-learning algorithms. This leads to the requirement to efficiently support UDFs inside a database, i.e., considering UDFs during query optimization as well as efficient parallel execution. Hellerstein and Stonebraker [14] were the first to examine the placement of expensive predicates and scalar UDFs within an execution plan. Neumann et al. [19] survey related work and prove that ordering predicates and map operators (i.e. functions) is an NP-hard problem and provide an optimal algorithm for that optimization problem.

Jaedicke and Mitschang [17] focus on the optimization and parallel execution of user-defined aggregate and table functions. While deterministic scalar UDFs are trivial to parallelize, user-defined aggregation functions are aggregated locally and later combined or reduced to the final result of the aggregate. The work of Jaedicke et al. [18] requires a static partition strategy be defined for a user-defined table function so that the database can parallelize it. In a more recent work Friedman et al. [10] identify opportunities for parallelization of user-defined table functions, which are derived dynamically from the context during query optimization.

In this work we address all classes of UDFs; the distinction between scalar, aggregate, and table UDFs is derived from its annotation. For example, UDFs whose input can be partitioned arbitrarily are treated as scalar UDFs if they only attach additional data to a row. We realize the parallelization of UDFs using the Worker Farm pattern (see [12] and Section 5 for details). This allows us to combine worker farm patterns and to exploit opportunities to parallelize the execution of the UDFs.

By default, most commercial databases do not allow for parallelization of user-defined table functions. In Oracle user-defined table functions can be annotated so that their input can be partitioned, and they can be executed in parallel. However, their optimization strategy does not seem to be published. Overall, the ability to parallelize user-defined table functions in relational databases is very limited today, which also limits the scalability of custom code. Some database vendors like Oracle or SAP [22, 2] offer an interface to standard map-reduce frameworks to benefit from the parallel execution capabilities of map-reduce, but those interfaces do not allow for optimizations across relational and custom logic.

Workflow Engines.

Pavlo et al. [20] were among the first to analyze the performance deficiencies of map-reduce compared to parallel databases for some basic analysis tasks. These limitations led to the development of workflow engines targeted at large-scale data analytics [7, 4, 23]. These engines allow for more flexible compositions of user-defined code than map-reduce while keeping the ability to parallelize tasks. Probably most closely related to our work is SCOPE [7, 24]. That work focuses on the parallelization of relational operators based on a property framework. In this paper we extend these ideas to support both relational operators and UDFs. A similar approach is PACT [4] where contracts are bound to tasks as pre- and post-conditions. These contracts enable rewrites of the workflow. This style of annotation is similar to our Worker Farm skeleton with split() and merge() methods. However, our approach explores a different design, by focusing on the integrated optimization of existing database operators and UDFs.

Like standard algebraic optimization for SQL, rewrites applied on workflows crucially depend on the availability of cardinality information. In that area we are only aware of Agarwal et al. [3] where a repository of statistics for workflows is maintained, and re-occurring plan patterns are detected and used for cardinality information. In our proposal, we use annotations on the UDF to derive cardinality information.

Advanced Analytics Applications.

Among the most prominent applications of map-reduce are machine learning algorithms because they are not supported well in databases today but also because they are performance-critical [11, 8]. Apache Mahout [1] is a collection of basic machine learning algorithms implemented on Apache Hadoop, the map-reduce implementation of the Apache project. As many scientific optimization problems, machine learning algorithms and data preprocessing tasks rely on iteration. HaLoop [6] proposed to integrate iteration natively into the map-reduce framework rather than managing iterations on the application level. This includes the integration of certain optimizations into Hadoop, e.g. caching loop invariants instead of producing them multiple times. In our framework iteration handling is explicitly applied during the optimization phase, rather than implicitly hidden in the execution model (by caching).

Another important application area that is not well-supported by databases today are advanced statistical applications. While basic aggregates like average, variance, or standard deviation are usually provided by databases, more advanced statistical functions are not supported well. Many algorithms are available in statistical packages like R or SPSS. Furthermore, domain experts feel more comfortable implementing their algorithms using R. Finally, the basic data for these statistics are stored in relational databases. Consequently, it is desirable to support statistical algorithms

```
CREATE TYPE D1(testID Integer, makeID Integer, status CHAR(10));

CREATE PROCEDURE sample(IN data D1, OUT sample D1)
READS SQL DATA LANGUAGE PSEUDOC AS
BEGIN PARALLEL
PARTITION(data(MINPART(NONE), MAXPART(ANY)))
EXPECTED(data(GROUPING(NONE), SORTING(NONE)))
BEGIN
    outRow = 1;
    forall inRow in 1:size(data):
      if(data[inRow].status == "OK" and math::random() > 0.1):
        continue;
      sample[outRow].testID = data[inRow].testID;
      sample[outRow].makeID = data[inRow].makeID;
      sample[outRow].status = data[inRow].status;
      outRow++;
END
ENSURE KEY(sample = data), PRESERVE ORDER(sample = data),
SIZE(sample = 0.05 * data + 0.1 * 0.95 * data),
RUNTIMEAPPROX(1 * data), DETERM(0)
END PARALLEL UNION ALL;
```

Script 1: SQLScript pseudo code of the sample UDF with partition ANY

implemented, e.g. in the R language, as a special kind of UDFs [13]. Similar to our preceding discussions, the integration of external statistical packages was mainly treated as a black box so far, and consequently, the execution of these workflows missed the potential for better performance. With the contributions of this paper we want to remedy this situation.

4. UDF ANNOTATIONS

SQLScript [5] procedures are a dialect for stored procedures in SAP HANA to define dataflows and also to include custom coding to be executed in SAP HANA. Several implementation languages are supported including R (RLANG) [13]. In this paper we abstract from the implementation language and use pseudo C (PSEUDOC) instead. In this section we discuss a number of annotations for SQLScript procedures to provide metadata that help the query optimizer to derive better query execution plans, in particular to increase the degree of parallelism for dataflows with UDFs. These optimizations are available independent from the implementation language chosen.

The annotations help us to distinguish between scalar, aggregate, and table UDFs based on their partitioning and grouping requirements. The classical Map and Reduce operations are only two possible instances we can describe with the given set of annotations.

Table 2 gives a short overview of all possible annotations. The annotations can be classified into three groups. The first part describes the partitioning pre-conditions expected by the UDF, the second group contains the post-conditions ensured by the UDF, and the third group contains optimizer hints. The keywords BEGIN and END enclose the UDF code, and BEGIN PARALLEL and END PARALLEL mark the beginning and end of the annotations. All of those annotations are purely optional, although without partitioning information the UDF will only be invoked once, and thus no data-level parallelism will be exploited. The example in Script 1 for the `sample` UDF introduced in Section 2 shows the complete set of possible annotations, even those that can be implied by others. Since UDFs support multiple inputs and outputs, annotations may apply only to specific input or output tables. This is realized by stating the name of the parameter and enclosing the properties in parentheses.

4.1 Partitioning Pre-conditions

The first annotation PARTITION precedes the code block of the procedure. We describe the granularity of partitioning supported by the UDF by defining MINPART and MAXPART for each input set. MINPART defines the lower bound of required partitioning, whereas MAXPART defines the upper bound of the highest possible partitioning. We distinguish between those two so that the optimizer can choose the granularity depending on the surrounding operators and their required partitioning. By default MINPART is set to NONE, which means that the UDF does not need partitioning, whereas the default for MAXPART is ANY, which means that the UDF can handle any partitioning. This default setting—like in Script 1—refers to a UDF table function implementing a scalar operation, which can be executed on each tuple of the relation independently. Since the UDF table function code can cope with multiple tuples at a time and could even consume the whole non-partitioned table, the optimizer can decide freely how to partition the input to the UDF and how to parallelize it.

Many UDFs (such as user-defined aggregates) operate on groups of tuples. A partitioning schema can be described by defining grouping columns for MINPART and MAXPART: The MINPART annotation over a set of columns with n distinct values enforces at least n distinct UDF instances for processing, but it does not ensure that the data is not further partitioned. The MAXPART annotation over a set of grouping columns with n distinct values ensures that the input relation is not partitioned into more than those n distinct parts. It effectively describes a group-by partitioning, but it does not guarantee that a UDF instance consumes only one distinct group. The annotation of MAXPART with the keyword NONE tells the optimizer that the UDF code does not allow partitioning and will only work if the UDF consumes the whole table in a non-partitioned way. Setting MINPART and MAXPART to the same set of columns ensures that each distinct instance of the grouping columns is associated to exactly one distinct UDF instance, which would be equivalent to a reduce function in map-reduce. However setting MINPART to NONE or a subset of MAXPART can help the optimizer to make better decisions by picking the optimal degree of partitioning.

Additionally to the global partitioning the annotation EXPECTED() followed by a list of SORTING and GROUPING actions and their respective columns describes local grouping and sorting of tuples within each partitioned table part. In the example of Script 1 the information is redundant with the annotation MAXPART(ANY) and could be removed.

4.2 UDF Behavior and Post-conditions

As we treat user-defined code as a black box, the behavior and possible side effects of the code is—in contrast to the well-defined relational operations—not known to the database management system. Without further information it is difficult for the optimizer to distinguish between user-defined aggregations, user-defined table functions or some other user-defined logic. It also cannot exploit any characteristics of the UDF that may allow optimizations to be applied to the dataflow. Hence, we allow for adding a set of post-conditions after the code block of the UDF.

The annotation KEY makes a statement about the behavior of the UDF regarding columns used as partitioning columns in the MAXPART annotation. To support a wide

Class	Annotation	Description
PARTITION (Pre-conditions)		Expected global partition properties: a set of input tables each attached to a MINPART and a MAXPART annotation.
	MINPART	Required logical partitioning: NONE (*Default*) as no partitioning required; ANY as expected logical partitioning down to single tuples; A set of column names $<C_1,...,C_n>$ required as GROUP BY partitioning.
	MAXPART	Highest possible supported logical partitioning: NONE as no partitioning supported. Pass table as copy; ANY (*Default*) as support of arbitrary partitioning (e.g. round robin); A set of column names $<C_1,...,C_n>$ supported as GROUP BY partitioning.
EXPECTED (Pre-conditions)		Expected partition local properties: A set of input tables each attached to a list of actions $\{\hat{A}_1, \hat{A}_2, ..., \hat{A}_n\}$; $\hat{A}$ is either a SORTING or a GROUPING annotation
	GROUPING	NONE (*Default*) as no partition local grouping required A set of grouping columns $\{G_1, G_2, ..., G_n\}^g$
	SORTING	NONE (*Default*) as no partition local sorting required A list of sorting columns $\{S_1^o, S_2^o, ..., S_n^o\}$ each given with an order $o \in$ {ASC, DESC}.
ENSURE (Post-conditions)		Ensured partition local properties after UDF: A set of output relations each attached to a list of actions $\{\hat{A}_1, \hat{A}_2, ..., \hat{A}_n\}$; $\hat{A}$ is either a SORTING or a GROUPING annotation
	GROUPING	NONE as no partition local grouping guaranteed by UDF A set of grouping columns $\{G_1, G_2, ..., G_n\}^g$
	SORTING	NONE as no partition local sorting guarantied by UDF A list of sorting columns $\{S_1^o, S_2^o, ..., S_n^o\}$ each given with an order $o \in$ {ASC, DESC}.
	KEY	UDF behavior regarding grouping columns = (*Default*) the grouping columns do only contain values which have been input as part of the local partition != the grouping columns may contain values which have not been input as part of the local partition (existing partitioning cannot be reused) $->$ describes functional dependencies for new columns derived from previous grouping columns
PRESERVE (Post-conditions)	ORDER (*Default*)	As alternative to ENSURE SORTING or GROUPING this annotation denotes FIFO logic preserving existing SORTING or GROUPING of the input table
- (Optimizer Hints)	DETERM	1 (*Default*): UDF has deterministic behavior. 0: UDF has no deterministic behavior.
- (Optimizer Hints)	SIZE	= (*Default*): expected size of the output relation is equal to the size of the input relation factor: expected size of the output relation can be derived by the size of the input relation multiplied by a factor
- (Optimizer Hints)	RUNTIME-APPROX	= (*Default*): expected run time of the UDF is determined by the size of the input relation factor: expected run time of the UDF is determined by the size of the input relation and a factor
END PARALLEL (Post-conditions)	UNION ALL (*Default*)	By default we assume an order-preserving concat merge combining the partitions
END PARALLEL (Post-conditions)	AGG	Instead of an order-preserving concat merge the UDF can also be followed by any kind of aggregation function known to the database system. In this case an additional repartitioning between UDF and aggregation maybe required.

Table 2: Overview of the annotations

range of possible UDFs those columns are not hidden from the UDF and can be manipulated during processing just like every other column. In order to combine UDF parallelism with relational operators it is often assumed that those grouping columns are not modified by the UDF. This behavior is annotated as KEY(=). Although the UDF may introduce new tuples or remove existing tuples the annotation KEY(=) states that the grouping columns contain no new values compared to the input table visible for each respective UDF instance. In contrast KEY(!=) annotates a modification of grouping columns, which effectively means that existing partitioning on those columns cannot be reused for further processing, since the values and thereby possibly the semantic have been changed. In a similar way KEY($->$) describes functional dependencies for new columns derived from previous grouping columns and indicates that an existing grouping is reusable even though original grouping columns are missing in the output schema.

If the UDF itself ensures sorting or grouping independent of the given input, it can be annotated by the keyword ENSURE followed by a list of SORTING and GROUPING actions and their respective columns, similar to the EXPECTED annotation from the pre-conditions. Alternatively the annotation PRESERVE ORDER—as in Script 1—states that the UDF implements a first-in-first-out logic preserving the order of tuples derived from the input.

Analog to BEGIN PARALLEL PARTITION annotation describing the expected partitioning the END PARALLEL annotation describes the combining of the parallel processed UDF results. By default we assume an order-preserving concat merge (UNION ALL) to be used. Alternatively the UDF may also be followed by any kind of aggregation function known to the database system. In this case an additional repartitioning between UDF and aggregation maybe required.

4.3 Optimizer Hints

In addition to the pre- and post-conditions describing data manipulation we introduce a number of annotations that describe runtime characteristics of a UDF. This may provide further hints to the optimizer to derive better execution plans. The DETERM annotation tells the optimizer whether the UDF is deterministic or not. By default we assume the UDF to behave deterministically. However, in our example (see Script 1) the UDF has non-deterministic behavior due to the random function used for sampling. The RUNTIME-APPROX annotation tells the optimizer something about the expected runtime for the UDF relative to the input respectively output size. In our example RUNTIMEAPPROX(1 $*$ data) states that the runtime of the UDF is linear to the input size. The SIZE annotation tells the optimizer the expected data size of the output relation compared to the input relation. In our example (see Script 1) we know for sure that the size will not increase and even expect the size to decrease depending on the column `STATUS` of the input data.

Further annotations are possible to describe UDF properties, e.g. commutativity, associativity or decomposability of aggregate functions. They enable optimizations beyond those discussed here, e.g. to be able to reorder UDFs [21].

5. PROCESSING WORKFLOWS

In this section we describe how we translate a workflow, which may include both UDFs and relational operators. Some of these features are only available as a prototype of SAP HANA and not yet part of the productive code. As a foundation for our experiments, we also briefly survey the properties used during optimization.

5.1 Structural Properties

Structural properties were introduced in SCOPE [24] for reasoning about partitioning, grouping and sorting in a uniform framework. Partitioning applies to the whole relation; it is a *global structural property*. Grouping and sorting properties define how the data within each partition is organized and are therefore *local structural properties*. As all of those properties (and the derived optimizations) do not only apply in the context of relational operations, but are also important in the context of parallel execution of UDFs, we decided to adapt this formal description and apply it along with our UDFs annotations. Thus we briefly summarize the formalism introduced in SCOPE [24]:

Definition (Structural Properties) The structural properties of a table $\mathcal{R}$ can be represented by partitioning information and an ordered sequence of actions:

$$\{\mathcal{P}; \{\hat{A}_1, \hat{A}_2, ..., \hat{A}_n\}\}$$

where the first part defines its global structural property, and the second sequence defines its local structural property. Global structural properties imply the partitioning function used. The annotations introduced in Section 4 assume a *non-ordered partitioning* (e.g. hash partitioning) for a given column set or a non-deterministic partitioning (e.g. round-robin or random partitioning) in the case of an empty column set. Local structural properties are represented by an ordered sequence of actions $\hat{A}_i$. Each action is either grouping on a set of columns $\{C_i, ..., C_j\}^g$, or sorting on a single column C^o.

Partition 1	Partition 2	Partition 3
$\{1,4,2\}$, $\{1,4,5\}$, $\{7,1,2\}$	$\{4,1,5\}$, $\{3,7,8\}$, $\{3,7,9\}$	$\{6,2,1\}$, $\{6,2,9\}$

Table 3: Partitioned relation with grouping and sorting

Definition (Non-ordered Partitioning) A table $\mathcal{R}$ is *non-ordered partitioned* on a set of columns $\mathcal{X}$, if it satisfies the condition

$$\forall r_1, r_2 \in \mathcal{R} : r_1[\mathcal{X}] = r_2[\mathcal{X}] \Rightarrow P(r_1) = P(r_2)$$

where r_1,r_2 denote tuples and P the partitioning function used.

Table 3 shows an instance of a table with three columns C_1, C_2, C_3 and structural properties $\{\{C_1\}^g; \{\{C_1, C_2\}^g, C_3^o\}\}$. These properties mean that the table is partitioned on column C_1 and, within each partition, data is first grouped on columns C_1, C_2, and, within each such group, sorted by column C_3.

The required structural properties of relational operations and the optimizations combining relational operations are extensively discussed in SCOPE [24]. The annotations we introduce in Section 4 describe the required structural properties of the input tables consumed by the UDF as well as the structural properties of the returned output tables. This allows combining the optimization of UDFs and relational operations.

5.2 Modeling Workflows in SAP HANA

The most convenient way to express complex dataflows in SAP HANA is using SQLScript [5]. In this paper we use syntax that is similar to SQLScript for our UDFs. Relational expressions in SQLScript are directly translated into the internal representation of our cost-based optimizer. Non-relational operations, including UDFs, are mapped to custom operators, which can be compiled into machine code using a just-in-time compiler. Developers of custom logic need to annotate UDFs with the metadata including the ones introduced in Section 4.

Query processing of workflows in SAP HANA is managed by the calculation engine (see [13] for details). Nodes represent data processing operations, and edges between these nodes represent the dataflow. Data processing operations can be conventional database operations, e.g. projection, aggregation, union or join, but they can also be custom operators processing a UDF. Intermediate results are commonly represented as in-memory columnar tables, and hence, they carry schema information and other metadata, which can be exploited during optimization.

At the modeling level the dataflow can be parallelized if operators are not in a producer-consumer relationship. At execution time operators are scheduled for processing, once all their inputs are computed.

5.3 Introducing Parallelism

Starting with the basic translation of the workflow into a canonical execution plan, we attempt to increase the opportunities to parallelize the execution of the dataflow graph.

To make sure that the requested partitioning described trough the global structural properties is generated, we support the same set of operators as SCOPE [24], i.e. Initial Partitioning, Repartitioning, Full Merge, Partial Reparti-

Notation	Description
OP	Any operation (or sequence of operations) – relational operator or UDF
UDF	UDF operation
ROP	Relational operator
HJ	Hash join operation
BJ	Broadcast join operation
$<$	Initial Partitioning operation
$>$	Full Merge operation
$\times$	Repartitioning operation
$>=$	Partial Merge operation
$=<$	Partial Repartitioning operation
$-$	Keep partitioning; if not partitioned - no-op
$\mathcal{P}$	Partitioning
*	Any properties (including empty)

Table 4: Notation for operators and properties

tioning, and Partial Merge. We support some additional merge-operations including Union All and Union Distinct. Similar to Jaedicke et al. [17] any kind of global aggregation function or custom merge can be used to combine partial aggregates or results of user-defined table functions. For the relational operators we apply the property derivation and matching as presented in SCOPE [24].

In this paper, we extend this work to also apply to UDFs. Given a non-partitioned input, the generic model to parallelize UDFs is to apply the Worker-Farm pattern. This pattern first applies an initial partitioning operation so that the UDF can be executed in parallel in the so-called work-method. Finally, a merge operation integrates the results of the parallel computations. In our previous work [12] we discuss further patterns which can be parallelized using the Worker-Farm pattern.

Obviously the enforcement of partitioning and a full merge for each UDF operation independently is a very simple approach. It is therefore the goal of our optimizer to break this isolated view and to take surrounding operations and their structural properties into account.

In Section 6 we discuss how we can avoid full merge and succeeding full partitioning between relational operators or UDFs. As we will analyze in Section 7, exploiting partitioning properties across these operations improves the robustness and scalability of the workflow execution.

6. OPTIMIZATION STRATEGY

As we treat UDFs as table functions it is often possible to define the UDF code in a way, that it can work both as single instance consuming a non-partitioned table as well as with multiple instances on a partitioned table. To reflect this we describe the required structural properties of our UDFs with a partitioning $\mathcal{P}$ that is in the range of a minimal partitioning column set $\mathcal{P}_{min}$ and a maximum partitioning column set $\mathcal{P}_{max}$: $\emptyset \subseteq \mathcal{P}_{min} \subseteq \mathcal{P} \subseteq \mathcal{P}_{max} \subset \top$ where $\emptyset$ indicates that the input table is not partitioned ($\perp$) and is passed as a copy to the UDF. On the other extreme and even more general than a partitioning on all available columns $\top$ indicates an arbitrary partitioning.

If the UDF consumes multiple input tables (e.g. table A and B) we assume all input tables to be partitioned in the same way ($\mathcal{P}_A = \mathcal{P}_B$). An exception is the case where the UDF allows partitioning on one input table ($\mathcal{P}_{maxA} \neq \emptyset$) but not for all other input tables ($\mathcal{P}_{maxB} = \emptyset$). In this case the non-partitioned table B is broadcast as copy to each UDF instance derived from the partitioning of the table A.

6.1 Rewrites

Based on the canonical parallelization of UDFs and relational operators described in Section 5 and the properties derived for them, we now discuss the basic plan rewrites in the case of a mismatch of structural properties. Each rewrite considers two consecutive operations OP$_1$ and OP$_2$. Where $\{\mathcal{P}_1;*\}$ describes the structural properties of the output of OP$_1$ and $\{\mathcal{P}_2;*\}$ the expected structural properties for the input of OP$_2$ and $*$ denotes any local structural property. In case of a mismatch of *local structural properties* the framework may further enforce explicit sorting or grouping operations (see e.g. [24]) not discussed in this report. Table 4 summarizes the notation we use in this paper. The most basic exchange operation to fix mismatching *global structural properties* is a full merge operation followed by a initial partitioning operation.

$$\begin{aligned}
\mathrm{OP}_1 >< \mathrm{OP}_2 &= \mathrm{OP}_1 < \mathrm{OP}_2 \text{ if } \mathcal{P}_1 = \emptyset \wedge \mathcal{P}_2 \neq \emptyset & (1)\\
&= \mathrm{OP}_1 > \mathrm{OP}_2 \text{ if } \mathcal{P}_1 \neq \emptyset \wedge \mathcal{P}_2 = \emptyset & (2)\\
&= \mathrm{OP}_1 \times \mathrm{OP}_2 \text{ if } \mathcal{P}_1 \neq \mathcal{P}_2 & (3)\\
&= \mathrm{OP}_1 - \mathrm{OP}_2 \text{ if } \mathcal{P}_1 \subseteq \mathcal{P}_2 & (4)\\
&= \mathrm{OP}_1 {=<} \mathrm{OP}_2 \text{ if } \mathcal{P}_1 \subseteq \mathcal{P}_2 & (5)
\end{aligned}$$

The first two rewrites ((1) and (2)) are two special cases where only one of the two involved operations is executed in parallel and has a none empty partitioning requirement. Consequently the repartitioning turns into an initial partitioning respectively a full merge operation. The third rewrite (3) is the most general rewrite and can always be applied as it just replaces a full merge and subsequent initial partitioning with a repartitioning, which can be parallelized independent of the concrete partitioning of either operation. The two alternatives ((4) and (5)) however can only be applied, if the result of the first operation OP$_1$ is partitioned on a subset of the partitioning column set required for the input of second operation OP$_2$. Those two rewrites thereby leverage the fact that a result partitioned on column C_1 alone is also partitioned on columns C_1 and C_2 because two rows that agree on C_1 and C_2 also agree on C_1 alone and consequently they are in the same partition ($\{C_1\}^g \Rightarrow \{\{C_1, C_2\}^g\}$). While rewrite (4) keeps the existing partitions from OP$_1$ intact, the alternative rewrite (5) allows to increase the number of partitions—and consequently the degree of parallelization—using the partial repartitioning operation $=<$. Which of those two rewrites is chosen by the framework is therefore implied by the degree of parallelization defined for each operation and whether the full degree of parallelization is used during each logical operation. The partial repartitioning operation ($=<$) also introduces another basic rewrite (6), which reverses the partial repartitioning operation by applying a partial merge operation ($=>$):

$$\mathrm{OP}_0{=<}\mathrm{OP}_1 >< \mathrm{OP}_2 = \mathrm{OP}_0{=<}\mathrm{OP}_1{>=}\mathrm{OP}_2 \quad (6)$$
$$\text{if } \mathcal{P}_0 \subseteq \mathcal{P}_2 \wedge \mathcal{P}_1 \text{ determ.}$$

The partial merge can only be applied together with a previous partial repartition, because a result partitioned on columns C_1 and C_2 is not partitioned on C_1 alone, since two rows with the same value for C_1 may be in different partitions ($\{\{C_1, C_2\}^g\} \nRightarrow \{C_1\}^g$). However with a previous partial repartition we take advantage of the fact that for $\mathcal{P}_1$ we can for instance apply $\{C_1^g, C_2^g\} \Rightarrow \{\{C_1, C_2\}^g\}$ whereas for $\mathcal{P}_2$ we may apply $\{C_1^g, C_2^g\} \Rightarrow \{C_1\}^g$ when reversing the partitioning on C_2 during the partial merge operation.

Algorithm 1: OptimizeExpr(*expr*, *reqd*)

```
Input: Expression expr, ReqdProperties reqd
Output: Set of QueryPlans plans
/*Enumerate all the possible logical rewrites */
LogicalTransform(expr);
foreach logical expression lexpr do
    /*Try out implementations for its root operator */
    PhysicalTransform(lexpr);
    foreach expression pexpr that has physical
    implementation for its root operator do
        ReqdProperties reqdChild =
            DetermineChildReqdProperties(pexpr, reqd);
        /*Optimize child expressions */
        foreach Child of pexpr do
            QueryPlans childPlans =
                OptimizeExpr(pexpr.Child, reqdChild);
            foreach planChild in childPlans do
                DlvdProperties dlvd =
                    DeriveDlvdProperties(planChild);
                if PropertyMatch(dlvd, reqd) then
                    EnqueueToValidPlans();
                end
            end
        end
    end
end
plans = CheapestQueryPlans();
return plans;
```

All of the above rewrites do also apply in the context of nested operations such as loops. In this case however we need to distinguish between inner rewrites affecting succeeding operations within a nested operation (e.g. loop(<OP_1>-<OP_2>) = loop($\text{OP}_1 \times \text{OP}_2\times$)) and outer rewrites (e.g. OP_1><loop(OP_2) = $\text{OP}_1\times$loop(OP_2)) affecting the boarders of the nested operation connecting inner operations with outer operations. We therefore apply first inner rewrites and then follow up with outer rewrites, which does in the case of a loop also take loop invariants into account. The details of loop invariant handling are discussed in [5]. If a nested loop operation requires a loop invariant input to be partitioned this global structural property will be passed to the outside and considered during outer rewrites.

6.2 Rewriting Strategy

Following the Algorithm 1 from SCOPE [24], we first traverse the dataflow plan downwards starting from the top nodes to the child nodes to propagate required properties. In the case of nested operations we first step down the embedded-plan before we continue to with the surrounding plan. For each logical operator the optimizer considers different alternative physical operators, derives which properties their inputs must satisfy, and requests plans for each input. For example, a hash join operation would request required partitions on the join attributes from its two child expressions.

In the case of an UDF operation, we use the annotations to derive required and provided properties. The function *DetermineChildReqdProperties* in Algorithm 1 thereby derives *structural properties* required by the UDF given the annotations referring to the UDF pre-conditions.

Example: Assume during the first traversal from parent to child we reach the `sample` UDF from *Test Sample* in Figure 1 with the requirement of its hash join parent that its result be partitioned on {`Id_make`}. Since the `sample` UDF has the annotations (**minPart**(NONE), **maxPart**(ANY) and **KEY**(=)) the optimizer considers at least the following three alternatives:

1. Execute the UDF without parallelization by setting partitioning of the UDF to NONE ($\mathcal{P} = \{\texttt{NONE}\}$) and propagate this requirement to its child expression.
2. Execute the UDF with partitioning ANY and request ANY partitioning ($\mathcal{P} = \{\texttt{ANY}\}$) from its child expression.
3. Execute the UDF with partitioning on {`Id_make`} by setting ANY to the requested partitioning of the parent node ($\mathcal{P} = \{\texttt{Id\_make}\}$) and therefore tunnel through this requirement to its child expression.

The algorithm passes all of those requirements while calling the OptimizeExpr function recursively for each child expression. This recursive call is repeated until data source nodes (leaf nodes) are reached returning the data scource node as *CheapestQueryPlans*.

From there on we traverse the dataflow plan in the reverse direction from child to parent. For each returned (sub-)query plan the *DeriveDlvdProperties* function is called to determine the delivered properties. In the case of an UDF operation the function *DeriveDlvdProperties* derives *structural properties* delivered by the UDF given the annotations referring to the UDF post-conditions and the properties of the UDF inputs.

Subsequently the *PropertyMatch* function compares the delivered properties of the child plan with the required properties of the current operation. In the case of property mismatch the optimizer introduces exchange operations in order to build a valid plan combining the delivered (sub-)plan with the current operation. As discussed in Section 6.1, the basic exchange operation regarding *global structural properties* is a full merge followed by an initial partitioning operation. Based on the rewrites discussed in Section 6.1, alternative exchange operations can be derived. If multiple different rewrites are applicable for the introduced exchange operation the optimizer examines all the alternatives and selects the best plan based on estimated costs. For each required property at least one plan alternative is added to valid plans, unless the optimizer prunes the alternative due to cost-based heuristics. It is also a cost-based decision to find the optimal degree of parallelism for each sub-plan. Available partitions for base tables, number of cores and nodes in the distributed landscape, but also communication costs and I/O are factors for these decisions. Furthermore, we use the cost and size annotations for UDFs and known cost formulas for relational operators for this decision.

Example: Assume during the reverse traversal from child to parent we reach the `sample` UDF from *Test Sample* in Figure 1 with the delivered property *dlvd* of its child plan (a hash join) partitioned on {`Id_test`}. Given the required properties *reqd* of the `sample` UDF ($\mathcal{P} = \{\texttt{NONE}\}, \{\texttt{ANY}\}, \{\texttt{Id\_make}\}$) at least following three plans are considered:

1. Execute the UDF without parallelization and add an basic exchange operation:
   $\text{HJ}_{\mathcal{P}=\{\texttt{Id\_test}\}}$><$\text{UDF}_{\mathcal{P}=\{\texttt{NONE}\}}$
2. Execute the UDF with partitioning on {`Id_test`} by setting ANY to the delivered partitioning of the child plan and add an exchange operation.
   $\text{HJ}_{\mathcal{P}=\{\texttt{Id\_test}\}}$><$\text{UDF}_{\mathcal{P}=\{\texttt{Id\_test}\}}$
3. Execute the UDF with partitioning on {`Id_make`} and add an basic exchange operation:
   $\text{HJ}_{\mathcal{P}=\{\texttt{Id\_test}\}}$><$\text{UDF}_{\mathcal{P}=\{\texttt{Id\_make}\}}$

Those basic exchange operations can be rewritten as follows:

1. $\text{HJ}_{\mathcal{P}=\{\texttt{Id\_test}\}}$><$\text{UDF}_{\mathcal{P}=\{\texttt{NONE}\}}$
   (a) Based on rewrite rule (2):
   $\text{HJ}_{\mathcal{P}=\{\texttt{Id\_test}\}}$>$\text{UDF}_{\mathcal{P}=\{\texttt{NONE}\}}$
2. $\text{HJ}_{\mathcal{P}=\{\texttt{Id\_test}\}}$><$\text{UDF}_{\mathcal{P}=\{\texttt{Id\_test}\}}$
   (a) Based on rewrite rule (4):
   $\text{HJ}_{\mathcal{P}=\{\texttt{Id\_test}\}}$-$\text{UDF}_{\mathcal{P}=\{\texttt{Id\_test}\}}$
   (b) Based on rewrite rule (5):
   $\text{HJ}_{\mathcal{P}=\{\texttt{Id\_test}\}}$=<$\text{UDF}_{\mathcal{P}=\{\texttt{Id\_test}\}}$
3. $\text{HJ}_{\mathcal{P}=\{\texttt{Id\_test}\}}$><$\text{UDF}_{\mathcal{P}=\{\texttt{Id\_make}\}}$
   (a) Based on rewrite rule (3):
   $\text{HJ}_{\mathcal{P}=\{\texttt{Id\_test}\}} \times \text{UDF}_{\mathcal{P}=\{\texttt{Id\_make}\}}$

The optimizer examines all the alternatives and selects the best plan for each property group based on estimated costs. Lets assume the optimizer selects following three plans and returns them as *CheapestQueryPlans* to its consumer:

1. $\text{HJ}_{\mathcal{P}=\{\texttt{Id\_test}\}}$>$\text{UDF}_{\mathcal{P}=\{\texttt{NONE}\}}$
2. $\text{HJ}_{\mathcal{P}=\{\texttt{Id\_test}\}}$-$\text{UDF}_{\mathcal{P}=\{\texttt{Id\_test}\}}$
3. $\text{HJ}_{\mathcal{P}=\{\texttt{Id\_test}\}} \times \text{UDF}_{\mathcal{P}=\{\texttt{Id\_make}\}}$

With the consumer being a hash join with required property of ($\mathcal{P} = \{\texttt{Id\_make}\}$) the following plans are considered during the next higher level of the recursion:

1. $\text{HJ}_{\mathcal{P}=\{\texttt{Id\_test}\}}$>$\text{UDF}_{\mathcal{P}=\{\texttt{NONE}\}}$><$\text{HJ}_{\mathcal{P}=\{\texttt{Id\_make}\}}$
   (a) Based on rewrite rule (1):
   $\text{HJ}_{\mathcal{P}=\{\texttt{Id\_test}\}}$>$\text{UDF}_{\mathcal{P}=\{\texttt{NONE}\}}$<$\text{HJ}_{\mathcal{P}=\{\texttt{Id\_make}\}}$
2. $\text{HJ}_{\mathcal{P}=\{\texttt{Id\_test}\}}$-$\text{UDF}_{\mathcal{P}=\{\texttt{Id\_test}\}}$><$\text{HJ}_{\mathcal{P}=\{\texttt{Id\_make}\}}$
   (a) Based on rewrite rule (3):
   $\text{HJ}_{\mathcal{P}=\{\texttt{Id\_test}\}}$-$\text{UDF}_{\mathcal{P}=\{\texttt{Id\_test}\}} \times \text{HJ}_{\mathcal{P}=\{\texttt{Id\_make}\}}$
3. $\text{HJ}_{\mathcal{P}=\{\texttt{Id\_test}\}} \times \text{UDF}_{\mathcal{P}=\{\texttt{Id\_make}\}}$><$\text{HJ}_{\mathcal{P}=\{\texttt{Id\_make}\}}$
   (a) Based on rewrite rule (4):
   $\text{HJ}_{\mathcal{P}=\{\texttt{Id\_test}\}} \times \text{UDF}_{\mathcal{P}=\{\texttt{Id\_make}\}}$-$\text{HJ}_{\mathcal{P}=\{\texttt{Id\_make}\}}$
   (b) Based on rewrite rule (5):
   $\text{HJ}_{\mathcal{P}=\{\texttt{Id\_test}\}} \times \text{UDF}_{\mathcal{P}=\{\texttt{Id\_make}\}}$=<$\text{HJ}_{\mathcal{P}=\{\texttt{Id\_make}\}}$

Since all of those alternatives deliver the same property ($\mathcal{P} = \{\texttt{Id\_make}\}$) for the next higher level of the recursion the optimizer might decide to select only one of those alternative based on estimated costs.

7. EVALUATION

In this section we evaluate the impact of our optimization strategy for workflows using both relational operators and UDFs. We use the example introduced in Section 2 to show the effect of optimizing this plan for increased parallelization and better scalability. Each plan was executed at least 5 times and in case of strong variation up to 15 times. The numbers we present in this section are the median values of those measurements.

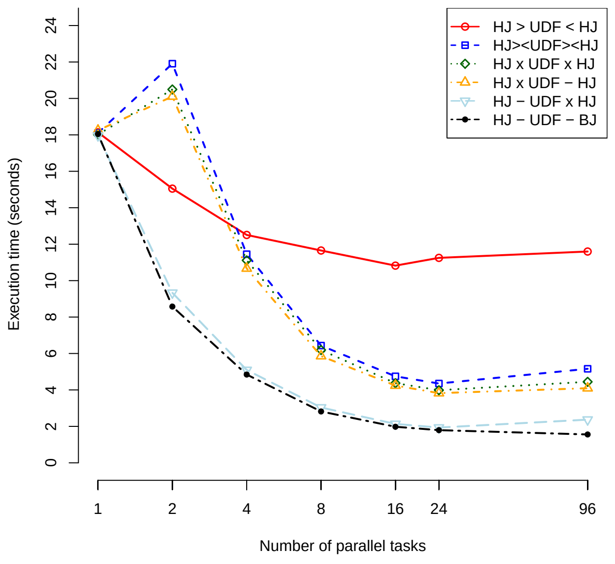
The experiments were conducted using the SAP HANA database on a four node cluster. Every node had two Intel Xeon X5670 CPUs with 2.93 GHz, 148 GB RAM and 12 MB L3 cache. As each CPU is equipped with 6 cores and hyper threading, this resulted in 24 hardware threads per node. The measurements on a single node are labeled *24 local*, and the measurements in the distributed landscape are labeled with *4x6 dist.* if four nodes with up to 6 parallel operators were used and *4x24 dist.* if all resources of all 4 nodes were used. We also used a stronger single-node machine labeled *24 LOC* because some experiments did not run with 148 GB RAM available. This machine is an Intel Xeon 7560 CPU with 2.27 GHz 24 MB L3 cache per CPU and 256 GB RAM and 16 execution threads per CPU including hyper threading.

7.1 Test Sample

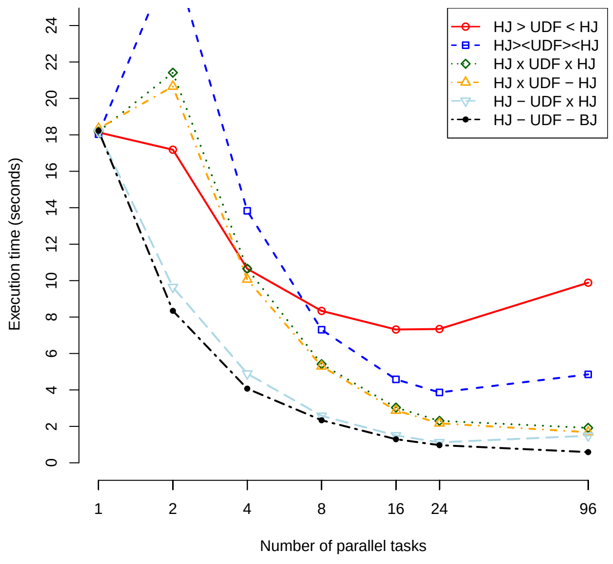
We start examining the sub-plan called *Test Sample* in Figure 1 of the example use case. Techniques for parallelizing the hash-join in a distributed environment are well-known [24]. For the measurements the table `Test_Cases` is partitioned based on `Id_Test`, and in the distributed case we also store the partitions for local access. As the join predicates of the two join operations are different, a repartitioning is required if we want to partition the input for both join operations. Alternatively, a broadcast join can be performed in either case.

We focus on optimizing the UDF (called `sample` UDF, see Script 1) which performs a biased sampling of the test cases. Without knowledge about the ability to parallelize this UDF, we have to first merge the input of the UDF, execute the UDF in a single call, and then potentially split its output again for parallel processing. This gives us our first baseline for our experiments: HJ>UDF<HJ. We have analyzed two more baseline plans that take parallelization of the UDF into account: with full merge repartitioning (HJ><UDF><HJ) and with parallel repartitioning (HJ×UDF×HJ).

It is our goal to do better than that, and the annotations introduced in Section 4 will help to rewrite the plan such that it can be parallelized better, and consequently evaluates faster than the initial alternative. The `sample` UDF can be processed in parallel because it does not require any specific partitioning (**minPart**(NONE), **maxPart**(ANY)). With that knowledge the optimizer can either align the UDF with the left join (HJ−UDF×HJ) or the right join (HJ×-UDF−HJ). Even without cost estimates available, the SIZE annotation of the UDF indicates that the first alternative is the better choice because sampling leaves only approximately 10% of its input. Avoiding repartitioning by using a broadcast join instead of a hash join implementation introduces three more plan alternatives: 1) HJ−UDF−BJ 2) BJ−UDF−HJ and 3) BJ−UDF−BJ. But since the first join works on very large input tables, the last two options are easily excluded, and thus we will only investigate the alternative



(a) Single host



(b) Four node cluster

Figure 2: Scale out of *Test Sample* (Join−UDF−Join) with different execution plans

HJ−UDF−BJ.

In Figure 2 we plot how the plans scale with an increasing number of parallel operators on the x-axis—either on a single node (Fig. 2a)[4] or on up to four nodes (Fig. 2b). In Figure 2b we use one node for the case of no parallelism, and two nodes for up to two parallel operators. For the other measurement in this figure we used all four nodes.

In Figure 2 the baseline plan (HJ>UDF<HJ) with only a single instance of the UDF has the worst scaling behavior. This clearly indicates the benefit of integrating the optimization of relational operators and UDFs. Once we parallelize processing the UDF with a shuffle (plan alternatives HJ×UDF×HJ and HJ×UDF−HJ) we observe a much better scaling behavior. The performance degradation from 1 to 2 parallel operators is caused by the additional overhead of shuffling while the benefit of parallelizing the UDF is still small.[5] But as we increase the degree of parallelism, performance improves significantly. As expected, we observe an even better performance when we keep the partitioning for the first join and the `sample` UDF (plans HJ−UDF×HJ and HJ−UDF−BJ) because this minimizes the communication effort while we enjoy the benefits of parallelization. For up to 24 parallel operators, it does not matter if we shuffle the output of the UDF and use a hash join or if we perform a broadcast join because we only deal with relatively small data. After that reshuffling the data for the hash join creates larger overhead than keeping the existing partitioning. Our measurements are similar for the setup with single-host and the scale-out indicating that data transmission costs over the network have limited impact for this part of the plan.

Overall, we see significant benefits by exploiting the annotations for the `sample` UDF as it allows us to parallelize its execution. Of course, the choice between the plan alternatives must be based on cardinality and cost estimates. Again, our annotations help here.

7.2 Build Graph

Next, we analyze plan alternatives for the sub-plan called *Build Graph* in Figure 1. To compute the connected components of related files we need to perform a self join on the table `Change_File` over the column `Hash`. As a result we get pairs of files identified by their `Id_File` that were changed in the same commit, i.e. with the same `Hash` of the change. After that a `filter` UDF removes all pairs of files below a threshold of 80% meaning that we only keep those pairs of files that co-occur in at least 80% of all submissions of both respective files. In addition to that, this UDF folds all `Hash` values associated with the selected pairs of files into a CLOB. The UDF has the following annotations: **minPart**(NONE), **maxPart**(`fileID1`, `fileID2`) and **EXPECTED GROUPING**(`fileID1`). The annotations imply, that no partitioned input is required, that the most fine-grained partitioning is on (`fileID1`, `fileID2`) and that the input must be grouped by `fileID1` within each partition.

The basic plan alternative to parallelize this plan is to perform a hash join exploiting the partitioning of the base table on column `Hash`, merge and split again for the UDF, i.e. HJ><UDF. A slightly more advanced alternative is to shuffle the data, i.e. HJ×UDF. In both cases the UDF would receive its input partitioned by `fileID1` and `fileID2`. In our experiments we will only consider the second alternative because the result of the join is so large that it cannot be processed on a single node. Also, the discussion in Section 7.1 showed that merging and repartitioning scales worse than the shuffle operator. The alternative plan (BJ−UDF) implements the self join using a broadcast join, partitioning one input of the self join over `Id_File` and broadcasting the second one, avoiding a repartitioning between self join and filter UDF.

Table 5 shows the execution times of both plans for execution on a single node (and more powerful) with up to 24 parallel operators (LOC), in the distributed setup with four

[4]the Figure for local measurements on the bigger machine (LOC) looks similar

[5]For better readability we have restricted the scale of the y-axis to 24 sec. The precise value for HJ><UDF><HJ in Fig. 2b is 27.89 sec.

Env.	Plan Version	Time	σ	Factor
24 LOC	HJ×UDF	748.66	143.90	-
24 LOC	BJ−UDF	545.10	19.90	1.37
4x6 dist.	HJ×UDF	433.00	14.30	-
4x6 dist.	BJ−UDF	298.6	5.20	1.45
4x24 dist.	HJ×UDF	448.86	4.50	-
4x24 dist.	BJ−UDF	272.89	3.00	1.64

Table 5: Build Graph

nodes and either up to 24 (4x6 dist.) and up to 96 parallel operators (4x24 dist.).

Avoiding the repartitioning of 1929 mio. records results in a speed-up of factor 1.37 for the local plan and 1.45 (resp. 1.64) for the distributed plans. We would expect an even larger improvement with a tuned implementation of the prototype of our broadcast join. Another finding is that—even though the machines are not directly comparable—as we move from the local plan to the distributed plan the execution time improves. Moreover, the standard deviation (σ) decreases as we increase paralellism, and thus increasing the parallelism makes the runtime more predictable.

Anyhow this speed-up can only be achieved because our two partitioning annotations **minPart** and **maxPart** yield the flexibility for the optimizer to choose a partitioning over `Id_File` instead of using a partitioning over `fileID1` and `fileID2` to execute the UDF in parallel. This flexibility is especially valuable because the following UDF1 (respectivly UDF3) requires a partitioning on `fileID1` (respectively `Id_File`). While the plan HJ×UDF needs to repartition its output again (which takes additional 15 seconds), the alternative (BH−UDF) can directly use the existing partitioning.

7.3 Connected Component Loop

We now turn our attention to the sub-plan called *Connected Components* in Figure 1. The input for this subplan is an edge relation (`fileID1`, `fileID2`) where each edge connects two nodes representing files that were often changed in one commit. The computation of the connected component uses two UDFs—`UDF1` and `UDF2`—inside a loop. The loop iteratively assigns each node a component `groupID` as *min*(`groupID`, `fileID1`, `fileID2`) until no new `groupID` is derived (see Script 2). Since the `UDF1` processes the data grouped by `fileID1` and the `UDF2` processes the data grouped by `fileID2`, both UDFs can be processed in parallel by partitioning the data accordingly. Consequently, a repartitioning is required between both UDFs.

In our initial implementation (shown in Script 2) `UDF1` and `UDF2` pass the `groupID` as part of the edge set, which means that the full edge set with 21.5 mio. records has to be repartitioned during the iteration between each UDF instance. This leads us to the two basic plan alternatives:

- loop(<UDF1×UDF2>) and
- loop(UDF1×UDF2×).

The former partitions and merges the edge set before and after each loop iteration[6], while the latter uses a shuffle operator.

[6]similar to the situation where the loop logic is handled by an application layer and therefore introduces a pipeline breaker [6]

```
Create Type D1(fileID1 Integer, fileID2 Integer,
         hashList CLOB, groupID Integer);
Create Type D2(changes Boolean);

Create Procedure break(IN control D2, OUT reloop Boolean)
LANGUAGE PSEUDOC AS BEGIN
    Boolean reloop = FALSE;
    forall row in 1:size(control):
      reloop = reloop or control[row].changes;
      if reloop:  break;
END;

Create Procedure UDF1(IN edgeSet D1,OUT edgeSet2 D1)
BEGIN PARALLEL PARTITION(edgeSet(minPart(NONE),maxPart(fileID1)))
EXPECTED(edgeSet(GROUPING(fileID1)))
BEGIN ...  END
ENSURE KEY(=), DETERM(1), PRESERVE ORDER, SIZE(=)
RUNTIMEAPPROX(1 * edgeSet)
END PARALLEL UNIONALL(edgeSet2);

Create Procedure UDF2(IN edgeSet2 D1,OUT edgeSet D1,OUT changes D2)
BEGIN PARALLEL PARTITION(edgeSet(minPart(NONE),maxPart(fileID2)))
EXPECTED(edgeSet(GROUPING(fileID2)))
BEGIN ...  END
ENSURE KEY(edgeSet = edgeSet2),
DETERM(1), PRESERVE ORDER, SIZE(edgeSet = edgeSet2)
RUNTIMEAPPROX(1 * edgeSet2)
END PARALLEL UNIONALL(edgeSet, changes);

Create Procedure connectedComp(IN edgeSet D1,OUT out D1)
LANGUAGE SQLSCRIPT AS
BEGIN
    do{
      call UDF1(:edgeSet, edgeSet2);
      call UDF2(:edgeSet2, edgeSet, changes);
    } while( call break(:changes));
    out = :edgeSet;
END;
```

Script 2: Code for edge set only loop logic

The alternative implementation (shown in Script 3) with `UDF3` and `UDF4` introduces a node set to transfer the `groupID` information between the UDFs. This has the advantage that instead of having to transfer the full edge set of 21.5 mio. records only the much smaller node set with 44 K records has to be passed between the UDFs. In the case of `UDF3` and `UDF4` the partitioning and thereby the parallel execution is based on the edge set (**maxPart**(`fileID1`) and **maxPart**(`fileID2`)), whereas the node set used for transferring the `groupID` update is broadcast (**maxPart**(NONE)) to each parallel executed UDF. As the node set is updated in each iteration, a full merge and repartition is required to integrate the updates of each partition. Given this annotation, the optimizer can make sure that the edge set is passed as loop invariant partitioned by `fileID1` for the `UDF3` and also partitioned by `fileID2` for the `UDF4`. This gives us the plan loop(<UDF3><UDF4>).

Table 6 shows the execution time of this iterative process involving nine iterations until convergence is reached. The plans were executed on one local machine with 24 parallel tasks (*24 LOC*/*24 local* as described at the beginning of the section), in the distributed setup with 4 nodes with up to 6 parallel operators per node (*4x6 dist.*) and also with up to 24 parallel operators per node (*4x24 dist.*).

In the local case (*24 local*) the machine reached the memory limit when processing the initial edge set including the CLOBs for the `Hash`es and started swapping the input tables in and out of memory. As a consequence, the execution time of these sub-plans were quite slow, but we also observed a large standard deviation (σ) in the execution time. An

```
Create Type D1(fileID1 Integer, fileID2 Integer,
             hashList CLOB, groupID Integer);
Create Type D2(changes Boolean);
Create Type D3(fileID Integer, groupID Integer);

Create Procedure UDF3(IN edgeSet D1, IN nodeSet D3, OUT nodeSet2 D3)
BEGIN PARALLEL PARTITION(edgeSet(minPart(NONE), maxPart(fileID1)),
                        nodeSet(minPart(NONE), maxPart(NONE)))
EXPECTED(edgeSet(GROUPING(fileID1)))
BEGIN ...  END
ENSURE KEY(nodeSet2 = edgeSet), PRESERVE ORDER(nodeSet2 = edgeSet),
DETERM(1), SIZE(nodeSet2 = nodeSet)
RUNTIMEAPPROX(edgeSet + nodeSet)
END PARALLEL UNIONALL(nodeSet2);

Create Procedure UDF4(IN edgeSet D1, IN nodeSet2 D3, OUT changes D2,
                      OUT nodeSet D3, OUT edgeSet2 D1)
BEGIN PARALLEL PARTITION(edgeSet(minPart(NONE), maxPart(fileID2)),
                        nodeSet2(minPart(NONE), maxPart(NONE)))
EXPECTED(edgeSet(GROUPING(fileID2)))
BEGIN ...  END
ENSURE KEY(nodeSet = edgeSet, edgeSet2 = edgeSet), DETERM(1),
SIZE(nodeSet = nodeSet2, edgeSet2 = edgeSet),
RUNTIMEAPPROX(edgeSet + nodeSet2),
PRESERVE ORDER(nodeSet = edgeSet, edgeSet2 = edgeSet)
END PARALLEL UNIONALL(nodeSet, edgeSet2, changes);

Create Procedure connectedComp2(IN edgeSet D1, OUT out D1)
LANGUAGE SQLSCRIPT AS
BEGIN
    nodeSet = SELECT Id_File AS fileID, Id_File AS groupID
              FROM Change_File GROUP BY Id_File;
    do{
      call UDF3(:edgeSet, :nodeSet, nodeSet2);
      call UDF4(:edgeSet, :nodeSet2, nodeSet, edgeSet2, changes);
    } while( call break(:changes));
    out = :edgeSet2;
END;
```

Script 3: Code for the loop logic with additional node set

Env.	Plan Version	Time	σ	Factor
24 local	<UDF1×UDF2>	2155.88	499.7	-
24 local	UDF1×UDF2×	2654.26	604.1	0.81
24 local	<UDF3><UDF4>	2501.55	901.5	0.86
24 LOC	<UDF1×UDF2>	652.72	33.0	-
24 LOC	UDF1×UDF2×	262.96	11.7	2.48
24 LOC	<UDF3><UDF4>	181.26	12.0	3.60
4x6 dist.	<UDF1×UDF2>	640.76	5.1	-
4x6 dist.	UDF1×UDF2×	200.80	4.5	3.20
4x6 dist.	<UDF3><UDF4>	87.80	1.5	7.30
4x24 dist.	<UDF1×UDF2>	645.55	30.9	-
4x24 dist.	UDF1×UDF2×	181.27	0.8	3.50
4x24 dist.	<UDF3><UDF4>	62.62	2.8	10.30

Table 6: Connected Component

analysis of this effect revealed that L3 cache misses were a significant contributor to the longer execution time and also the variance. In contrast to that, the results for the distributed setup show that optimization for the loop execution can speed up the execution considerably. Nevertheless the highest speed-up can still be achieved by providing an efficient UDF implementation, i.e. by moving from `UDF1` and `UDF2` to the optimized implementation using `UDF3` and `UDF4`. Because of this it is of particular importance that the language interface and the annotations are flexible enough to support multiple input and output data structures and can describe their behavior under parallelization independent from each other.

Env.	Plan Version	Time	σ	Factor
24 LOC	basic plan	1793.72	168.1	-
24 LOC	best plan	907.68	73.5	2.0
4x6 dist.	basic plan	1179.25	8.1	-
4x6 dist.	best plan	471.56	9.0	2.5
4x24 dist.	basic plan	1196.27	20.4	-
4x24 dist.	best plan	390.49	9.1	3.1

Table 7: Full Execution Plan

7.4 Full Execution Plan

Finally, we summarize our findings for optimizing the execution plan presented in Section 2. To show the potential of the optimizations presented in this paper, we compare the basic plan alternative with the best alternative we found for each sub-plan. For the sub-plan *Test Sample* we only used one fast alternative (HJ-UDF×HJ) because this part of the plan contributes only a very small fraction to the runtime of the complete workflow. The basic alternative combines the hash join filter UDF (HJ × UDF ×) with the unoptimized connected component (<UDF1×UDF2>) involving an additional repartitioning step. The best plan combines a broadcast join filter UDF (BJ−UDF−) with an invariant-sensitive connected component (<UDF3><UDF4>), which can directly reuse the partitioning done for the `filter` UDF also for the `UDF3`.

Table 7 shows the execution time for the full plan for the local execution (LOC 24). As the sub-plan *Build Graph* could not be executed in reasonable time on one of the nodes, we executed the local plan only on the more powerful machine mentioned at the beginning of the section. The distributed execution plans were generated for four nodes with up to 6 operators in parallel per node (4x6 dist.) and also for up to 24 operators in parallel per node with the same hardware as the previous experiments. As a consequence, the measurements on the local nodes and for the distributed environment are not immediately comparable because the nodes in the distributed landscape are less powerful.

The overall execution time is dominated by building the graph and the connected components. The experiments show that by increasing the parallelism (and distribution) in the execution plans we get lower and more robust execution times—this is a very encouraging result. Only by applying the best possible rewrites we achieve a speed-up of two on a single node. When adding compute power by distributing the plan across four nodes, we achieve even better performance. Also the benefit of an increased degree of parallelism during query execution is pronounced in the distributed case—the performance improvement goes up to a factor of 2.5 or even 3.1 compared to the basic plan alternative. In absolute numbers the slowest plan on the more powerful single node is more than four times slower than the fastest plan in the four node distributed landscape.

8. CONCLUSIONS

In this paper we argue that it is important to combine the optimization of relational operators and user-defined functions because many large-scale data analysis tasks can benefit. To be able to optimize such complex workflows, we propose a number of annotations that enable the optimizer to apply rewrites that increase the ability to parallelize plan execution. We believe that annotations are required because in general a query optimizer will not be able to derive the algebraic properties of a user defined function. Based on these annotations we have developed a set of rewrites that allow for better parallelization of complex workflows. In our experiments we show that these rewrites are beneficial for real-world scenarios. We observe significant speed-up and also lower variance in the query execution times after applying our optimizations. While these results are very encouraging, we need to integrate our rewrites into a cost-based optimizer—this is part of our future work.

9. REFERENCES

[1] Apache mahout. `http://mahout.apache.org/`.

[2] SAP HANA. `http://help.sap.com/hana`.

[3] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing data-parallel computing. In *Proc. of USENIX NSDI*, pages 21–35, 2012.

[4] A. Alexandrov, D. Battré, S. Ewen, M. Heimel, F. Hueske, O. Kao, V. Markl, E. Nijkamp, and D. Warneke. Massively Parallel Data Analysis with PACTs on Nephele. *PVLDB*, 3(2):1625–1628, 2010.

[5] C. Binnig, N. May, and T. Mindnich. SQLScript: Efficiently Analyzing Big Enterprise Data in SAP HANA. In *BTW*, pages 363–382, 2013.

[6] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *PVLDB*, 3(1):285–296, 2010.

[7] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proc. VLDB Endow.*, 1(2):1265–1276, Aug. 2008.

[8] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-Reduce for Machine Learning on Multicore. In *NIPS*, pages 281–288, 2006.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[10] E. Friedman, P. Pawlowski, and J. Cieslewicz. SQL/MapReduce: A Practical Approach to Self-describing, Polymorphic, and Parallelizable User-defined Functions. *Proc. VLDB Endow.*, 2(2):1402–1413, Aug. 2009.

[11] D. Gillick, A. Faria, and J. Denero. MapReduce: Distributed Computing for Machine Learning, 2006.

[12] P. Große, W. Lehner, and N. May. Advanced Analytics with the SAP HANA Database. In *DATA*, pages 61–71, 2013.

[13] P. Große, W. Lehner, T. Weichert, F. Färber, and W.-S. Li. Bridging Two Worlds with RICE Integrating R into the SAP In-Memory Computing Engine. *PVLDB*, 4(12):1307–1317, 2011.

[14] J. M. Hellerstein and M. Stonebraker. Predicate Migration: Optimizing Queries with Expensive Predicates. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, SIGMOD '93, pages 267–276, New York, NY, USA, 1993. ACM.

[15] J. Hopcroft and R. Tarjan. Algorithm 447: Efficient Algorithms for Graph Manipulation. *Commun. ACM*, 16(6):372–378, June 1973.

[16] ISO. *ISO/IEC 9075-2:2003 Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*. International Organization for Standardization, Geneva, Switzerland, 2003.

[17] M. Jaedicke and B. Mitschang. On Parallel Processing of Aggregate and Scalar Functions in Object-relational DBMS. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 379–389, 1998.

[18] M. Jaedicke and B. Mitschang. User-Defined Table Operators: Enhancing Extensibility for ORDBMS. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 494–505, 1999.

[19] T. Neumann, S. Helmer, and G. Moerkotte. On the Optimal Ordering of Maps and Selections Under Factorization. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 490–501, 2005.

[20] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-scale Data Analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 165–178, 2009.

[21] A. Rheinländer, A. Heise, F. Hueske, U. Leser, and F. Naumann. SOFA: An Extensible Logical Optimizer for UDF-heavy Dataflows. *arXiv preprint arXiv:1311.6335*, 2013.

[22] X. Su and G. Swart. Oracle in-database Hadoop: when MapReduce meets RDBMS. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 779–790, New York, NY, USA, 2012. ACM.

[23] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A Warehousing Solution over a Map-reduce Framework. *Proc. VLDB Endow.*, 2(2), Aug. 2009.

[24] J. Zhou, P.-Å. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the scope optimizer. In *ICDE*, pages 1060–1071, 2010.