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

Extended Abstract

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. 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 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. 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. 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.

As a middle ground between a high-level declarative query language and MR, data-oriented workflow engines seem to evolve, and we see our work mostly related to this stream of work, e.g. [1, 7, 3].

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, discussed in more detail in the technical report [6], 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.

SSDBM '14 June 30 - July 02 2014, Aalborg, Denmark

ACM 978-1-4503-2722-0/14/06 ... $15.00.
http://dx.doi.org/10.1145/2618243.2618274.

In 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 introduce in Section 3 the UDF annotations; we base our plan optimizations upon. In Section 4 we give a short overview how we build parallel execution plans based on the annotations and discuss in Section 5 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. We only wanted to execute the tests that are affected by a code change starting with the most relevant ones.

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. 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 [4]. 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[1] 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 details of all involved UDFs are discussed in the technical report [6]. 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

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

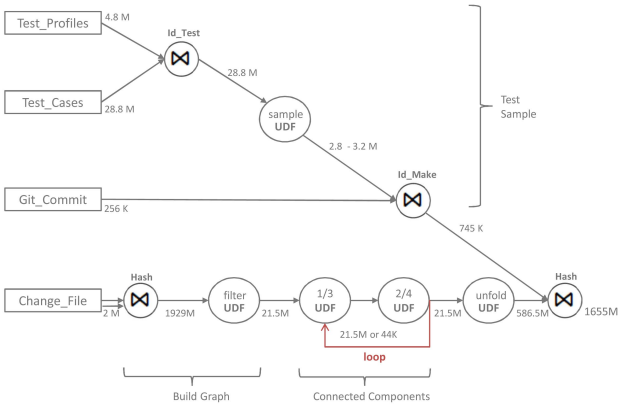

Figure 1: Data preprocessing with UDFs and Joins

difficult to analyze the UDF code and detect if it can and should be parallelized. To solve this problem, we propose annotations in Section 3 to declare opportunities for parallelization but also to annotate how the output cardinality of the UDF relates to its input.

3. UDF ANNOTATIONS

SQLScript [2] procedures are a dialect for stored procedures in the SAP HANA DB to define dataflows and also to include custom coding to be executed in the SAP HANA DB. Several implementation languages are supported including R (RLANG) [5]. 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.

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. A complete presentation of the annotations can be found in the technical report [6]. In this section we will mainly focus in the annotations used for the example in Script 1.

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 regarding the UDF 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.

The first annotation PARTITION precedes the code block of the procedure. We describe the granularity of partitioning

```
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

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 Map 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. Details can be fount in the technical report [6]. 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 depending on surrounding operations and their partitioning requirements.

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.

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 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. 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.

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, but does not implement sorting or grouping by itself.

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.

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. The RUNTIMEAPPROX 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.

4. BUILDING AN EXECUTION PLAN

In order to derive an execution plan including custom (UDF) operations and to leverage the UDF annotations for parallel execution the following steps are implemented[2]:

1. We translate the UDF annotations into a set of properties. Particular the *structural properties* introduced in SCOPE [7] are of relevance to describe grouping, sorting and partitioning properties considered for parallel execution. We enhanced the approach of SCOPE to allow a range of possible partitioning e.g. *global structural properties* for custom (UDF) operations.

2. We build a canonical execution plan combining relational operations and custom (UDF) operations based on the SQL and SQLScript [2] query. The basic dataflow is expressed as DAG using the calculation Engine of the SAP HANA. Details regarding the calculation Engine and custom operations can be found in [5].

3. On the ground of the canonical execution plan we introduce in a second step parallel execution for custom (UDF) operations based on the worker farm paradigm using the properties derived from annotations. This step involves the initial introduction of full (re-)partitioning and full merge operations surrounding the custom (UDF) operations. Details regarding the worker farm and the basic parallel execution plans are discussed in our previous work [4].

4. In a final step we rewrite the execution plan with the introduced parallel UDF operations into an optimized

[2]Discussed in more detail in the technical report [6].

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 1: Full Execution Plan

execution plan. This involves transforming and/or replacing the initial naive full merge and full repartition operations introduced during the previous step. Basic rewrites based on the structural properties of the involved operations and the rewriting strategy are discussed in the technical report [6].

5. 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 in Figure 1 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.

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. Our evaluation regarding the sub-plans can be found at the technical report [6].

Table 1 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.

Overall, we see significant benefits by exploiting the annotations of the 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.

6. 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.

7. REFERENCES

[1] 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.

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

[3] 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.

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

[5] 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.

[6] P. Große, N. May, and W. Lehner. A Study of Partitioning and Parallel UDF Execution with the SAP HANA Database. Technical Report TUD-FI14-03-Mai 2014, TU Dresden, `http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-144026`, 2014.

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