SQLScript: Efficiently Analyzing Big Enterprise Data in SAP HANA

Carsten Binnig, DHBW Mannheim
carsten.binnig@dhbw-mannheim.de

Norman May, SAP AG Walldorf
norman.may@sap.com

Tobias Mindnich, SAP AG Walldorf
tobias.mindnich@sap.com

Abstract: Today, not only Internet companies such as Google, Facebook or Twitter do have Big Data but also Enterprise Information Systems store an ever growing amount of data (called Big Enterprise Data in this paper). In a classical SAP system landscape a central data warehouse (SAP BW) is used to integrate and analyze all enterprise data. In SAP BW most of the business logic required for complex analytical tasks (e.g., a complex currency conversion) is implemented in the application layer on top of a standard relational database. While being independent from the underlying database when using such an architecture, this architecture has two major drawbacks when analyzing Big Enterprise Data: (1) algorithms in ABAP do not scale with the amount of data and (2) data shipping is required.

To this end, we present a novel programming language called *SQLScript* to efficiently support complex and scalable analytical tasks inside SAP's new main-memory database HANA. *SQLScript* provides two major extensions to the SQL dialect of SAP HANA: A functional and a procedural extension. While the functional extension allows the definition of scalable analytical tasks on Big Enterprise Data, the procedural extension provides imperative constructs to orchestrate the analytical tasks. The major contributions of this paper are two novel functional extensions: First, an extended version of the MapReduce programming model for supporting parallelizable user-defined functions (UDFs). Second, compared to recursion in the SQL standard, a generalized version of recursion to support graph analytics as well as machine learning tasks.

1 Introduction

Today, not only Internet companies such as Google, Facebook or Twitter do have Big Data but also Enterprise Information Systems of companies in other industries store an ever growing amount of mainly structured data (called Big Enterprise Data) that needs to be analyzed. For example, large SAP systems hold more than 70TB of structured data in a single database instance. Moreover, looking at complex system landscapes with multiple SAP systems the total data volume that needs to be analyzed is even much higher while the total amount of data is constantly growing.

In a classical SAP system landscape a central data warehouse (SAP BW) based on a stan-

dard off-the-shelf relational database is used to integrate and analyze all enterprise data. In SAP BW most of the business logic for complex analytical tasks (e.g., a complex currency conversion) is implemented in the application layer on top of the database using the imperative language ABAP in order to be independent from a certain database product. However, this architecture has two major drawbacks when analyzing Big Enterprise Data: First, algorithms implemented in ABAP do not automatically scale with the amount of data that needs to be analyzed. Second, data transfer time is growing with the amount of data that needs to be transferred from the database into the application layer.

In order to implement scalable data warehousing solutions today, MapReduce, in particular its open-source implementation Hadoop [DG08, HAD12], is often used instead of a classical data warehouse on top of a relational database. A major reason for the wide adoption of Hadoop is its simple but scalable programming model consisting of two higher-order functions (i.e., map and reduce), that allow complex user-defined functions that can be parallelized efficiently. High-level programming languages for composing MapReduce programs (e.g., Hive [TSJ+10], PigLatin [ORS+08]) as well as further extensions for iteration and recursion (e.g., HaLoop [BHBE10]) further quelled arguments in favor of Hadoop.

However, compared to relational databases Hadoop is inherently inefficient:

- First, Hadoop does not natively support efficient relational operations such as parallel joins in an efficient manner. Instead it supports only a strict sequence of map and reduce functions. This often leads to complex workarounds (e.g., for expressing joins).
- Second, Hadoop always executes full table-scans and does not provide indexes to selectively read data.
- Third, Hadoop does not provide sophisticated cost-based optimizations as relational databases typically do. Instead analytical tasks are often executed as implemented by the user instead of re-ordering operations in the execution plan.
- Finally, Hadoop has an inefficient execution model which materializes and re-partitions all intermediate results even if this is not required in many cases.

In this paper, we present a novel programming language called *SQLScript* that is currently provided by SAP HANA to support complex analytical tasks inside the database. In contrast to other existing work (e.g., HadoopDB [ABPA+09], Hadoop++ [DQRJ+10]) which mainly focuses on fixing the above mentioned shortcomings of Hadoop by integrating ideas from the database world into Hadoop, we directly integrate complex scalable analytical functions into a commercial main-memory database system (SAP HANA) by extending its query language SQL. Thus, we can directly benefit from the maturity of the database and its efficient query optimization and execution techniques.

In its current version that is commercially available *SQLScript* [SQL12] provides two major extensions to the SQL dialect of SAP HANA: A functional and a procedural extension. The functional extension allows the definition of optimizable (side-effect free) functions which can be used to express and encapsulate complex data flows on Big Enterprise Data.

The procedural extension provides imperative control flow constructs like cursors or exceptions as they are defined for SQL stored procedures. While the functional extension is designed to be highly optimizable and parallelizable to efficiently analyze large amounts of enterprise data, the procedural extension is designed to implement orchestration logic (i.e., to pre- and post-process data for the execution of analytical tasks).

As the main contributions, this paper presents two novel language constructs of the functional extension of *SQLScript* that are currently available as an internal prototype: First, we present the integration of a more flexible and efficient version of the MapReduce programming model into *SQLScript* for supporting parallelizable user-defined functions (UDFs) which avoids the above-mentioned drawbacks of its original version in Hadoop. Second, we present an extension to support a generalized version of recursion when compared to recursion in the SQL standard. These two extensions help to implement complex but scalable business functions inside the database. Both extensions are driven by real world use cases to support complex data analytics for Big Enterprise Data. We show an experimental evaluation based on these use cases to show the efficiency of *SQLScript*.

The outline of this paper is as follows: Section 2 introduces the novel programming language of SAP HANA called *SQLScript*. Section 3 then presents the integration of an extended version of the MapReduce programming model into *SQLScript* to support efficient and parallelizable UDFs. Section 4 discusses the second novel extensions to *SQLScript* to support recursion. Finally, the remaining two Sections show an experimental evaluation using two use cases of SAP and discuss related work.

2 SQLScript

2.1 Main Idea

As already mentioned in the introduction, in this paper we present a language called *SQLScript* which integrates complex scalable analytical functions into a SAP's main-memory database system HANA. Therefore, we first discuss why the existing programming models of relational databases (i.e., SQL and SQL stored procedures) are not well suited for analyzing Big Enterprise Data.

Relational databases traditionally offer two approaches to ship its code to the data: (1) declarative SQL statements or (2) stored procedures implemented using a dialect of SQL stored procedures (e.g. PL/SQL or T-SQL) which embed SQL statements for accessing the data. While SQL statements without SQL stored procedures do not allow to implement complex business logic, imperative language extensions such as SQL stored procedures cannot be efficiently optimized and parallelized.

In order to tackle the before-mentioned issues, *SQLScript* provides two major extensions to the declarative SQL dialect of SAP HANA: A functional and a procedural extension. While the functional extension allows the definition of declarative and optimizable (side-effect free) functions[1] to analyze Big Enterprise Data, the procedural extension provides

[1] Created as read-only procedures in the database.

imperative constructs to implement orchestration logic (i.e., to pre- and post-process data for the execution of an analytical task). Consequently, procedures in SAP HANA are either typed as functional (i.e., as read-only) and have a bag-oriented semantics or they are of a procedural type (i.e, with side-effects) and have a tuple at a time semantics [SQL12].

While procedural code is allowed to call functional code in *SQLScript*, this is not allowed vice versa (see Figure 1). The reason is that the functional extension is designed to be scalable to work on large amounts of data (see Section 2.2) while the procedural extension supports more complex language constructs which do not scale as well. Thus calling procedural code from the functional code would mitigate the scalable execution of functional procedures.

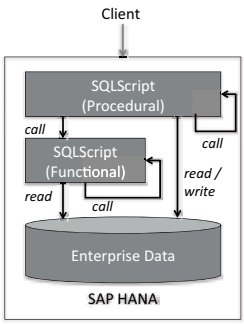

Figure 1: *SQLScript*: functional and procedural extension

The procedural extension provides control flow constructs as they are defined for SQL stored procedures including conditional statements as well as loops over result sets. Moreover, data definition and data manipulation statements (i.e., inserts, updates, deletes) are supported in the procedural extension.

The functional extension supports the definition of declarative read-only procedures (i.e., the side-effect free functions). Such a procedure can have multiple input and output parameters which can either be of a scalar type (e.g., INTEGER, DECIMAL, VARCHAR) or of a table type (as defined in the database catalog). Basic language constructs inside a procedure are single assignments and calls to other read-only procedures. Single assignments can be used to bind the result of a SQL statement (i.e., a table type) or a SQL expression (i.e., a scalar type) to a variable.

Figure 2 shows an example of a read-only procedure, which has two scalar input parameters and returns two output tables (of types `tt_publishers` and `tt_years`) to the caller. The underlying database schema is a simple star schema with a fact table called `orders` and a dimension table called `books`.

The first statement in the procedure assigns a list of identifiers of publishers that publish more books as given by the input parameter `cnt` to the variable `big_pub_ids`. This list of publishers is then used to select those orders of books that have a publisher which is in the given list of big publishers. The result is assigned to the variable `big_pub_books`. Finally, the two final assignments compute the results for the two output parameters: the revenue by publisher `output_pubs` as well as the revenue by year `output_years` (for

the last 10 years).

A complete reference of the current version of SQLScript as it is available in the commercial version of SAP HANA can be found in [SQL12].

```
CREATE PROCEDURE analyzeSales(IN cnt INTEGER, IN year INTEGER,
OUT output_pubs tt_publishers, OUT output_year tt_years)
LANGUAGE SQLSCRIPT READS SQL DATA AS
BEGIN
  big_pub_ids =   SELECT pub_id FROM books -- Query Q1
                  GROUP BY pub_id HAVING COUNT(isbn) > :cnt;
  big_pub_books = SELECT o.price, o.year, o.pub_id -- Query Q2
                  FROM :big_pub_ids p, orders o
                  WHERE p.pub_id = o.pub_id;
  output_pubs =   SELECT SUM(price), pub_id -- Query Q3
                  FROM :big_pub_books
                  GROUP BY pub_id;
  output_year =   SELECT SUM(price), year -- Query Q4
                  FROM :big_pub_books
                  WHERE year BETWEEN :year-10 AND :year
                  GROUP BY year;
END;
```

Figure 2: *SQLScript*: functional extension

The functional extension of *SQLScript* addresses the following drawbacks of the SQL dialect in HANA which also hold for many other SQL dialects in relational databases:

- Decomposing an SQL query can only be done using views. However when decomposing complex queries using views, all intermediate results are visible and must be explicitly typed. Moreover SQL views cannot be parameterized which limits their reuse. *SQLScript* supports decomposition by assignments and parameterization.
- An SQL query can only return one result at a time. As a consequence the computation of related result sets must be split into separate, for the database independent, queries which prevents optimization potentials. *SQLScript* supports multiple input and output parameters.
- Purely declarative SQL queries do not have features to express complex business logic (e.g. the currency conversion of SAP). Only calls to UDFs in a SQL query (as defined in the SQL standard) enable complex business logic. However, these procedures are implemented using imperative SQL stored procedures and thus can not be optimized and parallelized efficiently. The functional extension of *SQLScript* is declarative and thus supports efficient optimization and parallelization.

Moreover, the functional extension of *SQLScript* also addresses the following shortcomings of MapReduce programs in Hadoop: The declarative nature of *SQLScript* allows for optimizations inside the database which are not available for Hadoop programs. Moreover, the integration of *SQLScript* into a relational database provides a streaming execution model instead of an always materializing execution model with efficient relational operators as well as index structures for selectively reading data from tables. Details about the

optimization and execution of *SQLScript* read-only procedures of the functional extension are discussed in the following Section.

2.2 Optimization and Execution

For execution, a read-only procedure is compiled into a data-flow graph (consisting of relational operators) and optimized. For optimization, novel rules have been added to the rewriting phase of the optimizer of SAP HANA to rewrite graph-based plans instead of tree-based plans only (see [BRFR12] for more details). An execution plan for the example of Figure 2 is shown in Figure 3 (left hand side). For simplification, the plan shows boxes which represent the individual query fragments of the procedure instead of single relational operators.

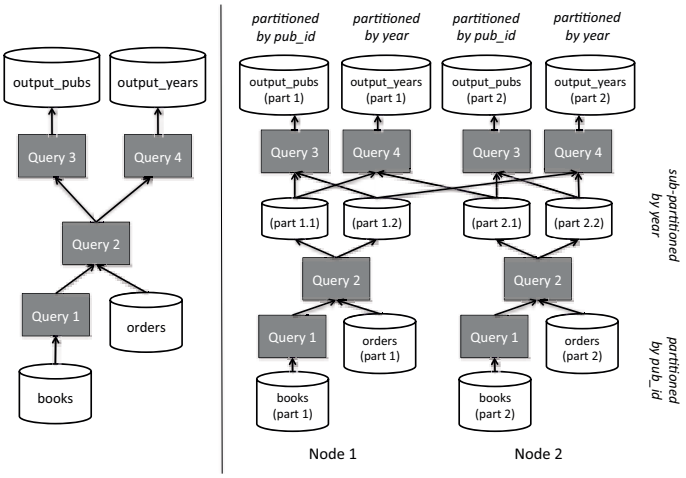

Figure 3: *SQLScript*: execution plan of a read-only procedure

During the execution, SAP HANA materializes intermediate results that are consumed by more than one operator. In the example before, the intermediate result produced by Query Q2 gets materialized since it is consumed by the operators of Query Q3 and Query Q4. Materialization in SAP HANA is also used to re-partition data for better parallelism.

Assume, that the tables `book` and `orders` in the example before are co-partitioned by the attribute `pub_id`. In this case, all queries (Query Q1, Query Q2 and Query Q3 shown in Figure 3 on the right hand side) can be executed in parallel using the same partitioning scheme. However, Query Q4 needs to repartition its input by the attribute `year`. Therefore, the intermediate result of Query Q2 is materialized using a partitioning scheme which partitions the result by `pub_id` and sub-partitions each partition by the attribute `year`. If the plan is executed on different nodes (as in the example), Query Q3 can read all local sub-partitions from one node while Q4 must read sub-partitions from different nodes (see Figure 3 on the right hand side).

3 Generalized MapReduce

3.1 Main Idea

MapReduce is a programming model introduced by Google to analyze big data [DG08]. MapReduce is often applied in use cases with unstructured and semi-structured data (e.g., log analysis) but can also be found as a replacement for classical warehouse solutions on structured data. Originally, the interfaces of both functions *map* and *reduce* are defined as follows:

$$map(k_1, v_1) \rightarrow list([k_2, v_2])$$
$$reduce(k_2, list([v_2])) \rightarrow list([k_3, v_3])$$

Logically, both functions *map* and *reduce* work on tuples with a key and a value. The *map* function processes each incoming tuple $[k_1, v_1]$ separately and produces a list (i.e., a table) of tuples $[k_2, v_2]$. Therefore, each individual map call could be executed in parallel without synchronizing. In a subsequent shuffle step, the output of the *map* function is grouped by the distinct key values of k_2. This step is implicitly executed by the framework. The result is then passed as input to the *reduce* function. Finally, the *reduce* function typically aggregates the values in $list([v_2])$ with the same group key k_2 and returns one or a several tuples $[k_3, v_3]$ as output (i.e., again a table).

In SAP HANA, we use the MapReduce programming model only for structured data (i.e., tables). Thus, we define the interfaces based on structured table types instead of key-value pairs. The table types define the structure of the input and output data. Moreover, to allow parameterization we extend the original interface definitions of both functions to support multiple input tables as well as multiple scalars (e.g., this enables the implementation of joins in both functions). Thus, in SAP HANA both functions *map* and *reduce* are logically defined as follows:

$$map(P, [T_1, ..., T_k], [s_1, ..., s_l]) \rightarrow Q$$
$$reduce(R \text{ GROUP BY } [a_1, ..., a_x], [T'_1, ..., T'_m], [s'_1, ..., s'_n]) \rightarrow S$$

The *map* function gets a table P (with a given table type) as input and applies the user-defined *map* function to each tuple $p \in P$ individually. For each tuple p, the *map* function can append multiple tuples to the output table Q (with a given table type) . The *reduce* function gets a table R (with a given table type) as input and applies the user-defined *reduce* function to each group in table R. The grouping specification is given by a list of group-by attributes $[a_1, ..., a_x]$. For each group, the *reduce* function can append multiple tuples to its output table S (with a given table type).

As mentioned before, compared to the classical MapReduce framework, the *map* and the *reduce* function has additional input parameters: (1) a list of input tables $[T_1, ..., T_k]$ respectively $[T'_1, ..., T'_m]$ and (2) a list of scalar values $[s_1, ..., s_l]$ respectively $[s'_1, ..., s'_n]$

that can be used to parameterize the code. While the input table P of the map function is processed row-wise and the input table Q of the reducer is processed group-wise, the additional input tables can be read completely by both functions. A typical example of such an additional input table T_i is a currency conversion table that is be used to lookup exchange rates inside the *map* function for each row in P.

Another major difference to the classical programming model of the MapReduce framework is that there is no strict sequence of *map* and *reduce* functions (i.e., the output of a *map* function does not need to be consumed by a *reduce* function in *SQLScript*). Instead, any arbitrary sequence of operations can be used (e.g., the output of a *map* function could be used by another *map* function or an SQL query). Thus, complex user-defined functions expressed as mappers or reducers can be mixed with any other SQL statements. Thus a mapper can also be seen as a row-level UDF in SQL with the difference that it can take additional parameters as input.

An example which extends the function in Figure 2 by a *map* and a *reduce* function is given in Figure 4 and Figure 5. The call of the *map* function in Figure 4 calculates a currency conversion (before the aggregations in Q3 and Q4). The *map* function in Figure 5 is defined as a separate read-only procedure which has a special type (i.e., type `MAPPER`). The *map* function is applied to each tuple of its input table `big_pub_books` and the output is assigned to the output parameter `big_pub_books_conv`. Additionally, the function gets a constant input table `conv_rates` and a constant scalar value `target_curr` to implement a currency conversion. The *reduce* function (which replaces aggregation Query Q4) is defined in a similar way as type `REDUCER` in Figure 5.

```
1  CREATE PROCEDURE analyzeSalesConv(
2  IN cnt INTEGER,  IN conv_rates tt_convrates
3  OUT output_pubs tt_publishers, OUT output_year tt_years)
4  LANGUAGE SQLSCRIPT READS SQL DATA AS
5  BEGIN
6    big_pub_ids =   SELECT pub_id FROM books -- Query Q1
7                    GROUP BY pub_id HAVING COUNT(isbn) > :cnt;
8    big_pub_books = SELECT o.price, o.year, o.pub_id, o.curr -- Query Q2
9                    FROM :big_pub_ids p, orders o
10                   WHERE p.pub_id =o.pub_id;
11
12   CALL mapConv(:big_pub_books, [:conv_rates], ["EUR"], :big_pub_books_conv);
13
14   output_pubs =   SELECT SUM(price), pub_id -- Query Q3
15                   FROM :big_pub_books_conv
16                   GROUP BY pub_id;
17
18   CALL reduceBooksByYear(:big_pub_books_conv GROUP BY year,
19     :output_year); -- Query Q4
20 END;
```

Figure 4: *SQLScript*: calling a *map* and a *reduce* function

Logically the user-defined code which is implemented by the mapper refers to one tuple in the input table `big_pub_books` (by calling the method `currentTuple()`). The additional constant input table `conv_rates` is referred as a complete table. In a similar

```
CREATE PROCEDURE mapConv(IN big_pub_books tt_big_books,
[IN conv_rates tt_conv_rates], [IN  target_curr CHAR(3)],
OUT big_pub_books_conv  tt_big_books_conv)
LANGUAGE C++ TYPE MAPPER AS
BEGIN
  //Pseudo code
  string srcCurr = big_pub_book.currentTuple().getColumn("currency");
  decimal price = big_pub_book.currentTuple().getColumn("price");
  decimal rate = getRate(conv_rates, srcCurr, target_curr);
  Tuple big_pub_book_conv = new Tuple();
  big_pub_book_conv.setColumn("convPrice", price*rate);
  //set other columns
  ...

  big_pub_books_conv.appendRow(big_pub_book_conv);
END;

CREATE PROCEDURE reduceBooksByYear(
IN big_pub_books_conv tt_big_books_conv GROUP BY year, [], [],
OUT books_by_year  tt_years)
LANGUAGE C++ TYPE REDUCER AS
BEGIN
  //Pseudo code
  decimal price = 0;
  int year = -1;
  for(Tuple big_pub_book_conv: big_pub_books_conv.currentGroup()){
    price += big_pub_book_conv.getColumn("price");
    if(year==-1)
      year = big_pub_book_conv.getColumn("year");
  }

  Tuple books_by_year = new Tuple();
  books_by_year.setColumn("price", price);
  books_by_year.setColumn("year", year);
  books_by_year.appendRow(book_by_year);
END;
```

Figure 5: *SQLScript*: *map* and *reduce* function

way, the user-defined code which is implemented by the reducer in the example refers to one group of tuples in the input table `big_pub_books_conv` with the same values for the group-by attribute `year`. However, as described in the next Section, the physical execution of both functions is different.

3.2 Optimization and Execution

For compilation a call to a *map* or a *reduce* function is translated into a *map* or a *reduce* operator in the plan. Figure 8 shows the compiled plan for the function in Figure 5. Compared to the logical execution of a *map* function, a *map* operator physically does not process a tuple at a time as input of its input table P. Instead, the *map* operator processes partitions of its input table P (i.e., a bag of tuples) and applies the *map* function to each

input tuple in its partition separately. Each input partition of table P can be processed in parallel by separate *map* operators. A *map* operator can produce multiple output tuples for each input tuple that are appended to its output.

A similar execution model is used for the *reduce* operator. Instead of processing only one input group of its input table Q at a time, the *reduce* operator gets an input partition which can contain multiple groups. Before execution, the *reduce* operator thus groups its input by the given grouping attributes. The *reduce* operator then applies the given user-defined code to each group individually. The *reduce* operator can also produce multiple output tuples for each input group that are appended to its output.

For both functions, the additional input parameters (i.e., the tables and scalar values) are logically replicated to all operators which process an input partition of table P and table Q. If the two operators which refer to the same input parameter are executed on the same node, no physical replication is necessary. In this case, both operators refer to the same input data. However, if two operators which refer to the same input parameter are executed on different nodes, the data must by physically replicated.

Figure 6 shows the execution of a *map* operator and a *reduce* operator which result from the procedures in Figure 5. In this example we see two instances of the *map* operator that are applied in parallel to each row of the two different input partitions. We do not show the two additional input parameters (i.e., the currency conversion table and the target currency) for simplicity. These two input parameters are logically replicated to all instances of the *map* and *reduce* operator.

Moreover, on the right hand side of Figure 6, we see one instance of the *reduce* operator which processes the union of the two output partitions of the two *map* instances. The *reduce* operator has to group its input by the attribute `year` and then processes the two resulting groups separately producing one output tuple per group.

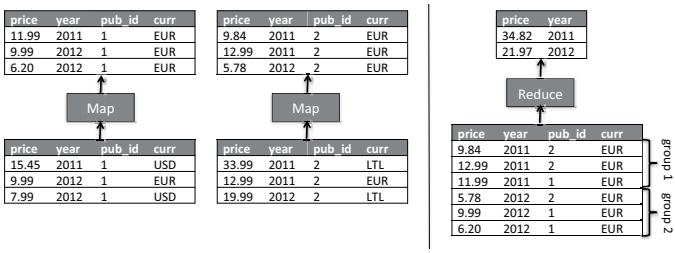

Figure 6: *SQLScript*: execution model for the *map* and the *reduce* operator

For the parallel execution, *SQLScript* allows annotations to define a partitioning specification for the input and output tables of both operators. Figure 7 shows an example of this annotations for both types of functions. The semantics of this partitioning specification is as follows: If the input of a *map* or a *reduce* operator is partitioned by the annotated partitioning specification, then the output is guaranteed to satisfy the given output partitioning specification as well. This helps to avoid irrelevant repartitioning in the plan which is expensive in a parallel distributed execution environment.

For example, if the input of the *map* function in Figure 7 is partitioned by the attribute

pub_id then the operator guarantees that the output satisfies the same partitioning specification. Defining the partitioning specification for a *map* or a *reduce* is optional. For the *reduce* operator, the partitioning schema must be compatible to the given grouping attributes (i.e., it has to guarantee that groups with the same group value are in the same partition).

```
1  CREATE PROCEDURE mapConv(
2  IN big_pub_books tt_big_book PARTITIONED BY pub_id,
3  IN conv_rates table_conv_rates, IN targetCurr STRING,
4  OUT big_pub_books_conv tt_big_books_conv PARTITIONED BY pub_id)
5  LANGUAGE C++ TYPE MAPPER AS ...
6
7  CREATE PROCEDURE reduceBooksByYear(
8  IN big_pub_books tt_big_books_conv GROUP BY year PARTITIONED BY year,
9  OUT books_year tt_big_books_year PARTITIONED BY year)
10 LANGUAGE C++ TYPE REDUCER AS ...
```

Figure 7: *SQLScript*: partitioning specification for a *map* and a *reduce* function

Using this partitioning specification, the optimizer can dynamically detect the need to re-partition the input tables of a *map* or a *reduce* operator. Consequently, the implicit grouping step which is executed in the original MapReduce framework before each *reduce* step can be avoided in *SQLScript* if the partitioning specification of the output of the previous step matches the input partitioning specification. The number of partitions as well as the partitioning method (e.g., by hashing) that are actually used for query processing is determined by the optimizer and depends on several factors (e.g., the partitioning scheme of the input tables, the degree of parallelism, the intermediate result sizes).

In Figure 8, we see a parallelized execution plan for the procedure in Figure 5. In this example, the input tables are partitioned by the attribute pub_id into two partitions. This partitioning scheme is kept for the input of the *map* operator. Thus, the output of the *map* operator is also partitioned by the attribute pub_id (as defined by the interface). This output can be consumed directly by Query Q3 without repartitioning. However, before the output can be consumed by the *reduce* operator (i.e., Query Q4) it must be re-partitioned since Q4 needs to group its input by the attribute year. This re-partitioning can be achieved either by a simple union of both output partitions or by sub-partitioning the output partitions by the attribute year (as described in Section 2 before).

Currently, no other optimizations (like selection-pushdown) are applied for a *map* or a *reduce* operator. However, additional annotations could help to find out which rewrite rules can be applied to these operators. Adding annotations for the rewriting phase is one avenue of future work.

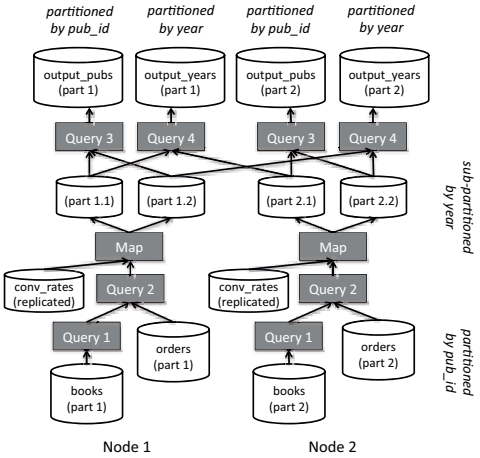

Figure 8: *SQLScript*: execution plan including a *map* operator

4 Generalized Recursion

4.1 Main Idea

Recursion enables different kinds of use cases in data analytics such as graph analysis and machine learning tasks. Major use cases for Big Enterprise Data include the ability to traverse hierarchical data (e.g., hierarchies of product groups, employee hierarchies) and to process graph algorithms (e.g., shortest paths or convex hulls). Moreover, algorithms for machine learning (e.g., k-means) are also require this language construct to examine enterprise data.

Compared to the classical definition of recursion in the SQL standard, *SQLScript* supports a generalized version of recursion. In the SQL standard the definition of a recursive view (using a `WITH RECURSIVE`-statement) is not parameterizable and does not support multiple output parameters. In *SQLScript*, recursion is defined on the procedure level (i.e., read-only procedures can call themselves) instead on the statement level. Thus, scalars and tables can be used as input parameters and a recursive procedure can produce multiple output parameters. Iterative problems can often be re-formulated using recursion. Thus, in the functional extension, we currently do not explicitly support a language construct for iteration. However, the procedural extension of SQLScript [SQL12] supports iteration (e.g., loops over result sets).

Inside a recursive procedure any other read-only procedure (i.e., also *map* or *reduce* functions) can be called. Thus, iterative machine learning algorithms as supported by HaLoop [BHBE10] are supported directly in *SQLScript* by calling *map* functions and *reduce* functions inside the recursion.

The recursive call in *SQLScript* is implemented using a `IF-ELSE` statement at the end of a function (i.e., we support only tail-recursive calls). The termination condition is given

by the predicate of the `IF` clause. The recursive call must be the first statement in the `IF` clause while the subsequent statements must assign results to all output parameters (using a simple assignment, a UNION or a UNION ALL statement). The `ELSE` block is executed if the termination condition holds. That block is only allowed to assign results to the output parameters (again using a simple assignment, a UNION or a UNION ALL statement).

Figure 9 shows an example table which describes the connections between customers (e.g., in a CRM system). A typical task on this graph structure is to compute a list of customers that are connected to a key customer only by edges which have a weight which exceeds a certain threshold. A possible input parameter to such a procedure is the depth, i.e., the distance of customers in the graph that are analyzed when using one certain customer as a starting point.

Frm	To	Weight
1	2	3
1	3	2
2	4	3
2	5	1
2	6	4
3	6	2
6	7	5

Figure 9: Table `CustomerConnections`

This task can be implemented in *SQLScript* using a recursive read-only procedure as the one shown in Figure 10. The procedure has the following input parameters: the maximal depth (parameter `depth`), the current depth (parameter `currDepth`) and a list of connections (parameter `current`) that resulted from the last recursion step (i.e., a table with a `from` and a `to` column). In the first call of the procedure, the parameter `current` holds the customer which is used as starting point.

The first assignment of the procedure filters the relevant connections that exceed a certain threshold on the weight attribute. This intermediate result is an invariant for all recursion steps. The intermediate result table `relevant` is then used to calculate a list of customers that are connected to the given list of customers (i.e., to the input table `current`). If the maximal depth is reached, the recursion stops. Otherwise the list of customers for the next depth in the graph is calculated.

4.2 Optimization and Execution

A recursive procedure is compiled into a cyclic data flow graph as already described in [BRFR12]. Figure 11 shows the data flow graph of the recursive procedure in Figure 7 (left hand side).

In order to optimize recursion, our extension to SAP HANA supports the following rewrites:

- **Materialize Invariants:** Invariants (i.e., partial plans that create intermediate results which are static over different recursion steps) are separated and executed only once.

```
CREATE PROCEDURE convexHull(IN depth INTEGER, IN currDepth INTEGER,
IN current tt_fromto, OUT hull tt_fromto)
LANGUAGE SQLSCRIPT READS SQL DATA AS
BEGIN
  relevant = SELECT Frm, To -- Query Q1
             FROM CustomerConnections
             WHERE weight >= 2;

  temp =     SELECT c.Frm, r.To  -- Query Q2
             FROM :current c, :relevant r
             WHERE c.To = r.Frm;

  currDepth = currDepth  + 1;

  IF(currDepth < depth)  -- Recusive Call C3
    CALL convexHull(depth, currDepth, temp, temp2)
    hull = :temp UNION :temp2;
  ELSE
    hull = :temp;
END;
```

Figure 10: *SQLScript*: recursive procedure

This optimization is shown in Figure 7 on the right hand side: the invariant which is stored in the intermediate result `relevant` is computed by a partial plan only once. Compared to HaLoop materializing invariants is not implicitly hidden in the execution model by caching but explicitly applied during the optimization phase.

- **Internal Rewrites:** Inside a recursive procedure, we can use all normal rewrites such as selection- and projection-pushdown.
- **Cross-Procedure Rewrites:** If the results of a recursive procedure are consumed by other procedures, we can apply the following rewrites: selection- and projection-pushdown of the calling procedure are supported if the respective operator can be pushed over the complete recursive procedure over an input table which is defined recursively. For example, if in Figure 7 a selection `c.frm=1` is executed on top of the result `hull` then this selection can be pushed over the input `current`.

For parallelization, we analyze the plan dynamically for possible partitioning schemes and add repartitioning operations into the plan as described before.

5 Experimental Evaluation

In this Section, we present the experimental evaluation of the two novel functional extensions for *SQLScript* based on use cases of SAP: the generalized versions of MapReduce as well as recursion. Both ideas are implemented at SAP as a prototype in SAP HANA to extend the commercially available version of *SQLScript*.

As hardware we used a single machine with 512GB of main memory and four Intel Xeon

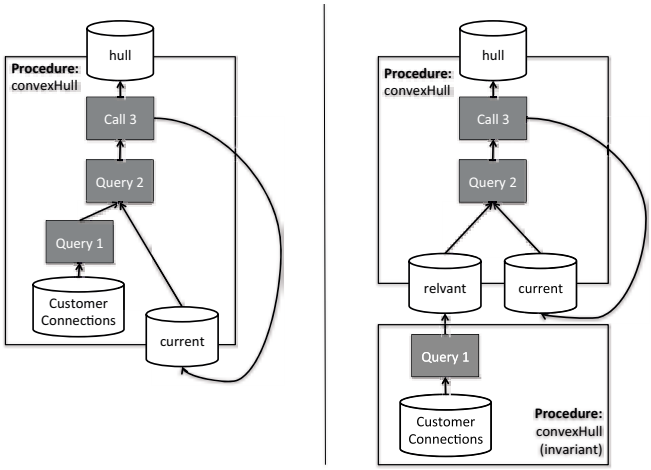

Figure 11: *SQLScript*: execution plan of a recursive procedure

X7560 processors each with eight cores (i.e., 32 cores in total). The software stack consisted of SUSE Linux 11 running a database instance of SAP HANA.

5.1 Experiment: Currency Conversion

The first experiment is based on the Shipping Priority Query (Q3) of the TPC-H benchmark [TPC12] which returns the first 10 selected rows. This query retrieves the unshipped orders with the highest value. Figure 12 shows the original Query Q3:

```
SELECT
   l_orderkey, o_orderdate, o_shippriority
  SUM(l_extendedprice*(1-l_discount)) as revenue,
FROM customer, orders, lineitem
WHERE c_mktsegment = '[SEGMENT]'
   and c_custkey = o_custkey
   and l_orderkey = o_orderkey
   and o_orderdate < date '[DATE]'
   and l_shipdate > date '[DATE]'
GROUP BY l_orderkey, o_orderdate, o_shippriority
ORDER BY revenue desc, o_orderdate
LIMIT 10;
```

Figure 12: TPC-H: query Q3

For the experiment, we extended this query to use a simplified version of the SAP currency conversion before the aggregation on the attribute `extended_price`. Therefore, we first pre-aggregated the data using the currency as an additional group-by attribute. Then, we applied the currency conversion using different implementations (as described below).

Finally, we post-aggregated the result removing the currency from the group-by attribute.

The simplified version of the SAP currency conversion is based on a currency conversion table as shown in Figure 13.

SCurreny	TCurrency	RefDate	Rate
EUR	USD	2012-10-15	1,3
USD	EUR	2012-10-15	0,769
LTL	EUR	2012-10-15	0,289

Figure 13: Table `CurrConv`

The currency conversion is based on the date of the conversion (i.e., the attribute `RefDate`) and has three cases:

- **Direct Conversion:** There exists a conversion rate from the given source to the target currency (e.g., as from EUR to USD or vice versa).
- **Inverted Conversion:** There exists only a conversion rate from the given target to the source currency. Thus, the inverted rate is used (e.g., as from EUR to LTL).
- **Indirect Conversion:** There does neither exist a direct nor a inverted conversion. In this case the conversion must be done using a reference currency (e.g., from LTL to USD we have to use EUR as reference currency).

For the experiment, we executed the Shipping Priority Query (Q3) of the TPC-H benchmark in three variants. (1-SQL: No currency conversion) the original version of Q3 using one SQL query, (2-SQLScript: Generalized MapReduce) a variant of Q3 including the currency conversion implemented as a *map* function which takes the currency conversion table as an additional input parameter and (3-SQLScript: Procedural) a variant of Q3 including the currency conversion implemented using SQLScript procedural code which is called for each row on the pre-aggregated result. Version (1) and (3) thus represent the baselines (lower and upper limit).

For the variant (2) and (3), we extended the original `lineitem` table in the TPC-H schema by a currency column `curr` and used the attribute `o_orderdate` as reference date for the conversion. We generated additional data for the new column `curr` in the table `lineitem` such that each case of the currency conversion must be executed with the same probability. Additionally, we generated a currency conversion table (as shown in Figure 13) holding information for all currencies and reference dates in the `lineitem` table. As target currency for the function call, we used *EUR*.

Figure 14 shows the result of the execution of the three variants mentioned before on different scaling factors (SF) for the TPC-H benchmark (up to SF 25). The variants (1) and (3) have been executed using 32 threads. For variant (2) which uses the *map* operator for the currency conversion, we used 16 and 32 threads (while all other operators of Q3 were still using 32 threads).

As we can see in Figure 14, the variant with the *map* operator is much faster than the procedural SQLScript implementation. The reason is that the procedural SQLScript variant

issues multiple SQL queries for the currency conversion while variant (2) only requires one (complex) user-defined *map* operator which internally builds a hash index on the currency conversion table to do fast lookups of the exchange rates. As a result, the complex user-defined function implemented as a mapper adds only 200ms with 32 threads and 250ms with 16 threads to the runtime of $Q3$ for each scaling factor (since the pre-aggregated result has the same size for all scaling factors). The SQLScript procedural extension adds additional 13s to the runtime of $Q3$.

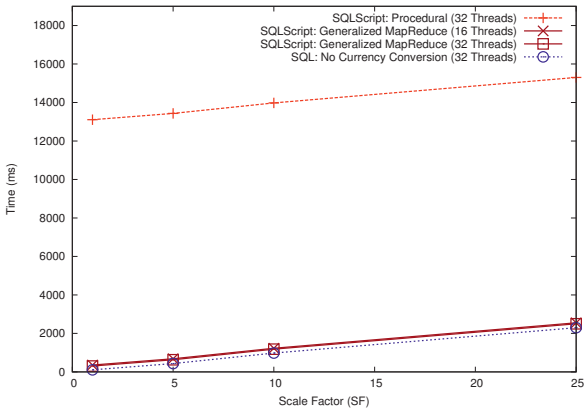

Figure 14: TPC-H query Q3 with and without currency conversion

5.2 Experiment: Graph Analysis

For this experiment we used a recursive procedure which is similar to the one shown in Figure 10 already. As data we used the table `Customer` of the TPC-H benchmark and a table `Livejournal` (which has a similar schema as the table in Figure 9). The table `Livejournal` comes from the Stanford Network Analysis Project [Les12] which provides data from social networks. The table `Livejournal` has approximately $68m$ entries with approx. $5m$ distinct nodes. For the table `Customer`, we used the SF 35 (i.e., approximately $5m$ customers) .

In order to select relevant entries from the table `Livejournal` we join this table with the table `Customer` based on the customer key. Moreover, we select customers from certain nations only to reach a selectivity from 10% to 80% of the `Livejournal` table. The result of this join corresponds to the table `relevant` in Figure 10.

We executed the recursive procedure with a maximal depth of 3 using different optimization variants[2]: one variant which materializes invariants and another variant without this optimization (i.e., the invariant is computed for each iteration). Moreover, we also varied the number of partitions used from 1 to 8 for each of these variants to exploit parallelism (while each operator was configured to use at most 8 threads). Figure 15 shows the runtime for the different variants using selectivities from 10% to 80% for the selection operator on the table `Customer`.

[2]We also executed the procedure with a maximal depth of 5 and 7 but the results looked similar.

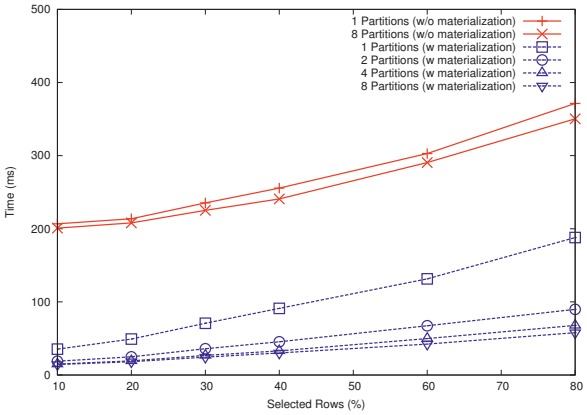

Figure 15: Recursive query with and without materialization of invariants

As we can see in Figure 15, the runtime (of the variants which do not materialize the invariant) is dominated by the redundant execution of the sub-plan which produces the invariant. Moreover, partitioning the plan (on one machine) speeds-up query processing due to parallelism. The results when using 4 and 8 partitions (for the variants with materialization of the invariant) does not show a huge difference since the CPUs of the machine were already saturated using 4 partitions (i.e., 8 threads per partition have been used). Thus, increasing the parallelism to 64 threads on 8 partitions did not show a huge difference in the resulting runtime.

6 Related Work

Most related to *SQLScript* are extensions to Hadoop to tackle its inefficiencies of query processing in Hadoop in different areas such as new architectures for big data analytics, new execution and programming models but also in the field of integrating systems like MapReduce and databases.

HadoopDB [ABPA+09] turns the slave nodes of Hadoop into single-node database instances. However, HadoopDB relies on Hadoop as its major execution environment (i.e., joins are often compiled into inefficient map and reduce operations). Only in its commercial version [BPASP11], HadoopDB presents a component called SideDB, which replaces the Hadoop execution environment by a database to execute operations like joins more efficiently.

Hadoop++ [DQRJ+10] and Clydesdale [KST12] are just two out of many other systems also trying to address the shortcomings of Hadoop, by adding better support for structured data, indexes and joins. However, like other systems, Hadoop++ and Clydesdale cannot overcome Hadoop's inherent limitations (e.g., not being able to execute joins natively).

PACT [ABE+10] and ASTERIX [BBC+11] suggest new execution models, which provide a richer set of operators than MapReduce (i.e., not only two unary operators) in order to deal with the inefficiency of expressing complex analytical tasks in MapReduce. Although

promising, *SQLScript* explores a different design, by focusing on existing databases and novel query optimization techniques.

HaLoop [BHBE10] extends Hadoop by recursive and iterative analytical tasks and improves Hadoop by certain optimization (e.g, caching loop invariants instead of producing them multiple times). *SQLScript* supports a more general version of recursion than HaLoop while optimizations are not implicitly hidden in the execution model (by caching) but explicitly applied during the optimization phase.

In the area of programming languages for big data analytics there are a lot of proposals as well. For example, Hive [TSJ+10] and PigLatin [ORS+08] have been proposed as high-level programming languages for defining map-reduce jobs in Hadoop. Those programs are optimized and then executed using Hadoop as execution environment. *SQLScript* extends these approaches for a better UDF support in databases so that the data-flow graphs including user code can be holistically optimized.

Moreover, major database vendors currently include Hadoop as a system into their software stack and optimize the data transfer between the database and Hadoop e.g. to call MapReduce tasks from SQL queries. Greenplum and Aster Data are two commercial database products for analytical query processing which support MapReduce natively in their execution model. However, to our knowledge they do not support the extended version as we do.

Finally, there has also been a lot of research work on the field of recursion in the context of SQL. In this paper, we extend many known techniques for single SQL queries to procedures with multiple in- and output parameters. For example, we extend the optimization rules presented in [Ord05] (i.e., selection pushdown) to work for recursive SQLScript procedures with multiple in- and output parameters.

7 Conclusions and Outlook

In this paper, we presented a novel programming language called *SQLScript* to support complex analytical tasks in the distributed in-memory database SAP HANA. *SQLScript* provides two major extensions to SQL: A functional and a procedural extension. The functional extension allows the definition of optimizable side-effect free functions which can be used to express and encapsulate complex data flows. Moreover, we presented two novel language constructs of the functional extension of *SQLScript*: First, an extended version of the MapReduce programming model to support parallelizable user-defined functions (UDFs). Second, an extended version of recursion (i.e., iteration) compared to recursion in the SQL standard, which takes multiple input parameters and can produce multiple output parameters. For both extensions, we showed optimization and execution strategies that were analyzed in an experimental evaluation to show their efficiency.

As future work, we plan to extend the static optimization and execution by adaptive techniques (i.e., changing the plan parallelism dynamically). Moreover, we also plan to add better rewrite techniques for the *map*- and *reduce* operators by further annotations. Another major issue includes the debugging and testing of these complex functions on the database side.

References

[ABE+10] Alexander Alexandrov, Dominic Battré, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. Massively Parallel Data Analysis with PACTs on Nephele. *PVLDB*, 3(2), 2010.

[ABPA+09] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1):922–933, 2009.

[BBC+11] Alexander Behm, Vinayak R. Borkar, Michael J. Carey, Raman Grover, Chen Li, Nicola Onose, Rares Vernica, Alin Deutsch, Yannis Papakonstantinou, and Vassilis J. Tsotras. ASTERIX: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[BHBE10] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *PVLDB*, 3(1), 2010.

[BPASP11] Kamil Bajda-Pawlikowski, Daniel J. Abadi, Avi Silberschatz, and Erik Paulson. Efficient processing of data warehousing queries in a split execution environment. In *SIGMOD*, pages 1165–1176, 2011.

[BRFR12] Carsten Binnig, Robin Rehrmann, Franz Faerber, and Rudolf Riewe. FunSQL: it is time to make SQL functional. In *EDBT/ICDT Workshops*, pages 41–46, 2012.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[DQRJ+10] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *PVLDB*, 3(1):518–529, 2010.

[HAD12] Apache Hadoop. http://hadoop.apache.org, 2012.

[KST12] Tim Kaldewey, Eugene J. Shekita, and Sandeep Tata. Clydesdale: structured data processing on MapReduce. In *EDBT*, pages 15–25, 2012.

[Les12] Jure Leskovec. Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/, 2012.

[Ord05] Carlos Ordonez. Optimizing recursive queries in SQL. In *SIGMOD Conference*, pages 834–839, 2005.

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[SQL12] SAP HANA SQLScript Reference. http://help.sap.com/hana/hana_dev_sqlscript_en.pdf, 2012.

[TPC12] TPC-H. http://www.tpc.org/tpch/, 2012.

[TSJ+10] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *ICDE*, pages 996–1005, 2010.