FunSQL: It is time to make SQL functional

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

Franz Faerber
SAP AG
franz.faerber@sap.com

Robin Rehrmann
DHBW Mosbach

Rudolf Riewe
DHBW Mosbach

ABSTRACT
With the rise of cloud-computing and cloud-scale data management the importance of shipping the code of an application to its data has increased tremendously. Especially when offering data analytics on top of traditional relational databases as a service in the cloud, new data-centric programming paradigms become necessary. Traditionally, relational databases offer two approaches to ship code close to the data: declarative SQL statements and imperative stored procedures. While SQL statements can be efficiently optimized and parallelized, stored procedures allow more complex logic that can be efficiently decomposed.

In this paper, we propose a novel functional language which extends SQL called FunSQL. FunSQL combines the best of both worlds: (1) it allows application developers to implement more complex application logic as in SQL only, (2) the application logic can be decomposed efficiently and (3) it can be efficiently optimized and parallelized.

1. INTRODUCTION
With the rise of cloud-computing and cloud-scale data management the importance of shipping the code of an application to its data has tremendously increased. Reasons are e.g. that shipping the code to the data avoids massive data copies from the cloud to the client and enables the usage of the massive resources in the cloud when processing the data [1]. Especially when offering data analytics on top of traditional relational databases as a service in the cloud, new data-centric programming paradigms become necessary.

Relational databases traditionally offer two approaches to ship its code to the data:
- SQL statements implemented using a declarative data definition and manipulation language
- Stored procedures implemented using an imperative language (e.g., PLSQL) which embeds SQL statements for accessing the data

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. DanaC 2012, March 30, 2012, Berlin, Germany. Copyright 2012 ACM 1-4503-1143-4/12/03 ...$10.00.

While SQL statements can be efficiently optimized and parallelized it is hard for application developers to implement complex application logic using SQL only. Reasons are that SQL does not provide an elegant way for decomposing complex statements into smaller (reusable and parameterizable) fragments which can be explicitly reused. Moreover, SQL statements can only return one result at a time.

As a result application developers often implement complex application logic using an imperative language such as Java in the client layer or using a stored procedure dialect as PLSQL in the database layer. While using a stored procedure dialect avoids data copies from the database server to the client, it allows developers to implement application logic that can not be optimized and parallelized efficiently by the database anymore.

The example in Figure 1 provides a very simple code fragment that can often be found in imperative application logic: The application code calculates the sum of selected sales in Europe. In this case, the code fragment could have been easily re-formulated using one single declarative SQL statement. However, application developers do not think purely declarative. Instead, they want to solve a problem in a step-wise manner. Moreover, another important aspect is that reusing the data of intermediate results to produce multiple output results is not possible in SQL in an elegant way.

Running the code fragment in Figure 1 results in a sequence of individual SQL statements that are optimized and executed in isolation. Assume, that the first query returns a set of 30k customers. In that case the while-loop (and thus the SQL statement inside the while-loop) is executed 30k times which leads to a poor performance of the application logic.

In this paper, we propose a novel functional language which extends SQL called FunSQL. FunSQL combines the
best of both worlds: (1) FunSQL allows applications developers to implement more complex application logic as in SQL only, (2) the application logic can be decomposed efficiently and (3) it can be efficiently optimized and parallelized.

The main idea of FunSQL is that application developers decompose logic into functions (with multiple input and output parameters) while a function binds intermediate results of SQL queries to variables using a static single assignment form (as in functional languages). The intermediate results can be used in other subsequent SQL queries as input in addition to database tables and input parameters. As a result, complex application logic can be easily decomposed into a sequence of steps. The example above could be implemented using the function shown in Figure 2 using two single assignments.

```
CREATE FUNCTION topSalesEur (OUT o1 DECIMAL)
BEGIN
VAR r1 = SELECT c_id FROM customer
WHERE c_region = 'EUROPE';
: o1 = SELECT SUM(o_total) FROM orders
WHERE c_id IN :r1 AND o_total > 1000;
END;
```

Figure 2: Application code rewritten in FunSQL

For execution, both queries can be compiled into a single execution plan which is then optimized. Different from SQL, FunSQL supports graph-based (and not only tree-based) execution plans including cycles. Moreover, FunSQL enables (1) the explicit re-usage of intermediate results and (2) supports the creation of multiple result sets per function call (as shown later in Figure 5). Compared to view stacking in SQL, FunSQL not only allows the reuse of the definition of intermediate results but more important the reuse of the data of intermediate results which results in execution plans as shown later in Figure 6. In other words, FunSQL makes multi-query optimization explicit on the language level.

The outline of this paper is as follows:

- Section 2 describes the language constructs (including recursion) of FunSQL.
- Section 3 gives an overview of the compilation process of FunSQL into graph-based execution plans and discusses novel optimization rules for graph-based execution plans (including cycles).
- Finally, Section 4 shows some preliminary results of performance experiments based on the data model of the TPC-H benchmark.
- In the last two sections we discuss related work and conclude with some avenues of future work.

2. LANGUAGE DESCRIPTION

2.1 Basic Constructs

FunSQL supports the definition of side-effect free functions (using the CREATE FUNCTION statement as shown in Figure 2). A function can have multiple input and output parameters which can also be either of a scalar type (e.g., INT, DECIMAL, VARCHAR) or a table type (i.e., a result of a SQL statement). Basic language constructs inside a function are single assignments and calls to other functions. 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.

**Single Assignments:** SQL expressions used in a single assignment return a value of a scalar type and consume other variables of a scalar type. SQL statements used in a single assignment return a table or a scalar value as a result can consume variables of both types (scalar and table types).

SQL statements that are used inside a single assignment are currently limited to read-only queries (i.e. the functions do not have any side-effects). One avenue of future work is to also support update statements which might be collected in an update queue that is applied after the functions has been executed (as it is also supported in XQuery Update Facilities).

**Function Calls:** Calls to other functions can hand over variables as input parameters and consume output parameters (while output parameters are seen as variables bound by a single assignment). Calls to FunSQL functions can also be embedded into imperative procedures or client side logic which prepares the input data for the function call and consumes the results.

Figure 3 shows an example function definition which uses the artifacts listed before.

```
CREATE FUNCTION relSales(IN iyear INT, IN iduration INT, OUT o1 TABLE)
BEGIN
//Function call (scalar result)
VAR salEur = CALL topSalesEur();
//SQL expression (scalar result)
VAR endyear = :iyear + :iduration;
//SQL statement (table result)
:o1 = SELECT SUM(o_total)/:salEur, c_country
FROM orders JOIN customer ON o_cid = cid
WHERE c_region='EUROPE' AND o_year BETWEEN :iyear AND o_year
GROUP BY c_country;
END;
```

Figure 3: Function definition

The function in the example first calls a function topSalesEur which returns the sum of the top sales in Europe (i.e., bound by the variable salEur) and then calculates the result of a scalar expression (i.e., bound by the variable endyear). Afterwards a SQL statement is executed that returns the relative sales per country in Europe in a certain timeframe compared to the top sales using both variables that were defined beforehand.

2.2 Recursion

Many data-processing algorithms (e.g., graph-processing, hierarchy processing) can only be implemented in a general way using either recursion or iteration. FunSQL offers tail-recursive function calls for this purpose which is one major difference from many other graph-based data processing languages [2, 7, 5, 10] that have been proposed in the area of data analytics in the cloud.

Compared to a normal function call a tail-recursive func-
tion call has to define an exit condition as well as a result in case that the recursion ends. For the result of a tail-recursive function call the caller can define if the result of the last function call of the recursion is returned or if the caller gets a union (using UNION OR UNION ALL) of all results produced by each function call.

The following example in Figure 4 shows a simple breadth-first search in a graph (e.g., to search for friends in a social network with a maximum given distance) using a tail-recursive function call. In this example the result of friends is composed of a distinct union of all friends with different distances (i.e. results from different recursion steps). Other graph-processing algorithms (e.g., the Dijkstra-algorithm) can be implemented in a similar way using tail-recursion as well.

```sql
CREATE FUNCTION getFriends(IN illevel INT, IN ifriends TABLE, OUT ofriends TABLE)
BEGIN
VAR newlevel = :illevel + 1;
VAR temp = SELECT i FROM k WHERE i.to = k from;
IF newlevel <= 10 THEN
  ofriends = :temp UNION CALL getFriends(:newlevel, :temp)
ELSE
  ofriends = :temp;
END;
```

**Figure 4:** Tail-recursive function

### 3. COMPILATION AND OPTIMIZATION

This section discusses the compilation of a FunSQL function into an graph-based execution plan (including recursion) as well as some novel optimization-rules for graph-based execution plans.

#### 3.1 Compilation

Functions defined in FunSQL are compiled into a graph-based execution plan which consists of relational operators (representing SQL statements) as well as scalar expressions. Compiling a function into a graph-based execution plan (i.e., a data-flow graph) is trivial because of using the static single assignment form of functional languages.

For simplicity, in this paper we focus on relational operators as nodes of the graph-based execution plan. The difference to execution plans that are compiled for single SQL statements is that the intermediate results in a graph-based execution plan can be consumed by more than one relational operator and cycles are supported (for supporting tail-recursion).

The example in Figure 5 shows a function that has two output parameters (i.e., salF and salO) which are calculated based on a common intermediate result (salYear):

```sql
CREATE FUNCTION sales(IN year INT, OUT salF TABLE, OUT salO TABLE)
BEGIN
VAR r1 = SELECT o FROM orders WHERE o_year = year;
:salF = SELECT SUM(o_total) FROM r1
WHERE o_orderstatus = 'F';
GROUP BY o_month;
:salO = SELECT SUM(o_total) FROM r1
WHERE o_orderstatus = 'O';
END;
```

**Figure 5:** Function with two output parameters

Tail-recursion is compiled as a cycle (i.e., as an iteration) into the graph-based execution plan in Figure 6. The sub-plans for the individual SQL statements of the function are highlighted in this representation.

**Figure 6:** Basic graph-based execution plan

Tail-recursion is compiled as a cycle (i.e., as an iteration) into the graph-based execution plan as it is implemented in most database systems. Different from common table expressions using recursion in SQL, recursive function calls in FunSQL support multiple input and output parameters. Figure 7 shows the plan which is the result of the compilation of a call of the function in Figure 4. The caller wants to know all friends of the person with c_id = 1. The resulting plan contains two new operators (recStart and recEnd) which implement the tail-recursion (as an iteration). The operator recStart marks the begin of the recursion and has two input edges for the same logical input parameter (ifriends in the example): one for the initial input by the caller and another for each recursion step. The operator (recEnd marks the end of the recursion and has two output edges: one to hand over the input to the next recursion step and another output to return the result of each recursion step. The final distinct is an n-ary distinct union over all intermediate temp results.

Finally, calls to other functions can be handled by inlining the sub-graph in order to enable a holistic optimization\(^2\).

#### 3.2 Optimization

For the optimization all rules for rewriting relational algebra expressions (e.g., selection-pushdown) can also be applied - The scalar variable illevel is not shown in the plan for simplicity.

\(^2\)Inlining is only done up to a certain level of complexity since it might cause a massive join-order problem in the resulting plan.
Filter predicate
predicates, a new selection operator is created which uses a projection and selection. In order to push down the two and $q$ pushing selection operators down in a graph-based execution plans.

Figure 7: Tail-recursive graph-based execution plan

Rule 1

Rule 2

Rule 3

Figure 8: Rules for selection-pushdown

Rule 2 is similar to the first rule but describes the push-down of the predicates $p$ and $q$ of two selection operators over a set operator (i.e., union, intersect, and minus).

Finally, rule 3 can be used to push down the predicates $p$ and $q$ of two selection operators over a join operator. The rule can be applied if the attributes of the two predicates $p$ and $q$ refer to only one of the inputs (i.e., input $R$ in the rule). The function attr returns the list of attributes of a predicate or a relation.

Projection-Pushdown: Similar rules can be found for the projection-pushdown. Figure 9 shows one rule which describes how the attribute lists $l$ and $r$ of two projections can be pushed over a selection operator using a new projection operator with the attribute list $s$ which is the result of a union of $l$ and $r$ and the attributes used in the selection predicate $p$.

Recursion: The operators recStart and recEnd are no barrier for optimization. For example, a selection operator which consumes the final result of a recEnd (e.g., in Figure 7) is allowed to be pushed down over the corresponding operator recStart, if it can be pushed down over all intermediate operators inside the recursion. In the example of Figure 7 a selection on an attribute of table ofriends which is added on top of the output table ofriends could be pushed down while a selection on an attribute of table knows could not be pushed down.

Other rules: Other optimization rules exist, e.g. for pushing down two aggregation operators (with the same group-by attributes) over one common operator. Another rule exists for replicating one operator (and its sub-plan) that is consumed by multiple other operators. In Figure 6 this rule could be applied to the selection operator $\sigma_{\text{year}=?}$ and its sub-plan. In this case replicating this operator and its sub-plan would transform the graph into a tree-based execution plan.

Moreover, all rules shown before can be generalized from two operators that consume the same intermediate result to

Figure 8: Rules for selection-pushdown

Selection-Pushdown: Figure 8 shows multiple rules for pushing selection operators down in a graph-based execution plan. Rule 1 can be applied to push down the predicates $p$ and $q$ of two selection operators over a unary operator (i.e., projection and selection). In order to push down the two predicates, a new selection operator is created which uses a filter predicate $p \lor q$. In the following, we list some new rewrite-rules for graph-based execution plans.

Selection-Pushdown: Figure 8 shows multiple rules for pushing selection operators down in a graph-based execution plan. Rule 1 can be applied to push down the predicates $p$ and $q$ of two selection operators over a unary operator (i.e., projection and selection). In order to push down the two predicates, a new selection operator is created which uses a filter predicate $p \lor q$. In the following, we list some new rewrite-rules for graph-based execution plans.

Figure 7: Tail-recursive graph-based execution plan

applied to a graph-based execution plan. However, without any additional rewrite-rules, operators which are consumed by more than one operator as well as the operators for recursion act like barriers (since none of the traditional rewrite rules for tree-based execution plans can be applied). As an example, the filters of the selection operators in both upper sub-queries of Figure 6 (i.e., salF and sal0) could not be pushed down into the sub-plan for r1.

In the following, we list some new rewrite-rules for graph-based execution plans.
n-operators which consume the same intermediate result.

The result of the logical optimization of the plan in Figure 5 using these rules is shown in Figure 10.

![Figure 9: Rules for projection-pushdown](image)

![Figure 10: Optimized graph-based execution plan](image)

For cost-based optimization again the existing approaches for tree-based execution plans could also be applied to graph-based execution plans (e.g., selectivity estimation). However, one difference is that join re-ordering must make sure that intermediate results (which are consumed multiple times) must either be preserved or sub-plans must be replicated. Finding optimal approaches for the cost-based optimization of graph-based execution plans is an interesting avenue of future work.

4. PERFORMANCE EXPERIMENTS

For our preliminary performance experiments, we implemented a prototype in Java 1.6 which compiles and optimizes FunSQL functions that can be holistically optimized over an imperative stored-procedure. The code executed for this experiment was similar to the examples shown in Figure 1 and Figure 2 (adopted to the TPC-H database schema).

**Experiment 1:** The goal of this experiment was to show the benefits when using FunSQL functions that can be holistically optimized over an imperative stored-procedure. The code executed for this experiment was similar to the examples shown in Figure 1 and Figure 2 (adopted to the TPC-H database schema).

**Experiment 2:** The second experiment shows the effectiveness of the novel logical optimizations for graph-based execution plans. Figure 11 shows a sketch of the execution plan used for this experiment: it first reads a list of orders from Asia and then calculates a list of certain European and American suppliers for these orders. The boxes in this figure represent sub-plans with the given filter conditions. For the experiment the complete plan was executed once without applying any graph-based optimization rule and once with all optimization rules applied (which pushed down the filter conditions from the upper blocks `european_supp` and `american_supp` into the lower block `asian_orders`).

**Experiment 3:** The goal of the last experiment was to show the effectiveness of re-using intermediate results and thus being able to produce multiple outputs based on the same intermediate result instead of running multiple queries (i.e., one for each output). Therefore we used a similar plan as the one in Figure 11 and executed the plan first as a graph-based plan and second we split up the graph-based plan into two tree-based plans and executed the two plans separately.

Showing the effectiveness of the tail-recursive function call is one avenue of future work.

**Figure 11: Experiment 2 (sketch)**

Experiments executed the following three experiments:

*Experiment 1:* The goal of this experiment was to show the benefits when using FunSQL functions that can be holistically optimized over an imperative stored-procedure. The code executed for this experiment was similar to the examples shown in Figure 1 and Figure 2 (adopted to the TPC-H database schema).

*Experiment 2:* The second experiment shows the effectiveness of the novel logical optimizations for graph-based execution plans. Figure 11 shows a sketch of the execution plan used for this experiment: it first reads a list of orders from Asia and then calculates a list of certain European and American suppliers for these orders. The boxes in this figure represent sub-plans with the given filter conditions. For the experiment the complete plan was executed once without applying any graph-based optimization rule and once with all optimization rules applied (which pushed down the filter conditions from the upper blocks `european_supp` and `american_supp` into the lower block `asian_orders`).

*Experiment 3:* The goal of the last experiment was to show the effectiveness of re-using intermediate results and thus being able to produce multiple outputs based on the same intermediate result instead of running multiple queries (i.e., one for each output). Therefore we used a similar plan as the one in Figure 11 and executed the plan first as a graph-based plan and second we split up the graph-based plan into two tree-based plans and executed the two plans separately.

Showing the effectiveness of the tail-recursive function call is one avenue of future work.

The results of the experiments for a TPC-H database with scaling factor 1 are shown in Figure 12. The table shows different lines for the intermediate results and the final result (i.e., Σ) of each experiment (e.g., for experiment 2 the line `asian_orders` indicates the result for intermediate result with the same name in Figure 11). Each line in the table shows the intermediate result sizes and the execution time for the corresponding sub-plans (not optimized and optimized).

The performance gain in experiment 1 for the function implemented in FunSQL compared to the non-optimized imperative code is tremendous (since only one combined plan is executed instead of approx. 30k single plans). For the other two experiments, we can see a performance gain of factor 2. For experiment 2 the reason is that the intermediate result sizes were reduced by a factor of 2 (by the filter-pushdown
rules). For experiment 3, the intermediate results have same size (optimized and not-optimized). However, in the optimized case the intermediate result \textit{asia\_orders} only needs to be produced once whereas in the non-optimized case the intermediate result \textit{asia\_orders} needs to be generated twice.

5. RELATED WORK

From the language perspective there have been a lot of similar approaches in the last years in the area of cloud computing. Most of these approaches for processing large volumes of data raised in the context of map-reduce based frameworks [3] (or in the context of other similar approaches like Dryad [4]) for parallel execution. Based on these approaches different high-level declarative language have been built on top (as e.g., [2, 7, 5, 10] and many more) which are optimized and then compiled compiled to map-reduce programs for execution. Compared to FunSQL these approaches mainly focus on parallelization aspects and not analyze optimization rules for graph-based execution plans in detail. Moreover, a main difference is that these approaches do not discuss how recursion is integrated into graph-based execution plans and optimized efficiently.

From the relational database perspective, there has been some work on multi-query-optimization which tries to find out whether different SQL queries share common intermediate results which could be reused [8]. Compared to this approach in FunSQL common intermediate results are made explicit instead of implicitly deriving them from multiple queries.

Another relevant area from the relational database perspective are execution models for graph-based execution plans. [6] discusses different ideas for acyclic graph-based plans which can easily be adopted to graph-based plans with cycles that are produced by FunSQL.

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented a novel functional database language called FunSQL. FunSQL combines the best of two worlds: (1) it allows applications developers to decompose application logic in a step-wise manner as in an imperative language and (2) it can be efficiently optimized and parallelized as single SQL statements. In the paper we discussed the compilation of FunSQL functions into a graph-based execution plans as well as novel optimization rules for graph-based plans. Our performance experiments have shown the benefits of FunSQL over imperative languages and the effectiveness of the optimization rules.

One important avenue of future work is to analyze how FunSQL can be parallelized efficiently. We believe that many techniques from parallel databases can be applied here as well. However, we think that additional rules are needed to parallelize graph-based execution plans including recursion. Other avenues of future work are to support update statements in FunSQL functions as well as finding optimal approaches for the physical optimization of graph-based execution plans. Moreover, other functional language constructs (e.g., higher-level functions such as map and reduce, functional loops) should be integrated in future as well.

7. REFERENCES