DBSpinner: Making a Case for Iterative Processing in Databases

Sofoklis Floratos*, Ahmad Ghazal, Jason Sun†,
Jianjun Chen†, Xiaodong Zhang*
*The Ohio State University, Columbus, Ohio, USA
†ByteDance US Lab

2021 IEEE 37th International Conference on Data Engineering (ICDE) | 978-1-7281-9184-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDE51399.2021.00273

***Abstract*—Relational database management systems (RDBMS) have limited iterative processing support. Recursive queries were added to ANSI SQL, however, their semantics do not allow aggregation functions, which disqualifies their use for several applications, such as PageRank and shortest path computations. Recently, another SQL extension, iterative Common Table Expressions (CTEs), is proposed to enable users to perform general iterative computations on RDBMSs.**

In this work[1], we demonstrate how iterative CTEs can be efficiently incorporated into a production RDBMS without major intrusion to the system. We have prototyped our approach on Futurewei's MPPDB, a shared nothing relational parallel database engine. The implementation is based on a functional rewrite that translates iterative CTEs to other existing SQL operators. Thus, query plans of iterative CTEs can be optimized and executed by the engine with minimal modification to the code base. We have also applied several optimizations specifically for iterative CTEs to i) minimize data movement, ii) reuse results that remain constant and iii) push down predicates to avoid unnecessary data processing. We verified our implementation through extensive experimental evaluation using real world datasets and queries. The results show the feasibility of the rewrite approach and the effectiveness of the optimizations, which improve performance by an order of magnitude in some cases.

I. Introduction

The vast majority of Relational Database Management Systems (RDBMSs) use the Structured Query Language (SQL) [1], which defines what data need to be retrieved and hides how data is processed. However, SQL is limited in expressing iterative computations. Examples of such iterative processing are the single source shortest path (SSSP) query that computes the shortest path between two points and the PageRank (PR) query that finds the most important nodes in a graph.

The current SQL standard supports recursive processing [2] through views or CTEs but it is limited to recursive union computations. This limitation is due to the assumption that a recursive query needs to reach a fixed point. Fixed point semantics in the context of SQL means that each iteration in recursion uses the previous iteration as input and stops at a fixed point where the input is the same as the output. This occurs when both input and output are empty. Thus, recursive queries cannot be used to express general purpose iterative computations like the PR query. Aggregate functions are not allowed in the recursive part of the query, the termination condition is implied and tuples can only be appended to the result and not updated.

Many users that need to execute iterative queries, usually switch from RDBMSs to specialized graph processing engines [3], [4], [5], [6], [7] that support custom vertex-based APIs [8] or Datalog [9] systems that can optimize recursive queries [10], [11], [12], [13], [14], [15] more effectively. Most of these solutions offer high performance, a critical aspect for many applications. However in our work, we explore the alternative scenario in which users have already stored their data into a RDBMS and want to avoid transferring them to a new engine, a step that can be very expensive or infeasible. Moreover, some users may prefer to use SQL as their preferred API, instead of learning a new one, or they may want to use the result of an iterative query directly as an input to another SQL query. In other words, our solution tries to accommodate users that prefer exchanging valuable productive time spent on integrating a new system into their workflow for performance, a requirement that came directly from multiple clients. Thus, we do not propose to replace or compare a SQL-based approach with any of these solutions but instead we explore an efficient way to enable relational SQL-based systems to run iterative queries.

Despite the fact that SQL is not a popular option for iterative processing yet, mainly due to the limitations imposed by the fixed point semantics, recent efforts [16], [17] have explored the possibility of extending recursive CTEs and proposed new SQL structures and operators that can accommodate iterative queries on relational data. Note that these SQL extensions are complementary to recursive queries and address use cases like the PR query. Recursive queries can still be used for hierarchical iteration like bill of materials.

The authors of [17] propose an implementation that creates stored procedures and executes them in the RDBMSs, while the authors of [16] create a middleware between the user and a target database engine. Both proposed frameworks, involve solutions that are implemented outside the system. We illustrate the logic of [16] using the PR query applied on a table that captures the connections between web pages (source page, destination page and other additional information). The computation starts with scanning the connection table and computes an initial rank for all pages. The page rank information is treated as a working table materialized at the

[1]This work was done while Ahmad Ghazal, Sofoklis Floratos, Jason Sun and Jianjun Chen were working with Futurewei.

978-1-7281-9184-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00273

target DBMS. It also gets updated by examining paths between pages through neighbours. Exploring such paths is a sequence of self joins and updating the rank is done through an update DML statement. In summary, the external solution in [16] interacts with the DBMS through the following operations: create and insert into temporary tables, issue self join queries, update and finally drop these temporary tables.

Although the external approach is flexible, as the user can choose the database engine of his/her preference and avoid data transforming and loading steps, it has some limitations. First, it is hard for external solutions to maintain ACID properties for long query executions. In addition, the workload manager is unaware of the iterative query as a whole and treats each of the basic operations as a query by itself. Furthermore, the basic operations sent to the DBMS are more heavy than needed. For example, intermediate results are explicitly defined as temporary tables which impose metadata overhead. The DML used in the external approach to initialize and update intermediate results is also an overkill, since it implies locking and other transaction overhead related to these operations. Finally, these external solutions do not have system specific optimizations, leaving space for performance improvement.

We propose a solution that addresses all the above limitations imposed by external implementations. Our approach extends mainly the planner and optimizer in order to support iterative CTEs natively. Other components, such as the parser and the database kernel, require some minor modifications that are also described. However, the overall solution is based mainly in a functional re-write that takes place inside the planner, while reusing existing structures and resources of the system. Our goal is to support iterative CTEs in a non-invasive way. The approach is prototyped on Futurewei's MPPDB database [18] (MPPDB for the rest of the paper), a scalable commercial relational database. We choose MPPDB, an OLAP solution, instead of a traditional RDBMS as iterative queries are mainly used in an analytical environment.

The MPPDB parser extension covers the new grammar for iterative CTEs and produces a parse tree similar to regular and recursive CTEs. The key implementation relies on converting the iterative CTE specification to a logical query tree inside the planner. This conversion is done through a new rewrite rule that converts the iterative CTE to existing DBMS operators like scans, joins and aggregations. The update logic mentioned before can be done through a scan from a temporary table to another temporary table. For the iteration logic, we have added a new simple operator that allows conditional redirection to a previous step in the execution plan. The proposed solution basically creates a single plan for queries involving iterative CTEs in order to solve all the problems discussed above.

The optimizer simply works for the rewritten iterative query. No changes are needed for cost based optimizations or the cost subsystems to handle iterative CTEs. We modified two rule based rewrites for iterative CTEs: predicates push down and common results rewrite. Pushing predicates cannot be done in a similar way as regular CTEs and need to be modified. In addition, there are more opportunities to share intermediate results among different parts of a query plan. If there are joins between tables in the iterative part, then in some cases, they can be materialized once, at the beginning of the query, and reused multiple times. We evaluate the proposed methodology by using real data and iterative queries such as PR, SSSP and a query that Forecasts the number of Friends (FF) through a geometric sequence. We test the efficiency of our functional rewrite and explore extensively when our proposed optimizations can accelerate the query execution. We observe an order of magnitude faster execution for optimized iterative queries in some cases.

We would like to re-iterate a few key differences between our approach and the procedural one. First, all the limitations mentioned for external solutions, also apply to stored procedures and UDFs. In general, procedural solutions are used not as an alternative to SQL but rather when direct SQL is not possible. Stored procedures and UDFs need to be written as a custom made solution to an iterative problem. The DBMS optimizer treats the UDF as a black box and processes each statement of the stored procedure in isolation. This leads to higher cost of application development and lower performance than a declarative SQL solution.

In summary, our contribution is to provide an effective implementation and necessary system optimizations for iterative CTEs in commercial RDBMSs through non-invasive extensions to their relevant components inside the engine. The extensions introduce two new execution operators: *rename* and *loop*. The optimizer extensions focus on adjusting existing optimizations, such as predicate push down, and extracting common results to work for iterative CTEs.

The rest of the paper is organized as follows: Section II contains background information on iterative CTEs and Section III describes how they can be implemented natively. Section IV presents the functional rewrite and Section V describes the optimizations that can be applied. Section VI discusses the extensions that need to be done in the execution engine. Section VII presents the experimental studies. Section VIII discusses related work and finally, Section IX concludes.

II. Background

This section describes the original SQL extension proposed in [16] and discusses the drawbacks of a middleware approach to motivate our work. We use the PR query as an example for iterative CTEs throughout the paper. The rank computation itself was extensively discussed in [19] and expressed using iterative CTEs in [16]. To describe the computation, we assume that all edges of the graph are stored in a relation named *edges* that has three attributes, *src*, *dst* and *weight*. Each tuple in that relation can be mapped to a graph as an edge that goes from node *src* to node *dst* and has an assigned number *weight*. We also denote the resulted rows of iteration i as R_i.

Recursive views and CTEs were the first efforts to support iterative processing and became part of the SQL ANSI [2]. Recursive queries cannot be used for iterative processing that requires aggregation due to the fixed point semantics. Iterative CTEs allow a termination condition explicitly defined by the

```
1  --Create tables
2  CREATE TABLE IntermediateTable (node int,
3                     rank float, delta float);
4  CREATE TABLE PageRank (node int,
5                     rank float,delta float);
6
7  --Non iterative Query
8  INSERT INTO PageRank
9       SELECT src, 0, 0.15
10       FROM (SELECT src FROM edges
11                  UNION SELECT dst FROM edges);
12
13  --Iteration 1
14  DELETE FROM IntermediateTable;
15
16  INSERT INTO IntermediateTable
17       SELECT PageRank.node,
18              PageRank.rank + PageRank.delta,
19              0.85 * SUM(IncomingRank.delta
20                         * IncomingEdges.Weight)
21       FROM PageRank
22        LEFT JOIN edges AS IncomingEdges
23         ON PageRank.node = IncomingEdges.dst
24        LEFT JOIN PageRank AS IncomingRank
25         ON IncomingRank.node = IncomingEdges.src
26       GROUP BY PageRank.node,
27                PageRank.rank + PageRank.delta;
28
29  UPDATE PageRank
30     SET rank = IntermediateTable.rank,
31         delta = IntermediateTable.delta
32    FROM IntermediateTable
33   WHERE PageRank.node = IntermediateTable.node;
34
35  --Iteration 2
36  ...
```

Fig. 1: PR Query using multiple SQL statements

user. Thus, they can be used to support more generic iterative computations as aggregations are permitted in the iterative part of the query. Before we describe how iterative CTEs work, let us consider how PR can be implemented through a custom made SQL-based application as shown in Figure 1.

The PR query in Figure 1 is expressed by multiple SQL statements. Lines 1 to 5 create the main and working tables. Lines 7-11 execute the non-iterative part and Lines 14 to 34 execute the iterative part of the query once. Then, the iterative part needs to be executed again for N number of iterations by copying Lines 14 to 34 multiple times. Part of the iterative computation is to update the page rank result. This solution works if N is known which is not always the case.

The SQL extension in [16] proposes iterative CTEs to address the problems mentioned above by allowing the user to describe the iterative computation and explicitly define its termination condition. An iterative CTE named R contains a non-iterative part R_0, and an iterative one R_i. As in recursive CTEs, R_0 is executed only once, while R_i is executed multiple times. A major difference between the two SQL structures is that iterative CTEs update the working table, instead of adding new tuples. Moreover, the query terminates when the termination condition T^c is satisfied. This does not only give the user the flexibility to define explicit termination conditions, but also removes the assumptions for fixed point semantics. Thus, aggregate functions can be used in R_i. The termination condition according to [16] can be based on data, metadata and delta values. When T^c is based on data, then the system checks if the regular SQL expression given by the user is satisfied by the main CTE table. If T^c is based on metadata, then the system checks the number of iterations or the number of updates performed. Finally, if T^c is based on delta values, then the system compares the latest version of the dataset with the previous one. Further details regarding the syntax and semantics of T^c can be found in [16]. The general form of an iterative CTE is:

WITH ITERATIVE R **AS (**R_0 **ITERATE** R_i **UNTIL** T^c**)** Q_f

The SQL in Figure 2 is an example of an iterative CTE that computes PR for all pages. At the beginning, the system executes R_0 and stores the result into the main CTE table. Then, for each iteration, it needs to i) execute R_i and store the result in the working table, ii) update the main CTE table with the rows that exist in the working table and, iii) check if T^c is satisfied in order to determine if another iteration is needed. To update the main table correctly, a unique row key/identifier needs to exist. In SQL, each tuple is considered part of a set and thus, without a unique identifier, updated values cannot be mapped to the original ones. If a primary key column is specified by the user in the schema of the involved tables, the system uses it to perform the update, otherwise it creates unique row IDs. Moreover, the system throws a run-time error if the user defines a CTE in which the iterative part results into a working table with duplicates of a single row/key. This is necessary, as there will be two (or more) updates for the same row in the main table and thus, the system will not know how to handle it. In this case, the user needs to redefine the iterative part and explicitly specify how duplicates need to be resolved through an aggregation, group by etc. After the completion of the iterative part, the system performs the query Q_f and returns the final result to the user.

```
1  WITH ITERATIVE PageRank(Node, Rank, Delta)
2  AS ( SELECT src, 0, 0.15
3       FROM (SELECT src FROM edges
4                  UNION SELECT dst FROM edges)
5   ITERATE
6    SELECT PageRank.node,
7      PageRank.rank + PageRank.delta,
8      0.85 * SUM(IncomingRank.delta
9                  * IncomingEdges.Weight)
10   FROM PageRank
11     LEFT JOIN edges AS IncomingEdges
12       ON PageRank.node = IncomingEdges.dst
13     LEFT JOIN PageRank AS IncomingRank
14       ON IncomingRank.node = IncomingEdges.src
15   GROUP BY PageRank.node,
16            PageRank.rank + PageRank.delta
17   UNTIL 10 ITERATIONS )
18  SELECT Node, Rank FROM PageRank;
```

Fig. 2: PR Query using iterative CTEs

Iterative CTEs in [16] are implemented as part of a middleware system that eliminates the need for custom SQL scripts as the one shown above. This approach, decouples the

SQL extension from the execution engine, providing a system independent solution. However, as other middleware solutions, this comes with several limitations. First, the query is executed as multiple `INSERT`, `SELECT` and `UPDATE` statements in a loop. This makes the query processing complicated as it is hard for the target engine to process failures and aborts. The entire query execution can be enclosed into a long transaction but it is not efficient and requires more intimate logic of the underlying DBMS transaction support and semantics. Second, the DBMS workload manager (if any) has no knowledge about the iterative CTE query and processes each of the basic operations one by one. Thus, scheduling and resource management is done at per statement basis and not for the entire iterative query. Third, the basic operations sent to the DBMS cause overhead. For example, the iterative results are maintained using creation, updates and dropping of temporary tables. The temporary table creation and dropping operations are DDL with metadata that pose overhead and extra locks. The DML used to initialize and update intermediate results requires locks and transaction management which adds to the overhead and complexity of the solution. Finally, the middleware approach may not benefit from certain query optimizations that can be applied only for complete query plans. Examples are push down predicates or optimizations that find common parts and benefit from them. Iterative CTEs have not been part of RDBMSs before and thus, there is no prior implementation known for effectiveness or performance. All these issues motivated us to investigate an effective approach that integrates iterative CTEs into the system and turns them into a reality in a production environment.

III. Naive implementation

In this section, we describe how iterative CTEs can be implemented within a RDBMS and overcome the limitations introduced by external solutions. Our testbed is Futurewei's MPPDB. For every SQL query that involves one or more iterative CTEs, our implementation in MPPDB, constructs a single query plan that eventually gets executed by the engine. This is similar to how regular or recursive CTE queries are processed by the database system. We reuse regular CTE structures to create, process and update temporary results in order to avoid the need for explicit DDL and DML operations like the ones used by external solutions. Finally, a new simple operator is needed to handle the loop of iterative CTEs, something that we further discuss in Section VI.

Before providing further details, we highlight the advantages of supporting a native solution that uses a single query plan. First, it ensures that ACID proprieties will be handled by the system without the need to create long transactions. Second, the workload manager can schedule iterative CTEs in the same way as other native database operations and queries, as a single execution plan can be examined as one processing unit. Third, unnecessary overhead introduced by DDL and DML operations is avoided. Finally, existing query optimizations and cost estimation can be applied by the planner to the entire iterative CTE and not only to individual SQL statements that are part of a bigger query.

TABLE I: Logical Plan of the PR query in MPPDB

Step ID	Description
1	Materialize *PageRank* with the results of the union operation between the selection of *src* and *dst* from table *edges*.
2	Initialize counter to zero.
3	Materialize *Intermediate_Results* with the results of the join between *PageRank* with *edges* and then with the results of self-join with *PageRank* using `GROUP BY` in attribute *nodes*.
4	Rename *Intermediate_Results* to *PageRank*.
5	Increment counter by 1.
6	Go to step 3 if counter < 10.

Our implementation inside MPPDB, applies the same execution flow as the one used by external solutions but without DDL and DML operations. The insert logic is implemented through the materialization of intermediate results which is a common execution operator in many DBMSs and used to store results of intermediate join and aggregate operations. Updates can also be implemented through another materialization with the proper selection of old and new values. To demonstrate our approach, we use the PR query as an example. Table I illustrates the abstract description of the logical plan which is used by the planner in order to produce the actual physical plan, similar to what happens with other more traditional SQL structures. Step 1, inserts the non-iterative part into a temporary result called *PageRank* by using a regular `WITH` SQL object. Step 3, executes the iterative part which computes the page rank using ranks from neighbors and connections from the $edges$ table. We want to highlight that *PageRank* is both input and output. Thus, we use another *intermediate result* to hold the values. If the query updates the entire dataset then, the system just renames the *intermediate result* to *PageRank* to avoid finding the updates and making unnecessary data movement in step 4. The *rename* is a new and simple step that got added to MPPDB. Steps 2 and 5 are also new additions to the execution engine to handle the loop and require support by other parts of the system as well.

Supporting general iterative processing in MPPDB involves extensions to the parser, rewrite subsystem, planner and execution engine. The parser handles the syntax, semantics and access rights for the SQL query. It also outputs a logical query tree (parse tree format) and passes it over to the rewrite subsystem. This part of the system is broken down into functional and optimization rewrites. Functional rewrites transform some operators not supported by the execution engine to other low level operators. Common examples are view reference expansion (plugging view definitions into the query tree) and transforming complex OLAP functions like `CUBE` or `ROLLUP` to a `UNION` of simple aggregate queries. Teradata [20] processes recursive and temporal queries [21], [22] through functional rewrites as well. Optimization rewrites (rule and cost based) get the logical plan from the functional rewrites and produce an optimized query plan tree. Examples

of optimization rewrites are predicate push down, UNION simplification, join elimination, etc. The join and aggregate planner converts the logical tree from the rewrite subsystem to a physical tree that includes join ordering and implementation, aggregation methods and data shuffle decisions. Then, the execution plan is further optimized through LLVM code generation and passed to the execution engine. During the final step of query processing the executor performs the actual query execution and returns the results to the user. To enable MPPDB to process iterative CTEs, we extended most of the components inside the system but without introducing major alterations to the code. The changes are summarized below:

- **Parser changes:** Most of the changes in the parser have been done to accommodate the new syntax introduced by iterative CTEs and to produce a new parse tree node that contains crucial information for the execution of the query such as the non-iterative part, the iterative part and information regarding the termination condition. The details of the parser extension are straightforward and thus, we will not give any further details.
- **New functional rewrite rule:** Inside the rewrite subsystem, we added a new rewrite rule to transform iterative CTEs into lower level operations similar to the PR example illustrated in Table I. Section IV explains the logic of this new rewrite rule.
- **New optimization rewrite rule:** MPPDB has numerous rules and cost based optimizations which can be applied without further changes to the logical query tree produced by the functional rewrite component. However, two of them require changes related to iterative CTEs, such as common result rewrite and predicate push down. Section V describes these changes.
- **Planner changes:** The join and aggregation planners do not need further changes to process the produced logical tree. However, a minor modification is required to recognize and pass to the physical plan, two new execution operators that accommodate *loop* and *rename* functionality. Again, the changes and implementation in this component are straightforward and thus, we will not provide further details.
- **Execution engine changes:** The execution engine is extended to cover two new simple operators: *loop* and *rename*. Nothing else is needed since the rewritten query plan is based on existing DBMS operators. This discussion is covered in Section VI.

IV. Core Algorithm

The core implementation of iterative CTEs in MPPDB is done through a functional rewrite that expands the parse tree into a sequence of SQL operators that cover the non-iterative part, the iterative part and the loop logic that allows multiple iterations. We use PR as an example to illustrate the mechanics of the functional rewrite and provide the generic algorithm used inside the planner.

Figure 3 depicts the tree of the PR query after being processed by the parser. The figure does not show the full detailed tree but focuses on the main nodes that cover the operators. The root is a selection from the iterative CTE *Pagerank*, which is an extension of the existing CTE node. The CTE node is marked as "iterative" to differentiate it from regular and recursive CTEs. The non-iterative part is a UNION operation extracting the *src* and *dst* from the *edges* table. The iterative part is a GROUP BY on top of two left outer joins involving the main CTE result and *edges*. Another child of the CTE describes the termination condition based on "UNTIL 10 ITERATIONS" which is of type "Metadata".

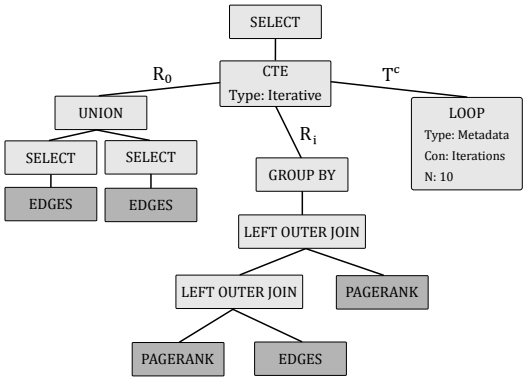


Fig. 3: PR Query Parse Tree in MPPDB

The functional rewrite expands the iterative CTE into a sequence of regular SQL operations that perform the needed computation. Figure 4 illustrates the result of the rewrite which is a tree that represents the logical query plan generated by MPPDB in the planner after processing the parse tree illustrated in Figure 3. The rewrite process is similar to other functional rewrites that take place in the planner, like view expansion or complex OLAP functions. In this case, it generates a sequence of operations that compute the non-iterative and iterative part. The first *materialize* node computes R_0 which can be any SELECT statement. The iterative part is another *materialize* step that has a GROUP BY applied on top of the two outer joins required by R_i and explained before. Finally, the last step is a *rename* operation that puts the intermediate results back into the main *PageRank* result. In queries that do not update the entire dataset, the planner adds before the rename, a SELECT that selects the old values from *Pagerank* and the new ones from *intermediate result* using the column that serves as a unique row identifier. Thus, in the next iteration, the main CTE relation will contain both updated and non-updated values. The PR query in Figure 2 updates the entire dataset as there is no WHERE clause in R_i (i.e. Lines 6-16). For this reason, Lines 5 and 6 are executed in Algorithm 1. In the SSSP query, shown in Figure 7, there is a WHERE clause in R_i (i.e. Lines 7-17) that updates only the explored nodes and not the entire dataset, thus Lines 8,9 and 10 from Algorithm 1 are executed. Note that the logical plan shown in Figure 4 is a simplified version of the original one, used for demonstration purposes and that the generic algorithm used by the planner of MPPDB can be found in Algorithm 1.

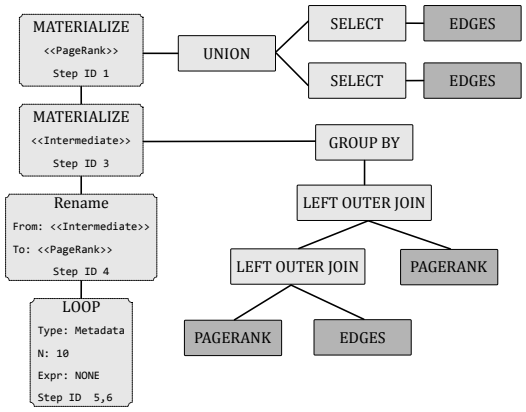


Fig. 4: PR Logical Tree in MPPDB

A crucial element in the execution of an iterative CTE is to determine when the termination condition is satisfied. As mentioned before, the termination condition can be based on data, metadata or delta values. In MPPDB this part is implemented by the *loop* operator in the query plan tree. The implementation of this part in the functional rewrite depends on the type of termination condition which is identified by the parser variable *Type* (as illustrated in Figure 3) and is passed to the planner through the new iterative CTE object. A simple new operator called *loop* which handles the conditional execution flow is added. The functional re-write checks if the parse tree is an iterative CTE and adds the *loop* operator in the logical plan tree. Then, it fills in all the needed information required by each different type of termination condition from the parse tree. The new operator captures three pieces of information: 1) type of the termination condition (Metadata, Data or Delta), 2) number of iterations or updates with an indicator to distinguish between the two options and 3) the SQL expression used for the data and delta termination conditions along with an extra indicator that accommodates the `ANY` keyword (further described in [16]). For the PR example, the query requires explicitly 10 iterations, thus the termination condition is based on metadata. The *loop* operator in the logical query plan tree is populated with `<<Type:metadata, N:10, Expr:NONE>>` as illustrated in Figure 4.

Finally, the planner initializes the *loop* operator right after the execution of the non-iterative part and updates it at the end of each iteration. For the simple case of the PR query, MPPDB starts a new counter (step ID 2 in Table I and Line 2 in Algorithm 1) before the execution of R_i and then increases the counter (step ID 5 in Table I and Line 11 in algorithm 1) before checking if another iteration is needed (step ID 6 in Table I and Lines 12-13 in algorithm 1).

V. Optimization Techniques

In the previous section we described how the planner performs a functional rewrite and transforms the iterative CTE into known SQL operations. During that process, it tries to minimize unnecessary data movement by eliminating the need to transfer data from the intermediate result back to the main one when possible. We further discuss the optimization of iterative CTEs in this section and explore elimination of unnecessary data movement in Section VII.

Algorithm 1: Functional Rewrite of an iterative CTE

```
1  materialize R_0 into cteTable;
2  initialize loop operator;
3  materialize R_i into workingTable;
4  if R_i does not have a WHERE clause then
5  |   rename workingTable to cteTable ;
6  |   recreate workingTable ;
7  else
8  |   mergeTable as ( select case
   |     when cteTable.col1 !=
   |     workingTable.col1
   |     then workingTable.col1
   |     else cteTable.col1
   |     end,
   |     ...
   |     from cteTable left join workingTable
   |     where cteTable.key = workingTable.key;
9  |   rename mergeTable to cteTable;
10 |   delete tuples from workingTable;
11 end
12 update loop with data from current iteration;
13 if conditional execution in loop returns true then
14 |   continue from Line 3
15 return Q_f;
```

The optimizer treats iterative CTE queries like any other regular SQL statement. It applies heuristic optimization rewrites like join elimination, outer to inner join conversions, etc. It also applies cost based rewrites, join re-ordering and aggregate planning as usual. No changes are needed for cost based optimizer or the cost subsystem (statistics, cost formulas, .. etc). However, two rule based optimizations required special considerations for iterative queries. First, predicate push down need to be restricted. A filter on an iterative reference cannot be applied blindly during the execution of the iterative part as it might eliminate relevant intermediate results. Second, joins in the iterative part that produce the same result in each iteration can be avoided and with them, redundant computations as well. The next two sections describe the changes to the optimizer for these two special cases.

A. Common Result Optimization

Common result rewrite is an optimization performed by RDBMSs that identifies, materializes and reuses blocks of a query that are the same. In this way, the system avoids recomputing identical parts of the execution plan more than one time. To illustrate how MPPDB materializes constant parts in iterative CTEs, we consider a modified version of the PR query that calculates the rank of only the active nodes in the graph. For this example, we add table `VertexStatus` that contains the availability of a node in the form of a tuple `<node, status>` (i.e. nodes that are unavailable in the `vertexStatus` will not be considered in the rank calculation). The modified query is called PR-VS and it the same

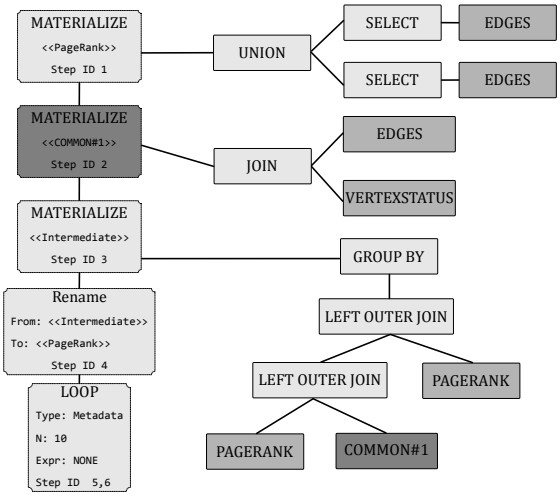


Fig. 5: Logical plan that materializes common results

as the PR query with adding a join with `vertexStatus` in the iterative part as :

```
      JOIN vertexStatus AS avail_pr
        ON avail_pr.node = IncomingEdges.dst
    WHERE avail_pr.status != 0
```

The rewrite in general, should be cost-based since it is not always more efficient to apply this optimization. First, reusing common results mandates materializing them, which can be expensive in terms of memory consumption due to the lack of pipe-lining. Second, this optimization might impact other ones, such as predicate push down. For example, if the common result originates from a query block that is a view/in-line view, then it might not be possible to apply predicate push down, since it will make the result not common anymore. Implementing common result optimization as a cost-based rewrite, is a complex problem and it is outside the scope of this paper. Some systems, like Teradata, implemented this optimization as a post join planning heuristic.

Common result rewrite is crucial for iterative CTEs since the iterative part may include repetitive joins for tables other than the iterative reference. Such joins, can be materialized once outside the loop and used throughout the iterative part. We extended MPPDB to include this optimization as a heuristic rewrite applied on the logical query tree (i.e. rule-based rewrites). We decided to implement this optimization as a heuristic and not as a cost-based optimization for two reasons. First, iterative CTEs mostly materialize intermediate results as illustrated in Section IV and thus, materialization of the common part does not add much overhead. Second and most importantly, the system avoids recomputing the same results not only once but for as many times as the number of iterations performed. Thus, the benefit of this optimization highly outweighs other possible drawbacks or techniques.

The logical query plan generated in the planner by the above query is similar to the original PR query in Figure 4. The main difference is that the iterative part (step ID 3 in Table I) has three joins, involving a self join, a join with `edges` and another join with `vertexStatus`. The common rewrite optimization materializes the join between `edges` and `vertexStatus` prior to the execution of the iterative part (i.e. `COMMON#1` in Figure 5). Then, the iterative logical query fragment is rewritten using this intermediate result, as depicted in Figure 5. The common result is computed in step 2 and reused multiple times in step 3. For the general case of identifying common parts in the logical tree of the iterative part, the system needs to reorder the joins as, for example, vertexStatus may not be joined directly with edges. However, join reordering is straightforward for inner joins but complex for outer joins [23] and thus, this is something that we will explore in future work.

B. Predicate Push Down

Predicate push down is a key rewrite rule in RDBMSs. The rewrite covers a few different cases where a predicate is pushed within or across query blocks (a query block corresponds to a select or sub-select statement). MPPDB has a wide coverage of predicate push down, including cases of similar rewrites that pull up or move around predicates.

For iterative CTEs, predicate push down across blocks (from the parent into the iterative CTE block) cannot be applied like other regular SQL statements. For example, if the main query in Figure 2 has the filter "`SELECT Node, Rank FROM PageRank WHERE Node = 10`" that restricts the results to only node 10, then for regular CTEs, the system will push the predicate "`Node = 10`" into the CTE. However in the PR query, such a predicate push down leads to incorrect result, since nodes other than 10 (i.e. neighbors) are needed for the correct computation of the rank during the iterative part. Note that there are more opportunities for predicate push down for recursive queries (see [20], [24]) since results are built incrementally through the recursive `UNION`.

```
WITH ITERATIVE forecast (node,friends,friendsPrev)
AS( SELECT src AS node, count (dst) AS friends,
       ceiling (count(dst) *
            (1.0-(src%10)/100.0)) AS friendsPrev
    FROM edges GROUP BY src
 ITERATE
    SELECT node AS node,
       round(cast((friends / friendsPrev)
          * friends AS numeric),5) AS friends,
       friends AS friendsPrev
    FROM forecast
 UNTIL 5 Iterations )
SELECT node, friends
FROM forecast WHERE MOD(node,100) = 0
ORDER BY friends DESC LIMIT 10;
```

Fig. 6: FF query

An example of an iterative CTE where a predicate push down can be applied, is the one in Figure 6. The query forecasts the number of friends (FF) as a growth factor of the previous year. The main query block picks one percent of the result as a sample using the predicate "`MOD(node,100) = 0`". Given that the query has no self join or aggregate

functions on `forecast`, MPPDB is able to push the main query predicate `MOD(node,100) = 0` to the non-iterative part (i.e. from Line 14 to Line 5) eliminating in this way a lot of unnecessary processing. This optimization is quite efficient when it can be applied, as depending on the selectivity of the pushed predicate, it can improve the performance by orders of magnitude. We further explore the properties of this optimization in Section VII.

VI. Execution Engine Changes

As mentioned in Section III, the plan generated by the planner contains already known operations to the engine in order to avoid an expensive implementation and most importantly, to enable the system to reuse existing optimization techniques. However, we still need to add two new simple operators, *rename* and *loop*, to enable iterative processing and to ensure an efficient execution. This section describes the implementation of the two operators inside the executor.

A. Rename Operator

In Section IV, we discussed how the *rename* operator is used to rename a reference to a temporary result. The execution engine has a lookup table that manages intermediate results in memory and consists of two columns. The first column has the name and the second one is a structure that has the schema along with a pointer to the memory location used to store the intermediate results. The *rename* operator looks up the old name and updates it with the new value. If the new name already exists (already points to some existing intermediate results) then MPPDB simply removes that entry and releases the memory associated with it.

B. Loop Operator

The other new operator is called *loop* and as described in Section IV, is used to explicitly handle the conditional execution flow introduced by iterative CTEs and their termination logic. This new operator needs to simply check a single variable (we named it *continue*), and point to the next SQL operator. For our case, if the value of this variable is true, it will point to the next iteration, otherwise, it will point to the end of the iterative part. The implementation for this part of the *loop* operator is simple, as it requires only two execution node pointers and an if statement. The termination condition captured in the *continue* variable is implemented for the three cases described before as follows:

- **Metadata**: If the query needs to terminate after N iterations or updates then the system uses a counter that gets updated during the execution of the iterative part, based on number of iterations or number of updated rows. If at the end of an iteration, the counter satisfies N then the *continue* variable becomes false.
- **Data**: When the termination condition is based on a SQL expression, then the operation that is equivalent to the query "`SELECT count(*) FROM cteTable WHERE expr;`" is executed by the system in order to count how many rows in `cteTable` satisfy `expr`, the SQL expression defined by the user. *Continue* becomes false, if the count exceeds N in the query.
- **Delta**: If the termination condition is based on delta values, then similar logic as before is used to check how many rows get updated in the current iteration relative to the previous one. For this case, we also keep data from the previous iteration.

VII. Experiments

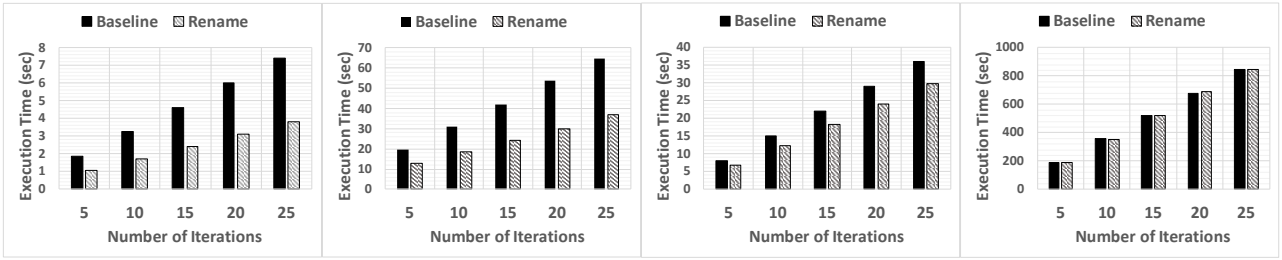
We evaluated our rewrite approach by running different iterative queries that vary in terms of complexity and used datasets from [25] with different sizes. We tested the effectiveness of the proposed optimizations and found that minimizing data movement in each iteration can decrease execution time by 48%, materializing parts that remain constant for all iterations can improve the performance by 21% and pushing down predicates to avoid unnecessary data processing when possible, can improve the performance by least an order of magnitude. For a very small number of iterations, the non-iterative part has a higher weight and as such, optimized and non-optimized CTEs have similar performance. As we increase the number of iterations the effectiveness of the applied optimizations depends on the query and more specifically on R_i. We discuss how different properties of the queries that we have tested impact the effectiveness of the proposed optimizations. Finally, we also compare optimized iterative CTEs with stored procedures that perform the equivalent computation and observed that CTEs can be executed at least 25% faster than stored procedures and in cases where predicates can be evaluated early, the difference can be much higher.

```
1  WITH ITERATIVE sssp (Node, Distance, Delta)
2  AS (SELECT src, 9999999, CASE WHEN src = 1
3           THEN 0 ELSE 9999999 END
4  FROM (SELECT src FROM edges
5        UNION SELECT dst FROM edges)
6   ITERATE
7    SELECT sssp.node,
8      LEAST (sssp.distance, sssp.delta),
9      COALESCE(MIN(IncomingDistance.delta
10          +IncomingEdges.weight), 9999999)
11   FROM sssp
12    LEFT JOIN edges AS IncomingEdges ON
13     sssp.node = IncomingEdges.dst
14    LEFT JOIN sssp AS IncomingDistance ON
15     IncomingDistance.node=IncomingEdges.src
16   WHERE IncomingDistance.Delta != 9999999
17   GROUP BY sssp.node,
18       LEAST (sssp.distance, sssp.delta)
19  UNTIL 10 ITERATIONS)
20 SELECT Distance FROM sssp WHERE Node = 10;
```

Fig. 7: SSSP Query using iterative CTEs

A. System Setup

During our experiments, we report the results using a server with Intel Xeon CPU E5-2680 v4 and 512GB of RAM. The operating system running on the machine is CentOS 7.8. To evaluate our system, we chose different iterative queries. The PageRank queries: PR as in figure 2 and its modified version PR-VS, the Single Source Shortest Path (SSSP) (which can be



FF query with DBLP Dataset | FF query with Pokec Dataset | PR query with DBLP Dataset | PR query with Pokec Dataset

Fig. 8: Using MPPDB with the *rename* operator

found in Figure 7 and is also published at [16]) and a query that Forecasts the number of Friends (FF) through a geometric sequence (Figure 6). Each one of these queries have different properties. The PR query processes the entire dataset in each iteration, while the SSSP query only nodes that are connected to the source. The FF query contains a very inexpensive iterative part as there are no joins or aggregations and the selectivity of the final part can be controlled by changing the value of "X" in "`MOD(node, X) != 0`" from 10 to 100 etc. For all queries, we used a termination condition based on metadata that explicitly defines the number of iterations. Finally, we used the DBLP dataset [26] that has 1,049,866 rows, the Google web dataset [27] that has 5,105,039 rows and the Pokec dataset [28] that has 30,622,564 rows.

B. Minimizing Data Movement

In this experiment, we explore how minimizing data movement in each iteration and avoiding synchronization between the main and working tables could result in faster execution times. Although this is not an optimization that was explicitly described in Section V, it was mentioned in Section IV as part of the *rename* operator. We think that it is important to demonstrate its necessity and discuss its properties.

Using the current methodology described in Section IV, queries that update the entire dataset, use only the *rename* operation. The fact that the new version of the dataset will completely overwrite the old one, makes the process of trying to find which rows have been updated by the last iteration redundant. Moreover, instead of transferring data to the temporary table and then back to the main one, we can use in the next iteration, the data that already exist in the temporary table, eliminating in that way unnecessary data movement. In our prototype, this is accomplished by renaming the temporary table to the main one. In this experiment, we compare the baseline execution that moves data from the intermediate table back to the main one (instead of using the *rename* operator) and also tries to identify updated rows even in queries that update entire datasets.

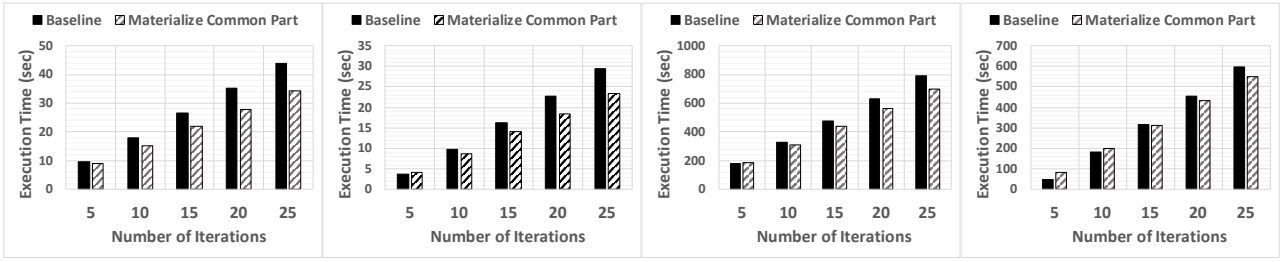
As can be seen in Figure 8, by minimizing data movement between each iteration and omitting the redundant update of the main table, we can improve performance up to 48% for the FF query. However, to understand the efficiency of this optimization, we need to analyze the iterative part of each query. In the FF query, the R_i is quite inexpensive as it computes a new value based on the previous one. Thus, there are no expensive operations (i.e. joins or aggregations). The most expensive part of the computation is to move and identify updated data between the working and the main table. Thus, by minimizing this cost, we can see a great performance improvement. On the other hand, when the query contains an expensive iterative part, then the performance improvement is not significant as can be observed by the PR query in Figure 8. The reason is that the R_i of the PR query contains multiple expensive joins that take much more time than just moving data around and updating the main table. For this reason, this optimization does not yield a great performance improvement or might not have any significant benefit as it focuses on optimizing a part of the query that already takes a very small percentage of the total execution time.

To conclude, this optimization should always be applied when possible (i.e. when there is no `WHERE` clause in the iterative part) as it always yields better or similar performance. However, the effectiveness highly depends on how expensive the iterative part of the query is.

C. Common Result Optimization

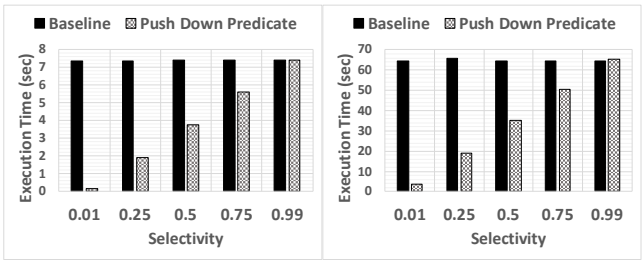
To test the effectiveness of materializing constant parts in iterative CTEs, we used the versions of the queries (for both PR and SSSP) that perform the computation only in nodes that are available by doing an additional join with the `vertexStatus` table as defined in PR-VS. In this optimization, we can see that by analyzing the iterative part of the query and materializing the part that remains constant, we can improve the performance by 20% (queries using the DBLP dataset in Figure 9). For some other cases, the performance improvement might not be that high, but is still significant at around 10% (queries using the Pokec dataset in Figure 9). In both queries, we materialize a big part of the `FROM` clause that performs multiple expensive joins that yield the same results in each iteration, something that takes a significant amount of computation time in the baseline implementation.

The effectiveness of this optimization depends i) on the number of iterations and ii) on how much time the materialized fixed part takes to be computed. The more iterations the query needs to satisfy T^c, the more redundant computations the baseline execution will need to perform. Thus, this optimiza-



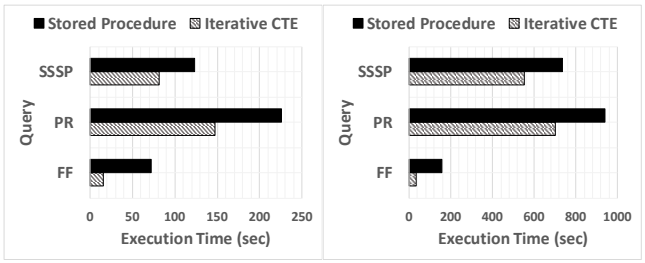
PR query with DBLP dataset | SSSP query with DBLP dataset | PR query with Pokec dataset | SS query with Pokec dataset

Fig. 9: Using MPPDB while reusing common parts



DBLP dataset | Pokec dataset

Fig. 10: Using MPPDB while pushing down predicates for the FF query



Google dataset | Pokec dataset

Fig. 11: Iterative CTEs and stored procedures

tion is pivotal in order to enable long iterative queries to be executed within reasonable time. Also, the bigger the constant part of R_i is, the bigger the performance improvement will be, as the system will be able to execute once and materialize most of the computation. This can also be seen from the difference in performance improvement between the DBLP and Pokec datasets. The DBLP dataset has 317,080 nodes and 1,049,866 edges whereas the Pokec dataset has 1,632,803 nodes and 30,622,564 edges. The constant part (i.e. table vertexStatus) has as many tuples as nodes, thus in the DBLP dataset is proportionally larger than the one in the Pokec dataset. Finally, the effectiveness is not affected by the query itself as both PR and SSSP queries demonstrate the same performance improvement patterns. This is expected as the optimization targets to minimize redundant computations in the `FROM` clause (similar for both PR and SSSP queries) and not on the `SELECT` or `WHERE` clauses.

D. Pushing Down Predicates

With this optimization we can observe more than an order of magnitude performance improvement if the selectivity of the predicate that we push is high as can be seen in Figure 10. We configured the FF query to run for 25 iterations and different selectivities. We achieved that by modifying the query as described in Section VII-A. A major limitation for most RDBMSs is that predicates are pushed mainly within the `WHERE` and `FROM` clause but not inside CTEs. In the FF query, we can see that the execution time for the baseline, remains the same independently of the selectivity. This happens because the system evaluates the CTE, and then filters out tuples using the predicate in Q_f thus, making the predicate selectivity irrelevant. However, by pushing the predicate as described in Section V, we see that the system eliminates tuples that do not contribute to the final result but still take a significant amount of time to be processed.

Moreover, by pushing the predicate to the non-iterative part, the system effectively reduces the time needed to perform a single iteration as it processes a smaller amount of data. Similar to the previous optimization, this means that the more iterations the query needs to perform, the more performance benefit this optimization will yield. Finally, the optimization is not affected by the dataset size and properties but only by the selectivity of the predicate that is pushed from the final part of the query to the non-iterative part.

E. Iterative CTEs vs Stored Procedures

Although we articulated before that declaring SQL is more efficient than stored procedures, we have further compared the performance between iterative CTEs and stored procedures as experimental evidence. We ran the PR, SSSP (both using `vertexStatus`) and FF (with 50% selectivity) queries using both iterative CTEs and stored procedures. We do not show the stored procedures code in this paper due to space limitation but we provide a brief description. For each query, we wrote a procedure that executes R_0 one time and then a loop that executes R_i for 25 times. Finally, we rewrote Q_f to call the store procedure and return the final result to the user instead of retrieving the data from a CTE table.

In Figure 11 we observe that optimized iterative CTEs are at least 25% faster than the equivalent stored procedures for the PR and SSSP queries and can be more than 80% faster for the FF query. The performance improvement for the PR and SSSP queries comes mainly from the fact that the system materializes the constant parts in R_i and uses the *rename* operator when possible, while for the FF query, the performance improvement comes mainly by evaluating predicates in Q_f earlier. Another important observation is that when the queries need to perform more iterations, then the optimized iterative CTEs are even more efficient as redundant computations have a higher impact on the cost.

VIII. Related work

One of the main applications of iterative processing is graph queries. Data management systems for graphs got a lot of attention recently due to the explosive growth of related applications like semantic web and social media. Two main approaches were taken to build graph processing engines: specialized graph systems like those in [29], [30], [31], [3], [32], [4], [33], [34], [35], [7], [19], [36] and SQL systems with graph extensions. Their difference is the mature yet complex SQL versus the custom vertex-based APIs [8] that are not as widely used. These no-SQL approaches describe computations in form of *push* or *pull* [37] communication between nodes in a graph. Our focus in this section is to discuss prior work related to extending SQL to support graph processing.

One approach to extend relational systems to support graph processing is by introducing new graph operators [38], [17]. Also, new extensions that convert UDFs [39] or vertex-centric queries [40] to SQL have been proposed. There are also systems that construct graph structures in RDBMs and use pointers from these structures to relations [41]. We believe that the above solutions are more intrusive to existing RDBMSs and require expensive implementations.

Recursive SQL queries [42], [43], [20], [24], [44] was a more native extension that mainly required changes to the API and not much to the engine itself. Also, a large number of publications focuses on Datalog [9] in order to accommodate a wider variety of recursive queries [45], [46], [10], [11], [12], [13], [47], [14], [15] efficiently. Despite that recursive queries are suitable to express some graph algorithms, they still cannot express effectively purely iterative computations. For this reason, iterative CTEs have been proposed [16], a SQL structure that constitutes the theoretical base for our work.

There is also some recent work in the context of improving performance of analytic applications written in procedural or mix of procedural and SQL. The authors in [48] proposed techniques to turn PL/SQL to SQL UDF or SQL recursive CTEs. Given the limitation of recursive queries, most conversion falls into the UDF which is less efficient than regular SQL. Even though this work is quite recent, we think it is promising and can greatly benefit from our proposed SQL extension which allows more PL/SQL being converted to SQL rather than the less efficient UDF option. The work in [49] provided optimization techniques to stored procedures which is quite useful for the general purpose stored procedures processing but not as good as cases where the application can be done completely in SQL. The authors in [17] also proposed an iterate operator similar to our loop operator. However, their approach relies on a complex engine extension to support graph operators natively. Also, accelerating iterative query plans for RDBMSs using GPUs has been studied in the context of nested SQL queries [50].

Finally, optimizations on purely iterative processing have been proposed in the context of parallel data flow systems [51], array databases [52] and MapReduce [53], [54] but their techniques cannot be applied directly to RDBMSs as the proposed optimizations depend on heterogeneous architectures and thus, on different assumptions and limitations.

IX. Conclusion

In this work, we propose a new and effective approach for implementing iterative SQL queries in RDBMSs. We describe how an iterative CTE can be rewritten using regular SQL operators and how the internal components of a RDBMS can be extended. We highlight the importance of a native solution and implement our approach in Futurewei's commercial MPPDB. Furthermore, we propose effective optimizations that i) minimize data movement, ii) reuse results that remain constant and iii) eliminate unnecessary data processing by evaluating predicates as early as possible. We evaluated our changes to the query optimizer which shows an order of magnitude improvement in some cases. Future work include estimating number of iterations for more accurate optimizer costing and expanding our optimization techniques by considering join reordering issues as mentioned in Section V-A.

References

[1] C. J. Date, *A guide to the SQL standard: a user's guide to the standard database language SQL*. Addison-Wesley Professional, 1997.

[2] S. Finkelstein, N. Mattos, I. Mumick, and H. Pirahesh, "Expressing recursive queries in SQL," *ANSI Document X3H2-96-075r1*, 1996.

[3] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[4] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 439–455.

[5] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From "think like a vertex" to "think like a graph"," *Proc. VLDB Endow.*, vol. 7, no. 3, pp. 193–204, Nov. 2013.

[6] G. Wang, W. Xie, A. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy," in *CIDR '13*, 2013.

[7] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *PPoPP '16*, 2016, pp. 11:1–11:12.

[8] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, p. 25, 2015.

[9] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about datalog (and never dared to ask)," *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 1, pp. 146–166, 1989.

[10] W. E. Moustafa, V. Papavasileiou, K. Yocum, and A. Deutsch, "Datalography: Scaling datalog graph analytics on graph processing systems," in *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 2016, pp. 56–65.

[11] J. Seo, S. Guo, and M. S. Lam, "Socialite: Datalog extensions for efficient social network analysis," in *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE, 2013, pp. 278–289.

[12] J. Seo, J. Park, J. Shin, and M. S. Lam, "Distributed socialite: a datalog-based language for large-scale graph analysis," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1906–1917, 2013.

[13] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo, "Big data analytics with datalog queries on Spark," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1135–1149.

[14] A. Shkapsky, K. Zeng, and C. Zaniolo, "Graph queries in a next-generation datalog system," *Proceedings of the VLDB Endowment*, vol. 6, no. 12, pp. 1258–1261, 2013.

[15] J. Wang, M. Balazinska, and D. Halperin, "Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1542–1553, 2015.

[16] S. Floratos, Y. Zhang, Y. Yuan, R. Lee, and X. Zhang, "Sqloop: High performance iterative processing in data management," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 1039–1051.

[17] K. Zhao and J. X. Yu, "All-in-one: Graph processing in RDBMSs revisited," in *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*. ACM, 2017, pp. 1165–1180.

[18] L. Cai, J. Chen, J. Chen, Y. Chen, K. Chiang, M. Dimitrijevic, Y. Ding, Y. Dong, A. Ghazal, J. Hebert *et al.*, "Fusion insight libra: huawei's enterprise cloud data analytics platform," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 1822–1834, 2018.

[19] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation," *IEEE Trans. on Parallel and Distributed Systems*, vol. 25, no. 8, pp. 2091–2100, Aug. 2014.

[20] A. Ghazal, D. Seid, A. Crolotte, and M. Al-Kateb, "Adaptive optimizations of recursive queries in Teradata," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 851–860.

[21] M. Al-Kateb, A. Ghazal, and A. Crolotte, "An efficient SQL rewrite approach for temporal coalescing in the Teradata RDBMS," in *International Conference on Database and Expert Systems Applications*. Springer, 2012, pp. 375–383.

[22] M. Al-Kateb, A. Ghazal, A. Crolotte, R. Bhashyam, J. Chimanchode, and S. P. Pakala, "Temporal query processing in Teradata," in *Proceedings of the 16th International Conference on Extending Database Technology*. ACM, 2013, pp. 573–578.

[23] A. Rosenthal *et al.*, "Outerjoin simplification and reordering for query optimization," *ACM Transactions on Database Systems*, vol. 22, no. 1, pp. 43–74, 1997.

[24] C. Ordonez, "Optimizing recursive queries in SQL," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 2005.

[25] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[26] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowledge and Information Systems*, vol. 42, no. 1, pp. 181–213, 2015.

[27] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[28] L. Takac and M. Zabovsky, "Data analysis in public social networks," in *International scientific conference and international workshop present day trends of innovations*, vol. 1, no. 6, 2012.

[29] M. J. Anderson, N. Sundaram, N. Satish, M. M. A. Patwary, T. L. Willke, and P. Dubey, "Graphpad: Optimized graph primitives for parallel and distributed platforms," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 313–322.

[30] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework." in *OSDI*, vol. 14, 2014, pp. 599–613.

[31] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph systems," *Proc. VLDB Endow.*, vol. 8, no. 9, pp. 950–961, May 2015.

[32] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[33] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.

[34] H. Wang, L. Geng, R. Lee, K. Hou, Y. Zhang, and X. Zhang, "Sep-graph: finding shortest execution paths for graph processing under a hybrid framework on GPU," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. ACM, 2019, pp. 38–52.

[35] Q. Wang, Y. Zhang, H. Wang, L. Geng, R. Lee, X. Zhang, and G. Yu, "Automating incremental and asynchronous evaluation for recursive aggregate data processing," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of data*. ACM, 2020.

[36] J. Zhong and B. He, "Medusa: Simplified graph processing on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, 2014.

[37] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, "To push or to pull: On reducing communication and synchronization in graph computations," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2017, pp. 93–104.

[38] L. Passing, M. Then, N. Hubig, H. Lang, M. Schreier, S. Günnemann, A. Kemper, and T. Neumann, "SQL-and operator-centric data analytics in relational main-memory databases." in *EDBT*, 2017, pp. 84–95.

[39] K. Ramachandra, K. Park, K. V. Emani, A. Halverson, C. Galindo-Legaria, and C. Cunningham, "Froid: Optimization of imperative programs in a relational database," *Proceedings of the VLDB Endowment*, vol. 11, no. 4, 2017.

[40] J. Fan, A. G. S. Raj, and J. M. Patel, "The case against specialized graph analytics engines." in *CIDR*, 2015.

[41] M. S. Hassan, T. Kuznetsova, H. C. Jeong, W. G. Aref, and M. Sadoghi, "Empowering in-memory relational database engines with native graph processing," *arXiv preprint arXiv:1709.06715*, 2017.

[42] F. Bancilhon, "Naive evaluation of recursively defined relations," in *On Knowledge Base Management Systems*. Springer, 1986, pp. 165–178.

[43] F. Bancilhon and R. Ramakrishnan, "An amateur's introduction to recursive query processing strategies," in *Readings in Artificial Intelligence and Databases*. Elsevier, 1988, pp. 376–430.

[44] C. Ordonez, "Optimization of linear recursive queries in SQL," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 2, pp. 264–277, 2010.

[45] Y. Bu, V. Borkar, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan, "Scaling datalog for machine learning on big data," *arXiv preprint arXiv:1203.0160*, 2012.

[46] M. Mazuran, E. Serra, and C. Zaniolo, "Extending the power of datalog recursion," *The VLDB Journal*, vol. 22, no. 4, pp. 471–493, 2013.

[47] A. Shkapsky, M. Yang, and C. Zaniolo, "Optimizing recursive queries with monotonic aggregates in deals," in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 2015, pp. 867–878.

[48] C. Duta, D. Hirn, and T. Grust, "Compiling pl/SQL away," in *CIDR '20*, 2020.

[49] K. Park, H. Seo, M. K. Rasel, Y.-K. Lee, C. Jeong, S. Y. Lee, C. Lee, and D.-H. Lee, "Iterative query processing based on unified optimization techniques," in *Proceedings of the 2019 ACM SIGMOD International Conference on Management of data*. ACM, 2019, pp. 54–68.

[50] S. Floratos, M. Xiao, H. Wang, C. Guo, Y. Yuan, R. Lee, and X. Zhang, "Nestgpu: Nested query processing on gpu," in *Data Engineering (ICDE), 2021 IEEE 37th International Conference on*, 2020.

[51] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl, "Spinning fast iterative data flows," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1268–1279, 2012.

[52] E. Soroush, M. Balazinska, S. Krughoff, and A. Connolly, "Efficient iterative processing in the scidb parallel array engine," in *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*. ACM, 2015, p. 39.

[53] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[54] M. Onizuka, H. Kato, S. Hidaka, K. Nakano, and Z. Hu, "Optimization for iterative queries on MapReduce," *Proceedings of the VLDB Endowment*, vol. 7, no. 4, pp. 241–252, 2013.