2018 IEEE 38th International Conference on Distributed Computing Systems

SQLoop: High Performance Iterative Processing in Data Management

Sofoklis Floratos*, Yanfeng Zhang*†, Yuan Yuan‡, Rubao Lee*, Xiaodong Zhang*
*The Ohio State University, Columbus, OH, USA
†Northeastern University, China
‡Google Inc., Mountain View, CA, USA

***Abstract*—Increasingly more iterative and recursive query tasks are processed in data management systems, such as graph-structured data analytics, demanding fast response time. However, existing CTE-based recursive SQL and its implementation ineffectively respond to this intensive query processing with two major drawbacks. First, its iteration execution model is based on implicit set-oriented terminating conditions that cannot express aggregation-based tasks, such as PageRank. Second, its synchronous execution model cannot perform asynchronous computing to further accelerate execution in parallel. To address these two issues, we have designed and implemented SQLoop, a framework that extends the semantics of current SQL standard in order to accommodate iterative SQL queries. SQLoop interfaces between users and different database engines with two powerful components. First, it provides an uniform SQL expression for users to access any database engine so that they do not need to write database dependent SQL or move datasets from a target engine to process in their own sites. Second, SQLoop automatically parallelizes iterative queries that contain certain aggregate functions in both synchronous and asynchronous ways. More specifically, SQLoop is able to take advantage of intermediate results generated between different iterations and to prioritize the execution of partitions that accelerate the query processing. We have tested and evaluated SQLoop by using three popular database engines with real-world datasets and queries, and shown its effectiveness and high performance.**

I. Introduction

Recursive and iterative query processing has become increasingly important in data analytics, as the rapid growth of social media, semantic web and knowledge database development, has imposed a large demand for iterative algorithms, and queries searching paths in a network or hierarchy relationships. Recent work has proposed specialized graph processing systems that express queries in a vertex-based API [1] that offer high performance. Moreover, extensions to the original Datalog [2] has been made in order to accommodate and execute a wider variety of recursive queries more efficiently. However, our work focuses on Relational Database Management Systems (RDBMSs) that have been used for decades and lack the ability to perform recursive or iterative queries effectively. First, our work is motivated by the fact that SQL is an already widely known and used query language, therefore, any incremental change will require a significantly smaller learning curve for the users than any other API. Second, a lot of data are already stored in RDBMSs and thus, there will be no need to move them or change their structure, which can be an expensive task both in terms of execution time and user productivity. Third, RDBMSs are well studied to handle large amounts of data, so, reusing them would enable us to exploit several existing optimizations. In this paper, we explore and address the existing limitations on the current SQL standard and propose optimizations that can accelerate iterative query processing on any RDBMS.

Common Table Expressions (CTEs) have been part of the SQL standard since 1999 [3], [4] and are used in SQL programming to reduce the complexity of complicated queries. In general, CTEs are temporarily named result sets that the user can reference within a SQL statement. A CTE also supports recursive evaluation that enables users to express hierarchy queries or path traversing algorithms. However, despite the fact that CTEs exist in the SQL language for almost 20 years and are supported by popular database engines like Microsoft SQL Server [5], PostgreSQL [6], SQLite [7] and Oracle MySQL [8], not all database systems implement the recursive evaluation.

SQL programming interface is a critical link between users and data processing systems to easily express and efficiently parallelize query tasks. However, current CTE-based SQL and its implementation has two major drawbacks to support a wide scope of iterative computations. First, its iteration execution model, based on implicit set-oriented terminating conditions, cannot express aggregation-based tasks, such as PageRank [9], and second, its synchronous execution model cannot perform asynchronous computing to further accelerate execution in parallel. To address these issues, we have designed and implemented SQLoop, a framework that extends the semantics of current SQL standard in order to accommodate iterative SQL queries. SQLoop interfaces between users and different database engines with two powerful components. First, it provides a uniform SQL expression for users to access any database engines so that they do not need to write database dependent SQL or move datasets from a target engine to process them in their own sites. Second, SQLoop automatically parallelizes iterative queries that contain certain aggregate functions in both synchronous and asynchronous ways. More specifically, SQLoop is able to take advantage of intermediate results generated between different iterations and also to prioritize the execution of partitions to asynchronously accelerate the query processing.

In pursuance of a generalized solution we avoid to alter the implementation of a current SQL database engine which will

2575-8411/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDCS.2018.00104

IEEE computer society

introduce dependency with a specific system and potentially further limitations such as the expensive and mundane data loading step. Instead, our system is implemented as a middleware that exists between the user and the database engine, providing in this way a general purpose implementation that can be used with most SQL engines that support transaction management and JDBC connections [10]. Figure 1 presents an overview of SQLoop.

To accelerate iterative CTE queries, our system spawns multiple threads that open different connections with the underlying database system. For each new connection that the database system receives, it spawns a new process in order to accommodate the additional computational needs. In this way, although our system does not have explicit control over the processes used by the underlining database engine, it is still able to utilize multiple CPUs while executing a single iterative query.

In order to parallelize the iterative CTE query submitted by the user, our system partitions the main CTE table into smaller ones. Then, for each partition of the main table, it generates multiple computational tasks that contain regular SQL statements. Each statement is submitted by SQLoop to the database using the JDBC connection obtained by the thread that executes this specific task.

We propose three different parallel execution methods that can be used if a query uses one of the standard SQL aggregate functions: SUM, MIN, MAX, $COUNT$ or AVG. The first one is the Synchronous Execution method which is based on a two-phase computation. During the first phase, the system performs all the computations that are related to a given partition and creates a message table with data that will be needed by other partitions. In the second phase, the system creates tasks that are associated with a given partition and their only purpose is to read data needed by other partitions. The second one is the Asynchronous Execution method and it is based on the delta-based accumulative iterative computation (DAIC) proposed in [11] that allows the computation to use intermediate results of the current iteration. Finally, the third parallel execution method is the Prioritized Asynchronous Execution and it implements a priority scheduling by performing first the computations on the dataset partitions that have the potential of accelerating the execution of the iterative query. We remark that query parallelization is transparent to the user. Except from the actual iterative CTE, the user does not need to specify any other properties to the system. SQLoop automatically parses and analyzes the query properties in order to execute it in parallel.

Finally, SQLoop implementation is lightweight, as it does not perform any actual computation on the data. Instead, it manages the data partitions, the queries to be submitted in the target database and the thread scheduling. Since our implementation uses JDBC connections, it can also work with remote database systems. In general, it is possible to create connections with multiple RDBMSs on different machines by specifying the URL of each target database engine and use SQLoop to redirect the queries on demand.

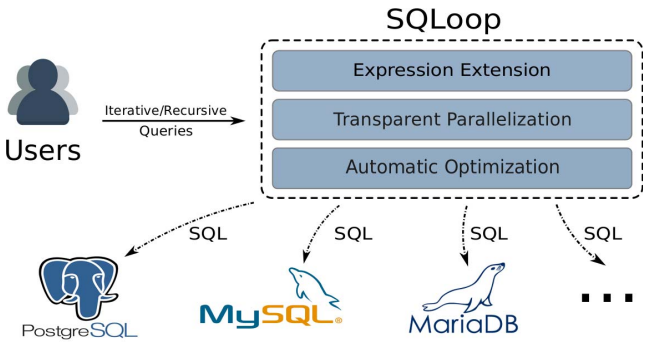


Fig. 1: Execution flow of SQLoop: Iterative/Recursive queries submitted by users are regulated by our CTE expression extension, and automatically parallelized with optimizations in both synchronous and asynchronous ways before being executed in different database engines, such as PostgreSQL, MySQL, MariaDB, and others.

We have evaluated SQLoop and its optimizations using three popular RDBMSs, namely, PostgreSQL [6], MariaDB [12] and Oracle MySQL [8]. Using real-world datasets and queries in our extensive experiments, we show the effectiveness and high performance of our system.

The **contributions** of this paper are summarized as follows:

- An extension is added to the current SQL standard to enable users to express iterative queries on relational data, which serves as a common interface between users and any database engine.
- The development of a set of optimizations has been made, which significantly accelerates execution by transparently parallelizing and asynchronously executing iterative queries that contain certain aggregate functions.
- The iterative query expressions, query optimizations and automatic parallelization are integrated into the SQLoop framework, which effectively bridges users to any database engines in a system independent way.
- A strong case is made for asynchronous iterative query processing not only for its high performance but also for our automatic approach in SQLoop, which would open a door to widely apply asynchronous parallel models in data processing.

The rest of the paper is organized as follows: Section II provides background information on recursive Common Table Expressions (CTEs). Section III introduces the iterative CTEs. Section IV presents SQLoop implementation. Section V discusses the proposed parallel execution algorithms. Section VI presents the experimental studies. Finally, Section VII discusses related work and Section VIII concludes.

II. Background

A. Recursive CTEs

In SQL 1999 standard, CTEs provide the ability to create a temporary view that is visible only within a single query. The purpose of CTEs is mainly to reduce repeated computations of complex queries that are referred to more than once. A CTE can be defined explicitly with the use of a `WITH` clause

or implicitly by the query optimizer. Furthermore, the user can define a recursive CTE by using the word RECURSIVE right after WITH. Although this is just a convention that is not followed by all database systems that implement recursive CTEs, we will use it in order to distinguish recursive CTEs from non-recursive ones. The general syntax of a recursive CTE is:

WITH RECURSIVE R **AS (**R^0 **UNION ALL** R^i**)** Q^f

In general, a recursive CTE named R is composed by a non-recursive query R^0 (also mentioned as anchor table or seed) and by a recursive one R^i. The recursive part R^i should refer to R only once in order to ensure linear recursion. The evaluation of the recursive CTE stops when no more rows are returned by the evaluation of R^i. While R^0 and R^i denote the non-recursive and recursive parts of the query in general, for simplicity, we will also use it as a reference to the result rows produced by R^i during the i^{th} execution, with R^0 referring to the result rows produced by the non-iterative part of the query. Table R contains the union of all intermediate results thus, $R = R^0 \cup R^1 \cup \cdots \cup R^n$ where n is the total number of recursions that were performed by the database engine. Finally, query Q^f will be executed on table R to calculate the final result. Example 1 illustrates how a recursive CTE is able to calculate the sum of the numbers in the Fibonacci sequence that are less than a thousand.

A widely known evaluation technique that is used by many databases (like PostgreSQL) is the *semi-naive* evaluation [13]. The baseline *naive* algorithm evaluates recursive queries multiple times until no more rows are returned. However, this approach is inefficient as it will reproduce already calculated rows in each recursion. The *semi-naive* approach is an optimization over the baseline *naive* algorithm that eliminates duplicates by providing as an input to the next recursion only the output rows of the last recursion (i.e. the delta between the last recursion and the current one).

A recursive CTE is almost identical with a recursive view with only one difference, the scope. While the CTE can be viewed only within a query, a view can be defined and used in more than one queries. In this paper, we explore recursive CTEs instead of views. However, the insights that apply to one SQL structure can be easily applied to the other, if we omit the scope difference. Previous work has focused on optimizing the evaluation of recursive views [14] and recursive processing in general is well studied in the context of Datalog [15]–[23].

B. Problem Statement

Although recursive CTEs extend the expressiveness of SQL as they can accommodate hierarchy queries and some graph traversal algorithms, their usage is still limited. One of the main limitations of recursive CTEs are aggregate functions, as they cannot be used on the recursive part of the query. This limitation makes CTEs unable to express efficiently a large category of iterative algorithms such as PageRank, SimRank [24], HITS [25], Connected Components [26], etc.

Example 1 Recursive CTE that calculates the sum of the numbers in the Fibonacci sequence that are less than 1000.

```
WITH RECURSIVE Fibonacci(n,pn) AS (
    VALUES (0,1)
UNION ALL
    SELECT n + pn, n
      FROM Fibonacci
     WHERE n < 1000
)
SELECT SUM(n) FROM Fibonacci;
```

Moreover, results between different recursions R^i can only be concatenated with the use of UNION ALL operator ($R = R^0 \cup R^1 \cup \cdots \cup R^n$). In other words, the final table R cannot be updated but only monotonically appended. This property can be a problem for algorithms like PageRank and SimRank that need to update the table and not to append it. The current alternative solution is to submit a SQL script with multiple queries. For each iteration the user will need to write queries that (i) calculate and store in a temporary table the desired results and (ii) update R with the results of the temporary table. A single SQL query in most engines can update only one table thus, to perform one iteration, the user will need to submit at least two queries. If we also take into consideration that the user will need to include SQL statements to manage all the necessary tables, indexes, etc., then the script can easily exceed 100 lines.

Parallelizing and accelerating recursive queries while following the semantics of the *semi-naive* evaluation that is used by most database systems can also be challenging due to the synchronous nature of the computation. Each recursion R^i should have as an input all the rows generated by the previous recursion R^{i-1}, a requirement that creates an implicit barrier between R^{i-1} and R^i. In a distributed computation this would imply two facts: (i) all the computations related to R^{i-1} should be completed before starting executing tasks for R^i, even if some of the computations are not relevant to R^i, (ii) even if there are available resources to start the computation of R^i, the system should wait for all the pending computations of R^{i-1} to finish.

Finally, perhaps the most fundamental problem of recursive CTEs is that they assume that a query needs to reach a fixpoint, a particular set of rows that satisfies the recursive relation. However, this is not the case for all iterative algorithms. For example, the PageRank algorithm is terminated when a particular threshold has been satisfied. This threshold is not expressed through the set of rows returned by the evaluation of the query but rather in the data values themselves. In other words, recursive CTEs append rows after each iteration, while in PageRank existing rows need to be updated iteratively. For this reason, we argue that recursive CTEs are queries that are focused on finding a **result set** rather than a **result value**.

III. Iterative SQL

A. Iterative CTEs

To address the problems explained in the previous section, we propose a new SQL structure, the iterative CTEs. In order

Type	T^c Syntax	T^c Satisfaction
Metadata	n **ITERATIONS**	After n iterations.
	n **UPDATES**	When R^i updates less than n rows on table R.
Data	$expr$	When $expr$ returns $\|R\|$ rows.
	ANY $expr$	When $expr$ returns at least 1 row.
	$expr <, =, > e$	When $expr$ returns a value smaller, equal or greater than e respectively.
Delta	**DELTA** $expr$	The user can refer to the previous iteration by using R^{delta} inside $expr$.
	ANY DELTA $expr$	
	DELTA $expr <, =, > e$	

TABLE I: Different types of termination conditions

to introduce an iterative CTE, we follow the same logical structure as the one that already exists in a recursive CTE. We replace the keyword `RECURSIVE` with `ITERATIVE` in order to distinguish between the two different types of queries. An iterative CTE starts by executing the initial query R^0 and by storing the result into table R. Then, the query executes the iterative part R^i and updates the values of R. We replace the `UNION ALL` keyword with `ITERATE` as the query now updates R instead of monotonically appending rows to it. The execution of R^i will be repeated by the query until the termination condition T^c is satisfied. Table R will contain the most updated version that has been resulted by the execution of the query R^i for the n^{th} time, where n is the number of iterations performed by the query in order to satisfy T^c. Finally, query Q^f will be executed to yield the final result. We define the general syntax of an iterative CTE as:

WITH ITERATIVE R **AS (**R^0 **ITERATE** R^i **UNTIL** T^c**)** Q^f

One assumption that our model makes, is that the first column of the table R is a primary key. We call this column R_{id}. We assume that once R^0 yields the first set of values for R_{id}, then this set cannot be changed by R^i. The user should define R^i in such a way that its execution will either return the same set or a subset of R_{id}. Using R_{id} is a necessary assumption that we need to make in order to update and partition table R correctly. The Partition of the table R is necessary in order to parallelize the query and it is discussed in Section V. The single-threaded execution algorithm for SQLoop uses a temporary table R^{tmp} to store the results of R^i in each iteration. We denote the primary key column of R^{tmp} as R_{tmp_id}. At the end of each iteration, SQLoop updates R with the rows of R^{tmp} by matching the keys in the columns, R_{id} and R_{tmp_id}. Thus, only the rows that have primary keys that satisfy $R_{id} \cap R_{tmp\_id}$ will be updated in a given iteration. Another way to understand R_{id} is to describe it as a unique row identifier that is used during the partition and the update of the CTE table. Iterative CTEs so far address the problem of "append only" issue that exists in recursive CTEs. By letting the user to update different rows of R in each iteration, we enable him to express iterative algorithms that update values and not sets.

B. Termination Conditions

Another key difference between a recursive CTE and an iterative one, is the termination condition. A recursive CTE terminates once it reaches a fix-point, that is, once executing R^i, it does not generate any new relations. During the execution of a recursive CTE, the termination is implied and it depends on the database engine that executes the query. However, this approach can be inefficient for iterative processing as most of the queries should terminate based on data values, metadata or delta differences which are hard to be captured using the fix-point semantics. As an example, we can consider some possible terminations for the PageRank algorithm. One of the most common termination condition is to simply repeat the algorithm for 10 iterations, which means that the termination of the query relies on the metadata. Another possible termination condition is to stop the calculation if the dataset reached a convergence point, something that can be expressed either by checking the data values of the dataset itself, if the convergence point is known, or to set a threshold e for which the delta rank (i.e. difference on the rank of a node between iteration i and $i-1$) should be smaller.

In order to address this problem, we propose three different types of termination conditions that are based on data, metadata and delta values. The termination condition T^c on an iterative CTE is defined explicitly by the user after the keyword `UNTIL`. As the keyword implies, the query should continue to execute R^i until T^c is satisfied. As in programming languages, the user is responsible to ensure that T^c is well defined and that it will be satisfied after n iterations. An overview of the different types of termination conditions can be found in Table I.

When T^c satisfaction is related to **metadata**, SQLoop checks either the number of iterations or the number of updates performed by R^i during the last iteration. If the user specified that R^i should be executed n times using the syntax `UNTIL` n `ITERATIONS`, then SQLoop simply uses a for loop to execute R^i for n times. If the user specified that R should be terminated if R^i updated less than n rows during the last iteration using the syntax `UNTIL` n `UPDATES`, then SQLoop checks after each iteration the number of updated rows resulted by the execution of R^i to decide if T^c is satisfied.

When T^c satisfaction depends on **data values**, SQLoop checks if the relations that exist in R satisfy a regular SQL expression $expr$ given by the user. SQLoop gives the user the ability to choose if $expr$ should be satisfied by all the relations in R or at least by one, something that can be specified by using the keyword `ANY`. To determine how many relations in

Example 2 PageRank

```
1  WITH ITERATIVE PageRank(Node, Rank, Delta)
2  AS(
3   SELECT src,0,0.15
4   FROM ( SELECT src FROM edges
5          UNION
6          SELECT dst FROM edges) AS alledges
7   GROUP BY src
8  ITERATE
9   SELECT PageRank.Node,
10   COALESCE(PageRank.Rank+PageRank.Delta,0.15)
11   COALESCE(0.85 * SUM(IncomingRank.Delta
12                  * IncomingEdges.weight),0.0)
13  FROM PageRank
14   LEFT JOIN edges AS IncomingEdges
15     ON PageRank.Node = IncomingEdges.dst
16   LEFT JOIN PageRank AS IncomingRank
17     ON IncomingRank.Node = IncomingEdges.src
18   GROUP BY PageRank.Node
19 UNTIL 100 ITERATIONS)
20 SELECT Node, Rank FROM PageRank
```

Example 3 Single Source Shortest Path

```
WITH ITERATIVE sssp (Node, Distance, Delta)                1
AS (                                                       2
 SELECT src, Infinity,                                     3
        CASE WHEN src = 1 THEN 0                           4
             ELSE Infinity END                             5
 FROM ( SELECT src FROM edges                              6
        UNION                                              7
        SELECT dst FROM edges) AS alledges                 8
 GROUP BY src                                              9
ITERATE                                                    10
 SELECT sssp.Node,                                         11
  LEAST(sssp.Distance,sssp.Delta),                         12
  COALESCE(MIN(Neighbor.Distance                           13
        + IncomingEdges.weight), Infinity)                 14
 FROM sssp                                                 15
    LEFT JOIN edges AS IncomingEdges                       16
      ON sssp.Node = IncomingEdges.dst                     17
    LEFT JOIN sssp AS Neighbor                             18
      ON Neighbor.Node = IncomingEdges.src                 19
  WHERE Neighbor.Delta != Infinity                         20
  GROUP BY sssp.node                                       21
UNTIL 0 UPDATES)                                           22
SELECT sssp.Distance FROM sssp                             23
                     WHERE sssp.Node = 100                 24
```

R satisfy the expression given by the user, at the end of each iteration, SQLoop executes $expr$. If the result set is equal to the number of rows $|R|$ (or at least one for `ANY`) that exist in R, then T^c is satisfied. Moreover, SQLoop can compare the result of $expr$ with an actual number e if the user inputs $expr <, =, > e$. In this way, a T^c can be based on the result of an aggregation. For this case, SQLoop assumes that $expr$ will return only one row, with only one value.

Finally, in order to give the user the ability to use **delta values** in T^c, we introduce the keyword `DELTA`. The purpose of this type of termination condition is to calculate the difference between values of the current iteration and the previous one. Calculating delta values enables the user to express a T^c that is satisfied if the progress of the last iteration was not significant. If SQLoop detects the keyword `DELTA` before $expr$ then at the end of each iteration, it simply copies the data from R to a new R^{delta} table and checks how many relations satisfy $expr$ in the next iteration using the same logic as before.

Allowing users to explicitly define a termination condition for their iterative queries does not only increase the expressiveness of our model but it also removes the limitations imposed by the *semi-naive* evaluation technique used to execute recursive CTEs. An important observation is that aggregate functions are allowed in the iterative part of the query and thus, iterative algorithms like PageRank can now be expressed more efficiently.

C. Examples

To provide a deeper understanding and further motivation, we provide two different examples of iterative CTEs. Example 2 applies the PageRank algorithm to the entire dataset, a purely iterative process. Example 3 finds the shortest path from a given source, a query that traverses the dataset by exploring connected nodes. For both examples we assume that the table `edges` contains rows with three attributes, `src`, `dst` and `weight`. Each row in table `edges` represents an edge going from node `src` to node `dst` and has `weight` calculated as $\frac{1}{outdegree}$.

In Example 2, we observe that the CTE defines a table `PageRank` with three columns, `Nodes`, `Rank` and `Delta`. Column `Nodes` contains all the node ids and also serves as the primary key (R_{id}). Column `Rank` contains the rank of each node and column `Delta` accumulates incoming rank. The non-iterative part of the CTE (Lines 3-7) queries the `edges` tables in order to obtain all the unique node ids. Also, it initializes all ranks to 0 and all deltas to 0.15. The iterative part of the query (Lines 9-18), selects all the node ids in order to return the same set of primary keys. Then, in each iteration `Rank` accumulates `Delta` and `Delta` accumulates all the incoming rank from the neighbor nodes based on the formula $0.85 * SUM(IncomingRank.Delta * IncomingEdges.weight)$. The `FROM` clause (Lines 13-17) of the iterative part, joins the `PageRank` with the `edges` table (Lines 13-15) in order to obtain all the incoming edges for a given node and then performs a self join of the resulting table with the `PageRank` table (Lines 16-17) in order to obtain the rank of the incoming edges. The query uses left joins in order to mark the nodes that do not have incoming edges with $NULL$ and uses the `COALESCE` function to replace $NULL$ values with the default values (which is 0.15 for the rank and 0 for the delta). Finally, the termination condition (Line 19) specifies that the PageRank algorithm should be repeated for 100 iterations and the final query (Line 20) returns all the nodes with their calculated ranks at the end. A more detailed explanation of the PageRank algorithm expressed by this query can be found in [11].

Example 3 defines a CTE `sssp` with three columns, `Nodes`, `Distance` and `Delta` that follow the same logic as the previous example. The main difference is that instead of rank, the query now calculates distance from a given source. The non-iterative part of the CTE (Lines 3-9) extracts all the node ids from the `edges` table and assigns infinite distance

to all of them except from the source, which in our example is node with ID 1. The iterative part (Lines 11-21) visits all the connected nodes, doing one step per iteration and calculating the new distance. In each iteration, the query stores the newly calculated distance to delta (Lines 13-14) and in the next iteration checks if the new distance is smaller than the current one (Line 12). The `FROM` clause (Lines 15-19) of the iterative part follows the same logic as Example 2. This query terminates when there are no more rows to update (Line 22). Lastly, the final query (Lines 23-24) returns the distance from the source to the destination, which in our example is the node with ID 100.

IV. System Design and Implementation

SQLoop consists of three components: (1) an API to accept recursive and iterative CTEs, (2) a translator to turn incoming CTEs into regular SQL queries executable in various database engines, and (3) a parallel execution engine. An overview of the system can be found in Figure 2.

A. System Architecture

To enable database systems to execute recursive and iterative CTEs, we developed SQLoop. SQLoop accepts CTEs from the user and translates them to regular SQL queries that can be executed by most database engines. Implementing our system as a middleware, allows us to develop flexible and effective solutions that are not engine dependent and thus, letting users process data in their own RDBMS without the need to transfer them. Moreover, this solution ensures a consistent API and SQL syntax across multiple systems. The user is able to connect with a target database engine by specifying only the URL and the port number. Then, CTEs can be executed without providing any further details. Our intension is to enable users to integrate SQLoop into their work-flow seamlessly and easily.

SQLoop uses JDBC connections to submit SQL statements to the target database systems. Several features offered by the JDBC drivers are vital to ensure correct and efficient execution for our system. First, JDBC offers the ability to batch multiple SQL statements together. Our system uses this feature, when possible, to avoid communication overhead. Second, JDBC drivers ensure efficient data and meta-data transferring between our system and the database engine. Even though SQLoop does not process or handle large amounts of data, it still needs to access them efficiently. Finally, SQLoop parallelizes iterative CTEs as described in Section V and thus, it needs to submit and execute multiple queries concurrently. The JDBC drivers not only give the ability to define the beginning and the end of a transaction but also to determine the transaction isolation level. These JDBC features are critical in order to execute concurrent queries efficiency.

B. Query translation

When the user submits a new query, SQLoop uses a custom parser created by antlr4 [27] based on the original SQL grammar in order to extract information. If the query is not a recursive or an iterative CTE, then it is marked as a regular SQL statement and it is executed as such. Otherwise, SQLoop searches for the keywords `RECURSIVE` or `ITERATIVE` in order to distinguish between the two different types of CTEs. The parser then yields further information regarding the CTE such as the name of the main table, the columns and the different parts of the query (i.e. R^0, R^i and Q^f).

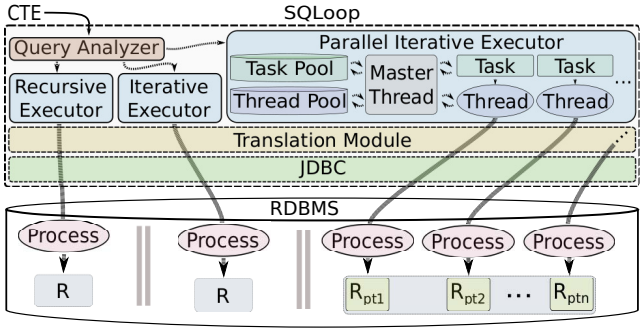


Fig. 2: SQLoop internal components and architecture. SQLoop accepts CTEs, analyzes them and uses different executors to process them. The communication with the RDBMS goes through the translation module and JDBC connections.

During the first step of the execution, SQLoop creates the main CTE table R with the corresponding columns in the target database engine using the SQL command `CREATE TABLE`. Then SQLoop executes R^0 and stores the result into R using the SQL command `INSERT INTO` R R^0. As explained in Section III, our system uses a table R_{tmp} in order to store temporarily the results of R^i. If SQLoop executes a recursive CTE, then it appends the data from R^{tmp} to R using again an `INSERT INTO` SQL statement. Otherwise, if the CTE is iterative, SQLoop updates R with the values of R^{tmp} by matching the keys in the R_{id} and R_{tmp_id} columns. In order to decide when to stop executing R^i for a recursive CTE, SQLoop checks if the query reached a fix-point, that is, if the last execution of R^i yielded any new rows. If the CTE is iterative, it applies the logic described in Section III in order to decide the termination of the query. Finally, query Q^f is executed for both iterative and recursive CTEs in order to yield the final results.

A common problem that arises with the use of different database systems is the SQL syntax. In order to resolve this problem, SQLoop has a query translation module that is dedicated to resolve syntax problems that are related to specific database systems. This module contains pre-defined rules that dictate how a given type of query should be rewritten for a given target database engine. In order to ensure transparency for the user, SQLoop uses this module every time before it submits a new query to ensure consistency. SQLoop is able to auto-configure the translation module based on the JDBC drivers that are used.

C. OLAP Architecture

We have built and designed SQLoop under the assumption that it will be used mainly as part of OLAP work-flows

such as data mining and analytics. Thus, SQLoop operates under the assumption that during the execution of an iterative query, the involved database tables will not be subject to any updates. Basically, the only assumption that SQLoop does is that tables involved in a recursive or an iterative CTE will not be altered during the execution of the query. The rest of the tables and queries, submitted to the target database, can still be executed in parallel (with ACID [28], [29] properties). Supporting transactions, for the queries that SQLoop currently runs and the tables involved, requires a complicated solution that is perhaps material for a future work. We acknowledge the importance of supporting transactional processing but we argue that in some cases the flexibility offered by our design can outweigh this limitation.

V. Parallel Execution

During the processing of an iterative query, most of the time goes into the execution of the iterative part. This section discusses how SQLoop automatically parallelizes and optimizes the iterative part of the CTE when certain properties are satisfied.

A. Query Analysis

SQLoop analyzes R^i in order to determine if its execution can be parallelized. The focus of this work is to parallelize queries that use regular SQL aggregate functions: SUM, MIN, MAX, $COUNT$ or AVG. A comprehensive list of algorithms that can be supported using aggregate functions can be found in [30]. Moreover, distributed aggregation and the limitations imposed by the properties of the aggregate functions have been studied in [31]. During the query analysis, SQLoop tries to verify that R^i contains one of the aggregate functions mentioned above in its `SELECT` clause. If the query does not contain a supported aggregate function, then SQLoop executes the query using the single-threaded method described in Section IV.

The next step for the query analyzer is to decide if there are any data that need to be exchanged during the parallel execution and under the assumption that the original table R will be partitioned. To determine that, SQLoop further analyzes the `FROM` clause of R^i in order to detect self-joins. If R is partitioned into multiple tables R_{pt1}, R_{pt2}, ... R_{ptn}, then in order to execute a self-join, every partition R_{ptx} where $x \in [1, n]$ will need to process rows from every other partition R_{pty} where $y \in [1, n]$. If no data need to be exchanged between the different partitions, then the parallelization of the iterative part is a straightforward process as it can be done by simply partitioning, and processing the dataset in parallel without any communication during each iteration. Thus, we focus mainly on self-joins as they need to exchange data and are used in iterative queries to express incoming information. Finally, the query analyzer also determines the columns of R that need to exchange information between the different partitions. To achieve that, it searches again the `SELECT` clause in order to identify references of tables that are the result of the self-join. We denote these columns as $R^{i_{delta}}$.

To provide better understanding, let us revisit the PageRank query in Example 2. The query analyzer searches the `SELECT` clause (Lines 9-12) in order to determine if R^i contains SUM, MIN, MAX, $COUNT$ or AVG. In our example, R^i contains SUM and thus, the query qualifies as parallelizable. Then, the analyzer searches the `FROM` clause (Lines 13-17) for a self-join and finds one in Line 16. Finally, the analyzer scans again the `SELECT` clause to determine which column calculation will need to process data from other partitions, which in our example is column `Delta`.

B. Query Parallelization

After the query analysis, SQLoop knows if the CTE can be parallelized and which column calculations will need to exchange data between different partitions. Before parallelizing the iterative part, SQLoop executes R^0 and stores the result to R. While developing SQLoop, we observed that join operations were the most expensive part of the computation. We also observed that part of the result remains constant. As an example, we can consider the first join in the PageRank algorithm in Example 2 (Lines 13-15) that finds all the incoming edges. During the execution of the query, table `edges` does not change and as such, incoming edges do not change either. To provide better performance, SQLoop materializes the part of the join result that remains constant. To achieve that, our system performs the join contained in the `FROM` clause and projects only the attributes that are used in a column computation of R^i but are not being updated. During a self-join, it also keeps the columns that contain the R_{id} of each intermediate table in order to maintain the row matching between them. In the PageRank example, these attributes are `PageRank.Node`, `IncomingEdges.weight` and `IncomingRank.node`. The columns `src` and `dst` of table `IncomingEdges` are omitted as they are not used by R^i. The result is stored in table R_{mjoin}. Finally, SQLoop rewrites the `FROM` clause of R^i and replaces the expensive joins with R_{mjoin}. This optimization greatly improves the performance, as a lot of redundant computations are omitted. Materializing the constant part of an iterative computation is not the focus of this work and therefore further description of the optimization shall not be provided. Related work [32] has explored this topic extensively.

SQLoop creates a thread pool in which, each thread opens a new connection with the target database engine. Despite the fact that SQLoop does not have explicit control over the resources used by the database systems, in general each new connection is accommodated by a separate process. Hence, creating multiple threads with different connections in SQLoop, results into parallel query execution. To avoid overhead created by concurrent queries that read and write on the same table due to locking, SQLoop partitions R into multiple tables R_{pt1}, R_{pt2} ... R_{ptn} by applying a hash function on R_{id}. Moreover, to avoid copying data at the end of R^i back to R, we re-define R as a view of $R_{pt1} \cup R_{pt2} \cup \cdots \cup R_{ptn}$.

SQLoop automatically determines the number of threads depending on the available CPUs of the system, assuming

that both SQLoop and the target database engine are on the same machine. In order to avoid occupying resources from the database engine, SQLoop uses half of the available CPUs. Moreover, SQLoop by default uses 256 partitions to take advantage of the asynchronous techniques that are discussed in the next section. However, both number of threads and number of partitions can be re-defined by the user through the SQLoop API.

C. Two-phase $Compute/Gather$ computation

For each partition R_{ptx} where $x \in [1, n]$, SQLoop creates two different types of tasks, named $Compute$ and $Gather$. A $Compute$ task is responsible to perform computations that are related only to R_{ptx} and a $Gather$ task is responsible only for gathering results needed from other partitions. To ensure consistency, SQLoop performs only one task at a time for a given R_{ptx}. In order to enable SQLoop to process R^i asynchronously, we have designed the $Compute$ and $Gather$ tasks based on the 2-phase update function of the delta-based accumulative iterative computation (DAIC) proposed in [11]. The $Compute$ task performs the first phase of the update function while the $Gather$ task the second. Figure 3 provides an overview of how $Compute$ and $Gather$ tasks are performed by SQLoop.

A $Compute$ task contains two steps. The first step ensures that other partitions will be able to access the needed data efficiently. Thus, it performs the computations of $R^{i_{delta}}$ and then stores the results to a separate table, named *message table*. Except from the results of $R^{i_{delta}}$, a *message table* also contains R_{id} in order to ensure that $Gather$ tasks will be able to gather the data correctly. Each time a new $Compute$ task is performed, it creates a new *message table*. Creating multiple *message tables* resolves the problem of having a $Compute$ task writing to R_{ptx} while other $Gather$ tasks are trying to read from it. Modern database systems are able to handle read and write conflicts but they will still introduce overhead due to the exclusive locking from the writing queries. After completing the creation of the *message table*, the task updates a global data-structure that is visible across all SQLoop threads. During the second step, a $Compute$ task performs the calculations in columns that are in $R^i \setminus R^{i_{delta}}$ (i.e. in R^i but not in $R^{i_{delta}}$). These computations do not need to access data from other partitions.

A $Gather$ task contains only one step, and this is to update $R^{i_{delta}}$ by reading all the unread *message tables* that have been created since the last time that a similar task for R_{ptx} was performed. Keeping all the available *message tables* into a global data structure makes this task easy. However, an important consideration during a $Gather$ task is performance, as it needs to read multiple *message tables*. To avoid the overhead of submitting multiple SQL queries that scan R_{ptx} every time in order to match the R_{id} columns, SQLoop creates a single query that contains the union of all the *message tables* that need to be processed. Moreover, indexes on all tables (R_{ptx} and *message tables*) ensure that unnecessary scans will be avoided when possible.

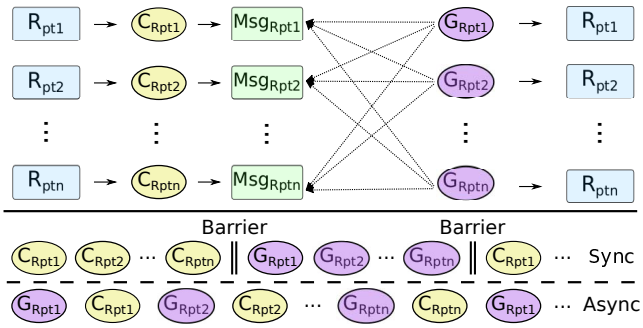


Fig. 3: Two-phase $Compute/Gather$ computation and scheduling policy of $Sync$ and $Async$ execution. C_{Rptx} and G_{Rptx} are $Compute$ and $Gather$ tasks for partition R_{ptx}. Msg_{Rptx} is the latest *message table* created by a $Compute$ task for partition R_{ptx}.

D. Accumulating data change

To enable asynchronous processing and avoid the need for synchronization, $Compute$ and $Gather$ tasks are designed to **accumulate** the changes that are made from other partitions on the $R^{i_{delta}}$ columns. This is a key point for the asynchronous parallelization and also the limitation that prevents SQLoop from parallelizing a wider variety of aggregate functions. Data accumulation in $R^{i_{delta}}$ is based on, (i) the data stored by the $Compute$ task to a *message table* and (ii) the function used during the $Gather$ task to accumulate the change from other partitions to $R^{i_{delta}}$.

Accumulating the changes on data for SUM, MIN and MAX is relatively simple, as they satisfy both the distributive and accumulative properties. This means that the $Compute$ task can store in the *message table* the SUM, MIN or MAX value for a specific row and then a $Gather$ task will be able to calculate a SUM, MIN or MAX for another row of another partition, using the data stored in the *message table* at a later time. For example, in the PageRank query, the $Compute$ task stores in the *message table* the current SUM of incoming rank of a given node and then, a $Gather$ task accumulates all the incoming rank from nodes that have incoming edges coming from other partitions, by performing a SUM over all of their values that exist in the respective *message tables*.

However, implementing the $COUNT$ and AVG aggregate functions in the same way will not yield the correct results. For the $COUNT$ function, the problem arises when the $Gather$ task tries to accumulate the results from other partitions. If the $Gather$ task applies the $COUNT$ function on the data of the *message tables* again, then it will basically calculate the number of incoming messages for each row and would not accumulate their count. To do that, it would need to perform a SUM over these values instead. Moreover, for the AVG aggregate function, in order to enable a $Gather$ task to accumulate results from other partitions, it would need both their SUM and $COUNT$ values. SQLoop is aware of these limitations and for this reason, it alters the function used on the $Gather$ task and the data stored in the *message tables* based on the aggregate function that is used in the query.

E. Asynchronous Parallel Execution

SQLoop has a master thread that is responsible for assigning $Compute$ and $Gather$ tasks to available threads from the pool. There are three different parallel execution methods that assign tasks to threads using different policies.

The simplest one is the Synchronous Execution ($Sync$) and ensures a purely iterative processing by performing a two-phase computation. During the first phase, SQLoop assigns all the $Compute$ tasks to available threads. If the number of threads is less than the number of tasks, then the master thread waits until more threads become available. During the second phase, SQLoop assigns all the $Gather$ tasks to available threads using the same logic. An iteration is complete when all tasks have been completed. This kind of execution has an explicit barrier between the two phases and each iteration is distinct.

While the $Sync$ Execution is simple and effective, the generated results are used by another $Gather$ task only after all other $Compute$ tasks are completed. To accelerate the query computation using newly generated results, the Asynchronous Execution ($Async$) implemented in SQLoop, schedules first a $Gather$ task and then a $Compute$ task. In this way, data that become available by the execution of a $Compute$ task are processed immediately by another $Gather$ task. Figure 3 contains the scheduling policy of n partitions for both $Sync$ and $Async$ executions. In the $Sync$ execution, the computation of the query advances only during the second phase when $Gather$ tasks are performed. In this case, all $Gather$ tasks process data only from the previous iteration. However, in the $Async$ execution, the computation advances every two tasks, propagating in this way the changes faster. For our example, G_{Rpt2} will not only process the available data from the previous iteration, but also the intermediate data (i.e. data from this iteration) of R_{pt1} as C_{Rpt1} has been already performed (and thus, a new *message table* exists). Following the same logic, we can observe that G_{Rptn} will process data from the previous iteration ***and*** intermediate results from R_{pti} where $i \in [1, n)$, accelerating in this way the query computations. An important observation is that the more partitions that exist, the faster intermediate results will be propagated as n becomes bigger for the same number of rows. Although asynchronous execution in the past has been used to cope with waiting problems related to distributed systems such as heterogeneous clusters, in our case, it accelerates the query by allowing intermediate results to be processed faster.

Finally, SQLoop also supports Prioritized Asynchronous Execution ($AsyncP$) in order to avoid scheduling tasks that do not contribute to the progress of the query. As an example, we can consider the SSSP algorithm that operates only on the traversed path. Thus, tasks related to partitions that do not contain nodes related to the path will not contribute to finding the correct result and will waste time. In $AsyncP$ instead of scheduling $Gather$ and $Compute$ tasks in a round-robin fashion, the master thread maintains a priority queue. SQLoop updates the priority at the end of each task by scanning the correlated partition. The priority dependents on a query provided by the user. In the case of the PageRank algorithm, the priority of R_{ptx} depends on the sum of rank of its nodes while in SSSP, it depends on the node that has the least distance from the source. Finding a priority function can be difficult and thus, SQLoop uses the user's input to define it.

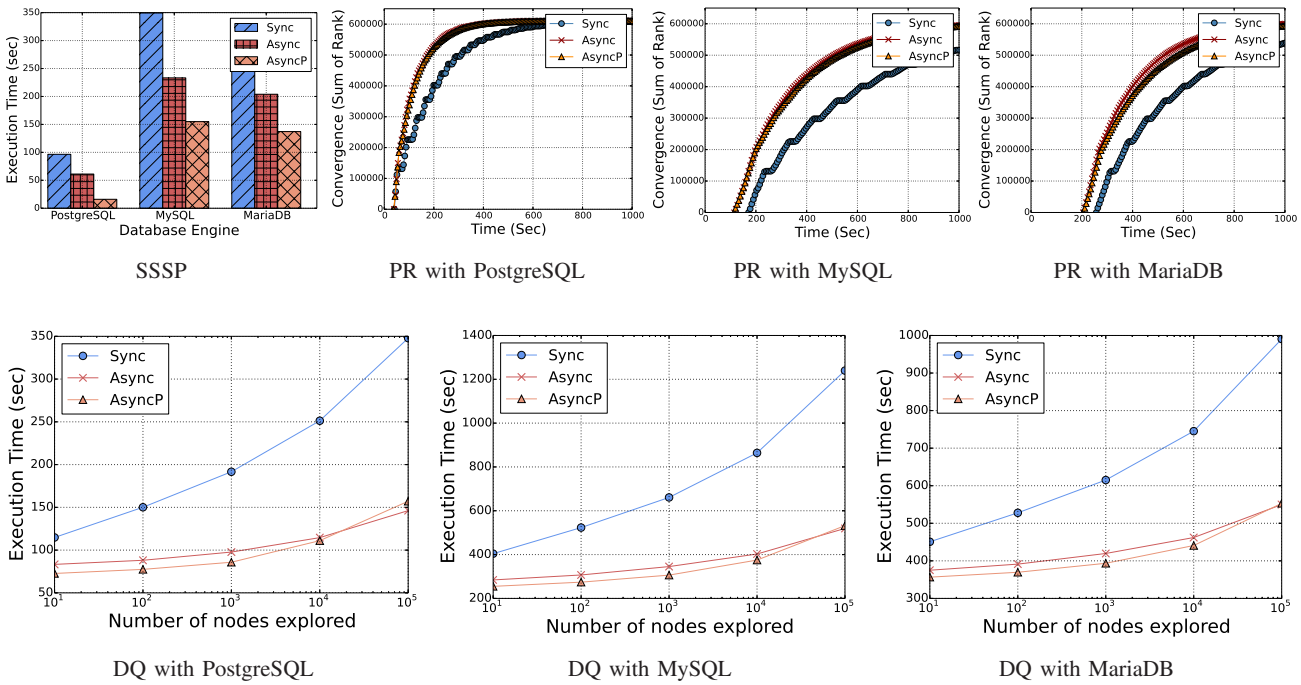
VI. Experimental Studies

In this section we provide an extensive experimental evaluation of SQLoop. First, we evaluate how intermediate results accelerate the computation process (§ VI-B) and then, we explore how the synchronous and asynchronous implementations can utilize multiple cores (§ VI-C). Finally, we compare SQLoop with the alternative solution, using SQL scripts (§ VI-D).

A. Configuration

We report the results using a single machine with 32 cores running at 2.6 GHz (Intel Xeon CPU E5-2650 v2) and 32GB of RAM. The operating system running on the machine is Linux Ubuntu 16.04. We tested SQLoop with PostgreSQL v9.6 and MySQL. To evaluate SQLoop's performance with MySQL, we used both the official MySQL v5.7 system and its other popular version MariaDB v10.2. For PostgreSQL, we set the shared memory buffers to 25% of the total available memory (8GB) following the official recommendation in the website and we used unlogged tables to avoid logging overhead. We also configured the size of the temporary buffers (index related buffers) to 4GB. For MySQL and MariaDB, we used MyISA as the underlining storage engine and configured the buffer sizes as previously.

To evaluate our system, we chose three different queries. The PageRank (PR) query that performs computations in the entire dataset (Example 2), the Single Source Shortest Path (SSSP) algorithm that performs computations only to some parts of the dataset (Example 3) and the Descendant Query (DQ) which answers the question *who is within n friend-hops from a given node* (also used in [33]). Both DQ and SSSP queries are similar in the way that they explore the dataset but different on how they calculate the distance/hops. Moreover, the DQ query returns all the discovered nodes whereas the SSSP query returns only the distance from a single destination. For the PR experiment we used the Google web dataset [34] (5,105,039 rows), for the SSSP experiment we used the Twitter dataset [35] (1,768,149 rows) and for the DQ experiment, we used the Berkeley-Stanford dataset [34] (7,600,595 rows) which contains web-pages from berkely.edu or stanford.edu. For this dataset, the DQ query calculates the number of clicks that the user needs to make in order to go from a given web-page to any other. All datasets are available in [36]. We executed 100 iterations and we report the sum of rank (i.e. convergence progress [11]) and the convergence time for the PR experiment, and the execution time for the SSSP experiment. As convergence time, for PR we define the time that the dataset achieved 99% of the total sum of rank and as execution time for SSSP when the shortest path between source and destination has been computed. To report



SSSP | PR with PostgreSQL | PR with MySQL | PR with MariaDB

DQ with PostgreSQL | DQ with MySQL | DQ with MariaDB

Fig. 4: SQLoop using a single thread

the results, we sampled the entire dataset using a separate thread every 5 seconds. For the DQ query, we sampled the entire dataset every second to report the number of web-pages that have been discovered. For all experiments, we created 256 partitions.

B. Intermediate results

The first experiment tests how the asynchronous implementations of SQLoop can accelerate the computation of the query. SQLoop uses a single thread. As we can see from the plots in Figure 4, the asynchronous implementations outperform the Synchronous Execution for all queries. More detailed, the baseline Asynchronous Execution is $1.5\times - 3\times$ faster than the Synchronous for the PR and DQ queries and the Prioritized Asynchronous Execution up to $3\times$ faster for the SSSP query.

For the PR use case, the main performance benefit comes mainly from the fact that the asynchronous implementations are able to use intermediate results from the current iteration in order to accelerate the convergence. Moreover, we remark that for this particular example the two different asynchronous implementations do not have any performance difference. This happens because we use a real dataset and we partition it using a hash function. Thus, the partitions that are created do not have much difference in their priority. Additionally, the PR query performs computations across the entire dataset, which means that prioritizing one task over the other does not result into a faster convergence.

In comparison, for the SSSP query, the two different asynchronous implementations have a performance difference. This is because during the execution of the query only some partitions are used in the actual computation of the shortest path. These partitions are the ones that contain nodes that are newly discovered and are part of the traversal path of the algorithm. Both asynchronous implementations are able to use intermediate results and thus, execute the query faster than the synchronous one, but prioritizing tasks from partitions that are active in this case, accelerates the computation of the query up to $3\times$.

For the DQ query, the asynchronous implementations are again $1.5\times - 3\times$ faster than the Synchronous Execution. When exploring a small number of pages Prioritized Asynchronous Execution gives a small performance benefit. This happens because only a small number of partitions will participate on the computation. However, as the query explores more web-pages, the performance difference between the Asynchronous Execution and the Prioritized Asynchronous Execution becomes smaller. More and more partitions will need to participate in the query computation and as a result, prioritization becomes irrelevant. Another observation is that as the number of explored nodes becomes larger, the performance difference between the synchronous and the asynchronous implementations becomes bigger. Web-pages that are further way, will need more iterations in order to be discovered, something that accumulates the improvement offered by the intermediate results in each iteration.

In general, the more iterations SQLoop needs to perform in order to calculate the final result, the bigger the performance difference will be between synchronous and asynchronous implementations. Moreover, the asynchronous implementations in most cases are $2\times$ faster than the synchronous one

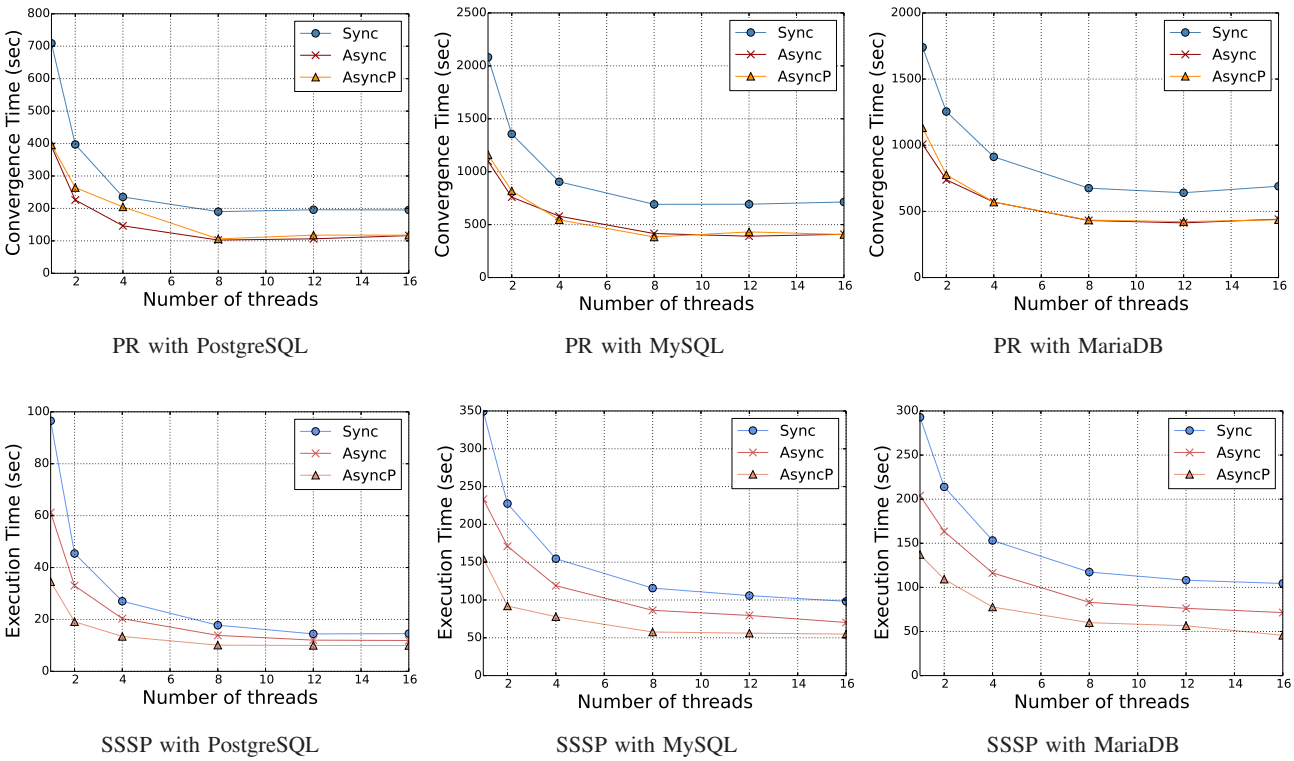


Fig. 5: SQLoop using multiple threads and CPUs

and Prioritized Asynchronous Execution will result into a significant performance improvement only if the iterative part of the query (i.e. the `WHERE` in R^i) eliminates a significant amount of rows in each iteration. Finally, the three different execution techniques follow the same performance pattern for all the different database systems.

C. Performance evaluation on multicores

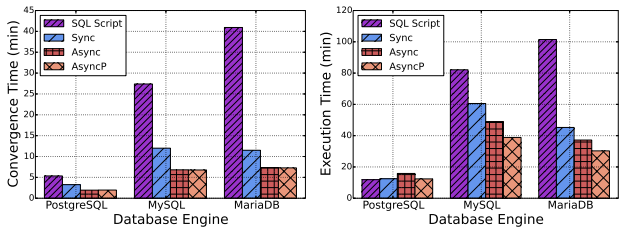
In this section, we test how our system scales up by using multiple threads and CPUs. We change the number of threads used by SQLoop to 1,2,4,8,12 and 16. As we can observe from Figure 5, increasing the number of threads in SQLoop also increases the performance. This happens because each thread is using a new JDBC connection and thus, more $Compute$ and $Gather$ tasks are executed concurrently. Basically, by opening multiple connections and using independent queries and partitions, the target database engine uses more resources to accommodate the same computational needs.

The scale of the performance improvement remains the same as we add more threads to the execution of the query for both synchronous and asynchronous implementations. The machine that we used in order to conduct our experiments has 32 cores but during the execution of the queries we observed that all CPUs were in use while we used 16 threads. This happens because the database systems used 16 different processes to answer incoming queries, SQLoop used another 16 threads to submit the SQL queries, one more thread in SQLoop was responsible for the task scheduling and finally, there was also a last thread that was sampling the dataset in order to observe the convergence. We also highlight the fact that all systems and queries in our experiments were able to significantly improve their performance with PostgreSQL achieving a $10\times$ better response time.

Another significant observation is that the execution time in SQLoop changes according to the query rather than the data input itself. Despite the fact that for the PR query we used a dataset that is approximately $3\times$ bigger than the one in the SSSP query, the execution took almost $6\times$ more time. As in most frameworks that perform iterative data processing, the execution time is more affected by the amount of data that is processed in each iteration rather than the total size of the dataset. In our case, while the SSSP query processes only a few rows in each iteration, the PR query processes the entire dataset. This is why the SSSP query takes only a few seconds in PostgreSQL while the PR query takes a few minutes. Ewen et al [37] also discuss the same problem in iterative processing by naming the two different kind of queries as *Bulk Iterations* and *Incremental Iterations*.

D. Other Comparisons

Finally, we compare SQLoop with the alternative solution of writing SQL scripts. In Figure 6, we report the time SQLoop needs to perform the PR query and the DQ query which in this case, calculates how many clicks away two specific web pages are. We picked two that are 100 clicks away. For both queries, SQLoop uses 16 threads. We also report the time each system



PR DQ

Fig. 6: Comparison of SQL scripts and SQLoop

needs to finish the execution of a SQL script that performs the equivalent computation. SQLoop is able to accelerate the computation up to almost $5\times$ for the PR query and up to $3\times$ for the DQ query. The performance improvements comes for several optimizations that were described in Section V such as the materialization of redundant join operations, the careful formulation of SQL queries and the use of indexes to avoid full scans. However, the biggest performance improvement comes from the fact that SQLoop is able to parallelize the query and use multiple cores. Despite the fact that low-level optimizations are not present, SQLoop is able to deliver high performance across all systems although the improvement varies depending on the underlying database.

Probably the most important point is that SQLoop not only greatly reduces the execution time by using advanced optimizations, but also the time needed by the user to prepare the actual query by bypassing the mundane and counter-productive process of writing complex scripts that are engine dependent. For our examples, SQL scripts in most cases were more than 200 lines, and we also needed to manually change the syntax for some SQL statements. On the contrary, SQLoop queries that are expressed through our proposed iterative CTE extension were composed by only 20-25 lines.

VII. Related work

A. Iterative processing

Iterative processing has been studied in the context of RDBMSs in the past. Zhao and Xu Yu [30] extend the current CTE model to enable graph processing on relational data. They propose the $union - by - update$ operator which enables recursive queries to execute iterative algorithms by updating the table in each recursion. Passing et al. [38] also propose an extension to SQL that enables iterative processing. However, our work is different as it discusses a model that considers different termination conditions, transparent parallelization and enables asynchronous computation. Moreover, Ghazal et al. [39] employ an adaptive technique to improve the performance of Teradata, a system that answers recursive queries by generating iterative query plans. Moreover, recent work also tries to bridge the gap between RDBMSs and imperative programming by converting UDFs [40] or vertex-centric queries [41] to SQL. Finally, optimizing iterative queries has also been studied in the past in the context of the MapReduce framework [32], [33], [42] and array databases [43]. While their optimizations are focused on different frameworks, they all stress out the importance of supporting efficient, high-performance iterative processing.

B. Asynchronous processing

Graph specialized systems have proposed and implemented asynchronous computation for iterative processing. GraphLab [44] aims to express asynchronous iterative algorithms with sparse computational dependencies while ensuring data consistency and achieving good parallel performance. Grace [45] is a single-machine parallel graph processing platform that allows customization of vertex scheduling and message selection to support asynchronous computation. Giraph++ [46] not only allows asynchronous computation while keeping the vertex-centric model but is also able to handle mutation of graphs. GiraphUC [47] relies on barrier-less asynchronous parallel (BAP), which reduces both message staleness and global synchronization. Maiter [11] proposes delta-based asynchronous iterative computation model (DAIC) and supports distributed asynchronous graph processing. GunRock [48] supports fast asynchronous graph computation in GPUs. Naiad [49] also supports iterative and incremental computations. Unfortunately, most of the above systems express queries using a vertex-based API [1] and do not support automatic asynchronization. Ramachandra et al. [50] automatically rewrite database applications in order to take advantage of asynchronous execution but their system transforms Java code and not SQL queries.

VIII. Conclusion

There is a gap between the declarative SQL nature for set/bag-based computations and imperative programming requirements for explicit control flows in iterative computations. Existing CTE-based approach in SQL cannot fundamentally bridge the gap because of its limitation in expressing aggregation-essential algorithms, such as PageRank. Our proposed solution in SQLoop provides a practically workable solution to allow SQL to express iterative computations. It exploits explicit, user-controllable terminating conditions to enhance SQL for general-purpose iterative computations without affecting SQL's declarative programming nature. Furthermore, SQLoop not only contributes to the language/expression level, but also utilizes an auto-mechanism developed to automatically turn iterative queries into asynchronous parallel execution, eliminating a challenging burden to database developers in their programming. In addition, our solution SQLoop creates a common preprocessing platform to enable database engine-independent execution for iterative and recursive queries on different engines.

IX. Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments and suggestions. This work has been partially supported by the National Science Foundation under grants CCF-1513944, CCF-1629403, CCF-1718450 and by the National Natural Science Foundation of China (61672141).

REFERENCES

[1] 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.

[2] 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.

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

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

[5] "Mircosoft SQL server," http://www.microsoft.com/en-us/sql-server, 2017.

[6] PostgreSQL Global Development Group, "PostgreSQL," http://www.postgresql.org, 2017.

[7] "SQLite," http://www.sqlite.org/, 2017.

[8] "MySql," https://www.mysql.com, 2017.

[9] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[10] G. Reese, *Database Programming with JDBC and JAVA*. " O'Reilly Media, Inc.", 2000.

[11] 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.

[12] "MariaDB," https://mariadb.com, 2017.

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

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

[15] 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.

[16] 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.

[17] W. E. Moustafa, V. Papavasileiou, K. Yocum, and A. Deutsch, "Datalography: Scaling datalog graph analytics on graph processing systems."

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] G. Jeh and J. Widom, "Simrank: a measure of structural-context similarity," in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2002, pp. 538–543.

[25] C. Manning, R. PRABHAKAR, and S. HINRICH, "Introduction to information retrieval, volume 1 cambridge university press," *Cambridge, UK*, 2008.

[26] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. D. Sarma, "Finding connected components in map-reduce in logarithmic rounds," in *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE, 2013, pp. 50–61.

[27] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[28] J. Gray *et al.*, "The transaction concept: Virtues and limitations," in *VLDB*, vol. 81. Citeseer, 1981, pp. 144–154.

[29] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ansi sql isolation levels," in *ACM SIGMOD Record*, vol. 24, no. 2. ACM, 1995, pp. 1–10.

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

[31] Y. Yu, P. K. Gunda, and M. Isard, "Distributed aggregation for data-parallel computing: interfaces and implementations," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 247–260.

[32] 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.

[33] 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.

[34] 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.

[35] J. Leskovec and J. J. Mcauley, "Learning to discover social circles in ego networks," in *Advances in neural information processing systems*, 2012, pp. 539–547.

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

[37] 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.

[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] 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.

[40] 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.

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

[42] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM international symposium on high performance distributed computing*. ACM, 2010, pp. 810–818.

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] K. Ramachandra, M. Chavan, R. Guravannavar, and S. Sudarshan, "Program transformations for asynchronous and batched query submission," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 2, pp. 531–544, 2015.