RASQL: A Powerful Language and its System for Big Data Applications

Jin Wang  
UCLA  
jinwang@cs.ucla.edu

Guorui Xiao  
UCLA  
grxiao@ucla.edu

Jiaqi Gu  
UCLA  
gujiqi@cs.ucla.edu

Jiacheng Wu  
Tsinghua University, Beijing  
wu-jc18@mails.tsinghua.edu.cn

Carlo Zaniolo  
UCLA  
zaniolo@cs.ucla.edu

ABSTRACT
There is a growing interest in supporting advanced Big Data applications on distributed data processing platforms. Most of these systems support SQL or its dialect as the query interface due to its portability and declarative nature. However, current SQL standard cannot effectively express advanced analytical queries due to its limitation in supporting recursive queries. In this demonstration, we show that this problem can be resolved via a simple SQL extension that delivers greater expressive power by allowing aggregates in recursion. To this end, we propose the Recursive-aggregate-SQL (RASQL) language and its system on top of Apache Spark to express and execute complex queries and declarative algorithms in many applications, such as graph search and machine learning. With a variety of examples, we will (i) show how complicated analytic queries can be expressed with RASQL; (ii) illustrate formal semantics of the powerful new constructs; and (iii) present a user-friendly interface to interact with the RASQL system and monitor the query results.

ACM Reference Format:

1 INTRODUCTION
The past decades have witnessed a growing number of advanced analytical applications in the booming of Big Data and AI, including online ads recommendation, social network analysis, fraud detection, cloud computing. It calls for efficient languages and systems for analytical queries on large-scale dataset. To address this challenge, researchers from both academia and industry have paid great efforts in building distributed data processing platforms with SQL or its dialect as the query language, such as Spark SQL [1] and SQL++ [5] for AsterixDB.

The popularity of SQL in big data platforms comes from the benefits it provides due to its declarative nature and portability. In fact, queries written in SQL on one platform can be easily translated into another platform that supports SQL interface. In addition to simplifying application development, it would also improve the performance and scalability by taking full advantage of the massive data parallelism technology that relational DBMS has long demonstrated [6].

Currently the ability of SQL to support advanced analytical applications is impaired by its limited ability to support iterative and recursive queries. Current SQL:2003 standard supports recursive queries by introducing the Recursive Common Table Expression (CTE). However, the current SQL standard only supports queries that are stratified w.r.t aggregates and negation. In other words, the recursive computation at one stratum should complete before the aggregates and negation can be applied to the results so produced.

Nevertheless, recent theoretical advances have shown that queries with aggregates in recursion satisfying the PreM (Pre-Mappability) property [8] produce the same results as the stratified program. This enables users to directly specify their optimized query via an unstratified program that (i) addresses the performance and termination issues plaguing the stratified versions; and (ii) also tends to appeal better to programmers’ intuition by summarizing the key optimization steps used in the procedural versions of recursive
queries. Such a promising theoretical results enables the concise expression and efficient support of a rich set of recursive queries, such as graph algorithms, data mining and machine learning [3].

In this demonstration, we propose the Recursive-aggregate-SQL (RaSQL) language and its system by extending the current SQL standard to support aggregates-in-recursion. With the formal guarantee provide by the PreM property, RaSQL is able to express a wide spectrum of analytical queries including graph algorithms and machine learning in a declarative fashion. We implement the RaSQL system on top of Apache Spark by devising effective query compilation and planning techniques. We also provide a user friendly interface to (i) help users gain better understanding of how the theoretical advances allows for the expression of powerful, yet elegant analytical queries and (ii) make it easier for users to interact with the system.

2 THE RASQL LANGUAGE

First we give a brief introduction to the RaSQL language. It introduces a simple extension in the syntax of SQL-2003 standard. Specifically, it allows basic aggregates, i.e. min, max, sum, count, in recursion by extending the Common Table Expressions (CTE) now supported in the current SQL standards. The basic syntax is shown below.

WITH [recursive] VIEW1 (v1_column1, v1_column2, ...) AS (SQL-expression11) UNION (SQL-expression12) ..., [recursive] VIEW2 (v2_column1, v2_column2, ...) AS (SQL-expression21) UNION (SQL-expression22) ... SELECT ... FROM VIEW1 | VIEW2 | ...

WITH RECURSIVE construct of RaSQL

The CTE starts with the keyword "WITH", which is followed by a sequence view definitions. The view content is defined by a union of sub-queries, which define the base table and recursive table. Here a table is a base table if its FROM clause definition does not refer to any recursive CTE; otherwise it is a recursive table which is denoted by the keyword "RECURSIVE". For aggregates used in the CTE, RaSQL adopts the implicit group by rule, i.e. no explicit group by clause is required in each sub-query, and all columns except the aggregates will be treated as the group by columns.

The formal guarantee regarding the semantics of RaSQL is provided by the Pre-Mappability (PreM) property [8]. Its definition is based on viewing an SQL query as function $T(R)$ where $T$ is a the relational algebra expression, and $R$ is the vector of relations used in the expression. Then, the constraint $y$ is said to be Pre-Mappable (PreM) to $T$ when the following condition holds: $y(T(R)) = y(T(y(R)))$. It allows pushing extrema aggregates, i.e., max or min, into recursion while preserving the semantics of the original stratified program.

A property that is useful for recursive queries in dealing with sum and count in recursion is named as Pre-Countable Cardinality (PCC), which was recently introduced for DataLog [7]. This idea can also be generalized to RaSQL to enrich the formal fixpoint semantics. Interested readers are referred to [4] for more theoretical results about RaSQL.

3 APPLICATIONS

3.1 Graph Algorithms

The first category of applications that can be effectively expressed by RaSQL is graph algorithms, which are naturally recursive queries. A typical example is All-Pairs Shortest Path (APSP) algorithm expressed in Query 1.

Query 1 - All-Pairs Shortest Path

```
WITH [recursive] VIEW1 (v1_column1, v1_column2, ...) AS (SQL-expression11) UNION (SQL-expression12) ..., [recursive] VIEW2 (v2_column1, v2_column2, ...) AS (SQL-expression21) UNION (SQL-expression22) ... SELECT ... FROM VIEW1 | VIEW2 | ... 
```

The APSP query returns shortest paths from each source node to all other nodes within the same graph. The graph consists of weighted edges stored in the base table `edge`. The recursive table `path` is initialized with edges in the graph themselves. The paths to other nodes are iteratively computed by joining existing paths with edges that start from the end of these paths. The `min` aggregate is used to select the shortest path. And the output will be the source, destination and the length of shortest path between each pair of nodes. More examples of graph algorithms and other kinds of recursive queries can be found in [4].

3.2 Machine Learning Algorithms

One popular ML applications that can be expressed with RaSQL is the training process with the optimization method of gradient descent (GD). Iterative-convergent machine learning (ML) models like SVM, perceptron, linear regression, logistic regression models, etc. are often trained with Batch Gradient Descent and can be written in a declarative manner as RaSQL queries. To enable the expression, the training data is expressed in a verticalized format `vtraining(ID, Col, Val, Y)`, where `ID` is the unique identifier of each training instance; `Y` is the target label; `Col` and `Val` denotes the dimension and the value along that dimension, respectively.
Query 2 expresses the process of training the Linear Regression model with Batch Gradient Descent using the following steps: First the parameters are initialized as a vector with value 0.01 in all dimensions. Then in the $i^{th}$ iteration, it makes the prediction on all training instances with the model obtained in the $(i-1)^{th}$ iteration. Then the gradient is computed by comparing the predicted results with the true values of all training instances. Finally, it updates the model with such gradient and move on to the next iteration. To describe the training process, we use three recursive tables as follows:

- **model** represents the trained model in verticalized form, where each tuple contains the following three fields: $J$ is the iteration counter; $Col$ is a dimension in the model; and $Param$ is the value of that dimension.
- **gradient** represents the results of gradient computed at each iteration. $G$ is the gradient value of the $Col^{th}$ dimension of the model.
- **predict** represents the intermediate prediction results with the model for each training instance. For each tuple, $YPredict$ is the predicted $y$ value for the training instance with id $ID$.

**Query 2 - Batch Gradient Descent for Linear Regression**

```sql
Base tables: vtraining(ID: int, Col: int, Val: double, Y: double)
WITH recursive model (J, Col, Param) AS
    (SELECT 0, vtraining.Col, 0.01 FROM vtraining)
UNION
    (SELECT 1+m.J, m.Col, m.Param+2.0/n*LR*g.G
     FROM model AS m, gradient AS g
     WHERE m.Col = g.Col and m.J = g.J),
recursive gradient(J, Col, sum() AS G) AS
     FROM vtraining AS t, predict AS p
     WHERE p.ID = t.ID),
recursive predict(J, ID, sum() AS YPredict) AS
    (SELECT m.J, t.ID, m.Param*t.Val
     FROM vtraining AS t, model AS m
     WHERE t.Col = m.Col)),
SELECT * FROM model
```

4 SYSTEM ARCHITECTURE

The overall architecture of RaSQL system is shown in Figure 1. It is built on top of Apache Spark (denoted by red dashed rectangle) and consists of three components:

**Web-based User Interface** We provide a carefully designed user interface that allows users to interact with the RaSQL system. Users can type-in the RaSQL queries as well as experience other advanced features without much effort. After the query is submitted to the RaSQL engine and gets executed, the query results would be returned and displayed on the user interface. More details will be described later in Section 5.

**Query Compilation and Planning** The RaSQL engine is implemented on top of Spark SQL, which supports the latest ANSI-SQL standard and provides a comprehensive set of functions that perform the parsing, analyzing, and planning of SQL queries. However, currently the Spark SQL compiler cannot support recursive queries. To address this issue, we propose novel compilation and query planning techniques. The RaSQL queries will first be compiled into a Recursive Clique driven by a new Recursive Operator which consists of a base plan and a recursive plan by leveraging the techniques from Datalog researches [2]. Then it will further be translated into recursive logical and physical plans, which consists both relational operator supported by Spark SQL and recursive operators proposed by us. We introduce a new fixpoint operator in order to evaluate the recursive plan and identify the termination condition for the recursion.

**Execution Engine** The query plan is then submitted to the execution engine, which is executed by the Apache Spark engine using RDDs and transformation operators. To efficiently support the execution of recursive RaSQL queries, we modify the core of Spark to support the fixpoint operator plan and propose a new data structure setRDD. We also propose several techniques to improve the performance, such as stage combination and partition-aware scheduling, which are explained in the full paper [4].

5 DEMONSTRATION PLAN

The demo is based on the RaSQL web interface and the screen shots are shown in Figure 2. This user interface consists of...
four main components: (i) Query input box (top left); (ii) Result display panel (bottom left); (iii) Configuration dashboard (bottom right); (iv) Query Plan display (top right). For novice users, we provide some predefined examples which can be loaded by simply clicking the name of queries in the bottom right drop-down list. Advanced users are welcome to come up with queries by typing into the input box.

The result display panel demonstrates the query results. The top right panel visualizes the recursive clique plan of the query, which can be generated by clicking the “Analyze” button after input the query. It demonstrates how the recursive RaSQL query is analyzed. Details of operators can be found by clicking the corresponding icons. It would help users to better understand the compilation and planning techniques of RaSQL. The configuration dashboard provides several options for user to setup in order to execute the queries. For example, users need to specify the data source from either local file or HDFS on the cloud and set the arguments (if any).

Our demonstration will consist of two steps:

(i) We will give a brief introduction to the main system functionalities, in which we will exhibit the key components and features of our system. To this end, we will run some simple queries, such as transitive closure, connected components and all-pair shortest path, on small graphs as examples.

(ii) A hands-on step in which the public is invited to directly interact with the system and test its capabilities. Users can write the RaSQL freely by themselves on different kinds of datasets, including both graph data and training data for ML models. We also encourage audience to observe the generated recursive clique and query plans to get an experience of how the recursive queries can be handled by the RaSQL engine.

REFERENCES