RaSQL: Greater Power and Performance for Big Data Analytics with Recursive-aggregate-SQL on Spark
Reproduction

George Mandilaras - DS1190012

December 2020

Abstract

Applications that are based on Social Networks, Internet of Things and Knowledge Graph usually represent their data as graphs. However, even though the graph data model gains popularity, not many systems exploit the advantages of using graphs. In this work, I present RaSQL (Recursive aggregate SQL) that enables the execution of recursive queries using recursive aggregation and other novel techniques, especially useful for graph algorithms and data mining. The system I present extends SparkSQL in order to be able to parse recursive queries and to execute them using recursive aggregation. In the end, I evaluate my implementation against other graph computing systems, and show that RaSQL outperforms them.

1 Introduction

Graphs are an essential data model which advances in our times. Graphs are essential for a plethora of application, such us applications that are based on Social Networks, Internet of Things, Knowledge Graphs like fraud detection, contact tracing, real-time recommendation systems and many others. When the data are highly connected, using graphs as the main data model it can lead to improved performance and to more powerful analytics that other conventional relational models can not provide. However, even though graphs are gaining space and becoming an important data model, there are not many systems that apply on graphs or provide many operations on them. Additionally, many applications that uses graphs, like the ones that utilize Social Networks data, require systems able to process and perform analytics on big data, and even though that there are some systems that provide relatively good performance, we need to invent new techniques and implement novel systems capable of processing large amount of data.

RaSQL [5] (Recursive Aggregate SQL) is a novel system that introduces techniques that enables recursive aggregate queries that applies on graphs. The innovation of this system is that it introduces a way to minimize the intermediate results of a recursive query, when the query applies aggregate functions like *min, max, sum* and *count*. Furthermore, the presented implementation of RaSQL extends the SQL engine of Spark (SparkSQL [1]) with recursive aggregate queries, which are performed distributedly. Consequently, this implementation is applicable on big graphs, containing millions of vertices and edges.

RaSQL, except of just the Spark-based implementation presented in this work, is not just a system but a set of novel ideas for extending the expressive power of SQL. First of all, by enabling aggregate in recursion in SQL queries we allow SQL to be able to able to express a broad range of data analytics queries, especially those commonly used in graph algorithms and data mining. Furthermore, the current SQL standard merely supports stratified aggregation queries (i.e. the aggregation function is applied to the final results) leading to redundant comparisons, that significantly slow down the performance of a query. Moreover, it presents an execution plan for recursive queries based on a *fixed-point operator*.

In this work, I will present you my implementation of RaSQL that extends SparkSQL with a Parser that parses recursive SQL queries into Logical Plans, an Analyzer that resolves the produced logical plans, a Spark planner that transforms these logical plans into physicals, and all the essential data structures and operations for the execution of these plans. Last, I provide detailed evaluation of my model in comparison with two other systems, GraphX and BigDatalog. For information about the code see Section 5.3.

2 Related Work

There are a lot of systems that work well with graph data. The main difference between a Relational Database(RDB) and a graph database(GDB) is that they don't arrange data in rows, columns and tables but they are designed to treat the relationships between data as equally important to the data itself. It is intended to hold data without constricting it to a pre-defined model. Instead, the data is stored like a graph - showing how each individual entity connects with or is related to others. The most well-known graph database is Neo4j[1] which was built to efficiently store, handle, and query highly-connected data in your data model. Neo4j is an open-source, NoSQL, native graph database that provides an ACID-compliant transactional back-end. As its declarative query language it uses Cypher [3] which is similar to SQL but more optimized for graphs.

Similar to graph databases are the triple-stores that are designed to store data that follows the RDF data model. RDF (Resource Description Framework) is a data model designed for interlinking pages and entities within the Web, but it eventually became a model for generally representing linked data. In RDF, data is stored into triples consisting of a subject, a predicate and an object. The subject is a unique URI that describes an instance, object is a property and the predicate expresses the relation between subject and object. RDF triples are considered as an alternative way of representing graphs and it is widely used for Knowledge Graph and more generally in the field of Semantic Web. SPARQL [2] is the main query language for querying RDF data and it is a very powerful query language as it can leverage the power of using ontologies and hence description logic. Probably the most famous RDF triple-stores are Virtuoso [2] and GraphDB [6].

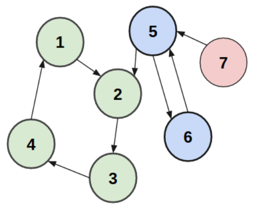
Even though the systems I mentioned work well with graph data, they don't natively support recursion in their queries and they are not designed to run in distributed environments in order to process big graph data. A system that satisfies both these properties is BigDatalog [7] which is a system developed on-top of Apache Spark and uses Datalog as its query language to support data analytics on distributed datasets. BigDatalog implements most of the heuristics presented in RaSQL and many more different types of Recursion.

[1]`https://neo4j.com/`

[2]`https://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`

```
Base Table: edge(Src, Dst)
WITH recursive cc(Src, min AS CmpId) AS
(SELECT Src, Src FROM edge) UNION
(SELECT edge.Dst, cc.CmpId
FROM cc, edge
WHERE cc.Src = edge.Dst)
SELECT count(distinct cc.CmpId) FROM cc
```

(a) Recursive Query: Count all Connected Components



(b) Connected Components of a graph, each colour indicates a different component

Figure 1: Connected Components (CC) Example

Last, few words about Spark. Apache Spark [9] is an open-source cluster-computing framework that uses a master/worker architecture. There is a Driver that talks to a single coordinator called master which manages worker in which Executors run. Driver is responsible to split the job into tasks, to schedule them to run on Executors and to coordinate the overall execution. Executors are distributed agents that execute the tasks in parallel (or sequentially). At the core of Apache Spark is the notion of data abstraction as a distributed collection of objects, known as Resilient Distributed Dataset (RDD) [8]. RDDs allow the user to apply a series of transformations (i.e. map, filters, etc), creating a lineage graph which will not be executed before calling an action (i.e. count, write to file, etc). All transformations and actions are performed in parallel, as each Executor is assigned with portion of the overall data known as partition and the execution of the transformation linkage graph run inside a task. The number of concurrent tasks is configurable and it is defined by the user and the available resources.

Additionally, GraphX [4] is a component of Spark, developed for graph-parallel computation. At a high level, GraphX extends the Spark RDD by introducing a new Graph RDD and as well as an optimized variant of the Pregel API abstraction, including a growing collection of graph algorithms and builders to simplify graph analytics tasks.

3 Dive in RaSQL

In this section, I will present the main parts of the original paper.

3.1 Recursive Queries

With recursive queries, we mean queries that define a *CTE* (Common Table Expression) which is build by recursively applying the query that constructs it, until it reaches a fix-point, for instance until no new records are discovered. A simple example of such query is the Connected Component query (see Figure 4a), that applies the algorithm for discovering the connected components of a graph. A connected component or simply component of an undirected graph is a sub-graph in which each pair of nodes is connected with each other via a path.

First, the query requires the table *edge* to be registered in DB. The query starts with an expression that defines the name and schema of the produced table and then follows the queries which will construct it.

This declarative expression always starts with a $With$ keyword which instructs the parser to expect a CTE. In this particular example, the query contains an aggregation function, and using this peculiar syntax we inform the parser that this is a RaSQL query and hence it will use aggregate-in-recursion and a non-stratified execution.

Afterwards, two $SELECT$ queries follow. The first one is called the *base*, which is executed first and initializes the recursive table (**Note**: do not confuse it with base relation, base query initializes the recursive table while the base relation is the relation with which we join, i.e. *edge*). Then follows the recursive query which is performed recursively, producing new records which we add to the recursive table. When we reach the fix-point (e.g. no new records were discovered), the recursion ends, and we have completed the recursive table. Finally we then execute the last query on the new table, where in the CC query counts the distinct components id.

3.2 Pre-Mapability

A great advantage of RaSQL derives from the Pre-Mapability ($PreM$) property, which when it's applied, reduces the size of the intermediate results of a join. $PreM$ has long been recognized and used as a cornerstone for parallel databases and MapReduce. $PreM$ property is defined as:

Consider a function $T(R_1, R_2, ...R_k)$,
we say that a constraint γ is PreM to $T(R_1, R_2, ...R_k)$
when the following property holds:

$$\gamma(T(R_1, R_2, ...R_k)) = \gamma((\gamma(R_1), \gamma(R_2), ...\gamma(R_1)))$$

For instance, if T is union and γ is max, then we have:

$$max(R_1 \cup R_2 \cup ... \cup R_k) = max(max(R_1) \cup max(R_2) \cup ... \cup max(R_k))$$

The advantages of $PreM$ becomes clear when we use unions and joins as it reduces the load of the input tables. Furthermore, its benefits are even more distinctive when we work in distributed environment, as each node is able to compute locally the requested results of its subparts which will then be reduced in order to generate the final results. This way we avoid not only unnecessary comparisons, but also it decreases the size of shuffling data which plays a very important role in parallel programming, as Network I/Os can significantly deteriorate the performance of a system. Some widely used aggregation functions that are $PreM$ are *min*, *max*, *sum* and *count*.

3.3 Aggregation in Recursion

The idea of aggregate in recursion is quite simple and it is displayed in Algorithm 2. First we initialize the recursive table based on the base query, which in Algorithm 2 is *cc*. Then, by performing the aggregation with the base table, we compute new records and we store them in cc_{new}. In order to avoid adding duplicate records in our results, we differentiate the newly computed elements with the existing, in order to get the actual new records, which are then added into the recursive table. The iteration restarts by initializing the δ *cc* with the new distinct records, and it finishes when it reaches to a fix-point, which in the CC example is when no new records are added to the set.

Algorithm 1 Recursive Aggregation in CC

```
1: $\delta$cc $\leftarrow$ $edge(Src, Src)$
2: $cc \leftarrow \delta$cc
3: **do**
4:     $cc_{new} \leftarrow \Pi_{cc.Src\ =\ edge.Src}(\delta\text{cc}(Src, CmpId) \bowtie edge(Src, Dst))$
5:     $\delta\ cc' \leftarrow cc_{new} - cc$
6:     $cc \leftarrow cc \cup cc'$
7:     $\delta\ cc \leftarrow \delta\ cc$'
8: **while** $\delta\ cc = \emptyset$
```

In a MapReduce environment, this process can be parallelized but it requires some special cases in order to avoid additional shuffling. First, we join and apply the transformations using a Map stage to all the partitions that co-exist in the same nodes, and then using a reduce stage we shuffle the data around in order to compute the overall results of the iteration. Ideally, if the aggregation key is the same with the reduce key, we can avoid data shuffling by implementing the whole procedure partition-wise, as each partition will contain all the necessary records. Therefore we would be able to perform both join and the aggregation for each partition individually without shuffling data, and then to apply a global aggregation in order to produce the final results. Unfortunately, in most cases like in CC, the aggregation key is different from the reduce key (in CC the aggregation key is Src while the reduce key of the table edge is Dst which becomes Src, see Figure 4a).

4 From Queries to Spark Plan

In this section, I will start introducing some parts of my work. In more details, I describe the process of generating a Spark Plan given a query. The whole procedure consist of three main steps. First comes the parser which parses the input query producing a Logical Plan. Then the Analyzer analyzes this plan producing a resolved Logical Plan. Last, the Spark Planner reads the resolved plans and produces the execution workflow, known as Spark Plan. Generally, between Analyzer and Spark Planner, there is also an optimization step, which I will omit as I did not introduce any optimization technique. The whole procedure is coordinated by the *QueryExecution* class, which is initialized by Spark's SQL context. For more information for the execution pipeline of a query in SparkSQL, I recommend you to read the documentation of the *QueryExecution* class.

As I already mentioned, the input query gets to the parser, which I call $RaSQLParser$. $RaSQLParser$ is based on the scala-parser-combinators[3] library that Apache Spark[4] uses for parsing the input queries. The parser uses pattern matching in order to match the input query into pre-defined accepted patterns, and execute the corresponding code. This way, parts of the query (e.g. the base query) are matched and processed by a specific code-block. The main goal of $RaSQLParser$ is to create the default logical plan for each sub-query, and then for each one (the recursive and base query) to create partial aggregates that partially apply the aggregation function and to pass it to a *RecursiveAggregate* plan. The *RecursiveAggregate* plan indicates to create *RecursiveAggregate* spark plan, where certain code will be executed. The results of the *RecursiveAggregate* is forwarded to a final aggregation that applies the aggregate function to the overall results.

[3]`scala-parser-combinators`

[4]Apache Spark uses this library in the version 1.6, which also my whole implementation is based on.

```
'With Map(cc -> 'Subquery cc
+- 'Aggregate ['Src], ['Src,unresolvedalias((mmin('CmpId),mode=Complete,isDistinct=true) AS CmpId#27)]
   +- 'Subquery cc_RA
      +- 'RecursiveAggregate cc
         :- 'Aggregate ['Src], ['Src,unresolvedalias((mmin(unresolvedalias('Src AS CmpId#21)),mode=Partial,isDistinct=true) AS CmpId#26)]
         :  +- 'Project [unresolvedalias('Src AS Src#20),unresolvedalias('Src AS CmpId#21)]
         :     +- 'UnresolvedRelation `edge`, None
         +- 'Aggregate ['Src], ['Src,unresolvedalias((mmin('CmpId),mode=Partial,isDistinct=true) AS CmpId#25)]
            +- 'Subquery RR
               +- 'Project [unresolvedalias('edge.Dst AS Src#22),unresolvedalias('cc.CmpId AS CmpId#23)]
                  +- 'Join Inner, Some(('cc.Src = 'edge.Src))
                     :- 'CacheHint
                     :  +- 'UnresolvedRelation `edge`, None
                     +- Subquery cc
                        +- RecursiveRelation `cc`, [Src#14,CmpId#15]
)
+- 'Project [unresolvedalias((count('cc.CmpId),mode=Complete,isDistinct=true) AS Src#24)]
   +- 'UnresolvedRelation `cc`, None
```

Figure 2: Logical Plan Produced for the CC query

```
+- TungstenAggregate(key=[], functions=[(count(if ((gid#34 = 1)) CmpId#35 else null),mode=Partial,isDistinct=false)], output=[count#67L])
   +- TungstenAggregate(key=[CmpId#35,gid#34], functions=[], output=[CmpId#35,gid#34])
      +- TungstenAggregate(key=[CmpId#35,gid#34], functions=[], output=[CmpId#35,gid#34])
         +- Expand [List(CmpId#27, 1)], [CmpId#35,gid#34]
            +- Project [CmpId#27]
               +- TungstenAggregate(key=[Src#22], functions=[(mmin(if ((gid#32 = 1)) CmpId#33 else null),mode=Final,isDistinct=false)], output=[CmpId#27])
                  +- TungstenAggregate(key=[Src#22], functions=[(mmin(if ((gid#32 = 1)) CmpId#33 else null),mode=Partial,isDistinct=false)], output=[Src#22,mmin#65])
                     +- TungstenAggregate(key=[Src#22,CmpId#33,gid#32], functions=[], output=[Src#22,CmpId#33,gid#32])
                        +- TungstenAggregate(key=[Src#22,CmpId#33,gid#32], functions=[], output=[Src#22,CmpId#33,gid#32])
                           +- Expand [List(Src#22, CmpId#25, 1)], [Src#22,CmpId#33,gid#32]
                              +- RecursiveAggregate cc
                                 :- TungstenAggregate(key=[Src#20], functions=[(mmin(if ((gid#28 = 1)) Src AS CmpId#21#29 else null),mode=Final,isDistinct=false)], output=[Src#20,CmpId#26])
                                 :  +- TungstenAggregate(key=[Src#20], functions=[(mmin(if ((gid#28 = 1)) Src AS CmpId#21#29 else null),mode=Partial,isDistinct=false)], output=[Src#20,mmin#60])
                                 :     +- TungstenAggregate(key=[Src#20,Src AS CmpId#21#29,gid#28], functions=[], output=[Src#20,Src AS CmpId#21#29,gid#28])
                                 :        +- TungstenAggregate(key=[Src#20,Src AS CmpId#21#29,gid#28], functions=[], output=[Src#20,Src AS CmpId#21#29,gid#28])
                                 :           +- Expand [List(Src#20, Src#20, 1)], [Src#20,Src AS CmpId#21#29,gid#28]
                                 :              +- Project [Src#2 AS Src#20]
                                 :                 +- InMemoryColumnarTableScan [Src#2], InMemoryRelation [Src#2,Dst#3], true, 10000, StorageLevel(true, true, false, true, 1),
                                    TungstenExchange RoundRobinPartitioning(12), None, None
                                 +- TungstenAggregate(key=[Src#22], functions=[(mmin(if ((gid#30 = 1)) CmpId#31 else null),mode=Final,isDistinct=false)], output=[Src#22,CmpId#25])
```

Figure 3: Spark Plan Produced for the CC query

Figure 2 shows the logical plan produced by the parser. There are two partial aggregation, one which happening once for the base query, and another that occurs in each iteration, both plans are children of *RecursiveAggregate*. Is important to mention that the join is happening between the base relation and the recursive relation, which is not yet initialized therefore the Analyzer will not be able to detect it and it will prevent the execution. In order to avoid that, I initialize an empty Dataframe (that works as a table view) so it can be resolved. Furthermore, I would also like to mention that unlike the recursive relation which get updated in every iteration, the base relation (the *edge* table) remains the same. So, in order to avoid re-executing this plan, I have added a cache hint that indicates the Spark plan to cache it in order to avoid re-computations.

After the parser, follows the Analyzer that resolves the logical plan. This is happening by recursively applying certain rules that resolve the plan, like attribute resolve, function resolve, aliases resolve and many mores. In my implementation, I have extended Spark's default Analyzer by overriding certain functions in order to resolve my logical plans. In more details the rules that I needed to override in order my plans to pass the Analyzer, is the Relation Resolver and the Aliases-related Resolvers. Even though there is not much to say about this part, this was one of the hardest parts of my work as it required many changes in the parser and in the Analyzer in order my logical plans to pass the check analysis.

The last step before the execution is the construction of the Physical Plan, also known as Spark Plan. For this purpose, Spark uses *SparkStrategies* which is an abstract query planner that merely serves as a "container" (or a namespace) of the concrete execution planning strategies, such as aggregation, join selections etc. Hence, in my planner I extend the strategies by adding two new strategies that simply identify my logical plans and point for execution to the corresponding execution plans. The first strategy is for identifying the *RecursiveRelation* and the *RecursiveAggregate*. I use the second one in order to indicate Spark to not use the default join method which is a Sort-based join, but my join which is a shuffle hash join[5]. This is an old join that was removed in the recent versions of Spark as it tends to shuffle a lot. However, in RaSQL we focus on a partition-wise execution, where most of data will remain in the same partitions based on the partition key. Furthermore, in this strategy and inside the code that executes the join, I cache the base relation as indicated by the logical plan.

Figure 3 depicts the Spark Plan produced for the CC query. As you can see, most of the plans are Tungsten[6] which is an engine that aims to improve the execution of certain spark plans in terms of CPU time, by adjusting on the hardware Spark runs on.

5 Recursive Aggregate

In this section, I describe the my implementation of Recursive aggregate and the necessary data structures for the implementation.

5.1 Data Structure

As mentioned in Section 3.3, the data structures of the main variables need to be able to perform set operations, like union and differentiation. Moreover, since we are working with Spark, those structures must be based on RDDs. Hence the paper proposes an extended type of RDD which it calls *SetRDD*. As described in the paper, each partition of this RDD will not be considered as an array but as a HashSet, in order to enable fast set operation and to always store only distinct values. The same structure is also used in BigDatalog, so I borrowed it. However, I realised that this HashSet-based implementation used in BigDatalog, does not fit to the needs of my algorithm, so I changed the inner structure of the SetRDD. Therefore, the SetRDD class that I borrowed from BigDatalog works as just a wrapper that calls my functions of my new Inner data structure.

As shown in Algorithm 2 the data structure must not only be able to contain distinct values, but also to be able to perform fast updates. Therefore the described data structure is not a HashSet but a HashMap, and hence in my implementation the partitions of SetRDD are represented by a class I call $InnerHashMap$ that contains a HashMap. So, consider that two partitions are co-partitioned and we want to find the distinct new values, so we first differentiate them in order to find the new values and then insert them to the main R relation. However the results of the differentiation must not be just the new keys, but also the pairs where the values of the keys are "better" than the existing ones. So, I insert a record if the key does not exist, but also in case that the key does exist, I insert it if the value satisfies the aggregation function (i.e. if the new value of the key is greater than the existing one, in case the aggregation function is max). Then in the insertion, I update the existing value with the more suitable ones, as exactly is explained by the Algorithm 2.

Consequently, the SetRDD class which I borrowed from the BigDatalog works just as a plain wrapper and all the important functionality occurs inside $InnerHashMap$. Furthermore, there is also an $InnerIterator$ class in order to fast and safely transform a SetRDD into a RDD of Rows.

[5]https://www.waitingforcode.com/apache-spark-sql/shuffle-join-spark-sql/read
[6]https://jaceklaskowski.gitbooks.io/mastering-spark-sql/content/spark-sql-tungsten.html

Algorithm 2 Map/Reduce Stage w/ Max Aggregate

```
1: function MapStage($\delta\ R, B$)
2: Require $\delta\ R, B$ co-partitioned on join key $K$
3: for each partition pair of ($\delta\ R, B$ do
4:     $P \leftarrow \Pi_{k,v}(\delta\ R \bowtie_{\delta\ R.K=B.K} B)$
5:     emit Partial_Aggregate(P, func=max, key=k)
6: end for
7:
8: function ReduceStage($\delta\ R', R$)
9: Require $\delta\ R', R$ co-partitioned on join key $K$
10: for each partition pair of ($\delta\ R', R$) do
11:     for each partial aggregated $(k, v)$ do
12:         if $k$ not in $R.keys$ then
13:             put $(k, v)$ in $R$, add to $D$
14:         else if $v > R(v)$ then
15:             $update(k, v)$ in $R$, add to $D$
16:         end if
17:         emit $D$
```

5.2 Implementation

In recursive aggregate the main variables are: R which I will call *allRDD*, $\delta\ R$ which I will call *delta*, and the $\delta\ R'$ which I will call *delta_*. The first step of *RecursiveAggregate* is to execute the base query (i.e. the left plan) in order to initialize the recursive relation. The resulted RDD is transformed in a SetRDD and saved as *allRDD*. Additionally, I also save it in a relation catalog of the RaSQL context, in order to be used in the next iteration. Then by calling the function *doRecursion* the iterations start.

In *doRecursion*, first we execute the recursive query (i.e. the right plan) in order to discover the new records. This plan requires the recursive relation, so in the Spark plan of the *RecursiveRelation* I fetch from the relation catalog of the RaSQL context the RDD I saved after executing the base query. After completing the execution of the recursive query, I transform the resulted RDD into a SetRDD and save it as *delta_*. Then, by differentiating with the *allRDD* I find the new key, values pairs that must be added, and store it in *delta*. Then update the *allRDD* by uniting it with *delta*. Last, I update the RDD saved in the relation catalog of RaSQL context with the RDD of *delta_*, so it will be updated in the next iteration and then start the next iteration.

The iterations stop when no new entities are discovered, hence when $delta = \emptyset$. The result of recursive aggregate is in *allRDD* which I globally aggregate it for a last time, as the definition of PreM specifies. Note that in order to make the partitions to be co-partitioned, the last step of each plan is a partial aggregation that aggregates based on the aggregation key. However this does not necessary mean that all the records with the same key will be in the same partition, but most of them will be.

If I could further extend my implementation, I would make the join operation to be partition-wise. I tried very hard to make such an implementation, but I failed because it was messing the bytes of the records for reasons yet unknown to me.

5.3 Implementation Details

The whole implementation was build with Scala 2.10 and Spark 1.6. The main reason I chose this old version of Spark, is because I started by reading the codebase of BigDatalog in order to understand the execution pipeline of a SparkSQL query. Then, I tried to implement it on Spark 2.4 but it was completely different from what I had become familiar with. Generally the whole implementation was quite hard as it was very difficult to understand the inner procedures of Spark, as most of the things are happening recursively, making hard to follow the process using debugger, and most of inner structures contain the values in bytes and hence they were not visible by the debugger. However, despite the difficulties of this project it was very educative about how and SQL engine works.

You can find my code in this repository. The code is in the folder

`sql/rasql`

and in

`/examples/src/main/scala/org/apache/spark/examples/sql/rasql/`

In the repository you will also find detailed instructions for building and re-producing the experiments.

6 Evaluation

```
Base Table: edge(Src, Dst, Cost)
WITH recursive path(Src, min AS Cost) AS
(SELECT 1, 0) UNION
(SELECT edge.Dst, path.Cost + edge.Cost
FROM path, edge
WHERE path.Dst = edge.Src)
SELECT Dst, Cost FROM path
```

(a) Single-Source Shortest Paths (SSSP): Find all shortest paths from a given source

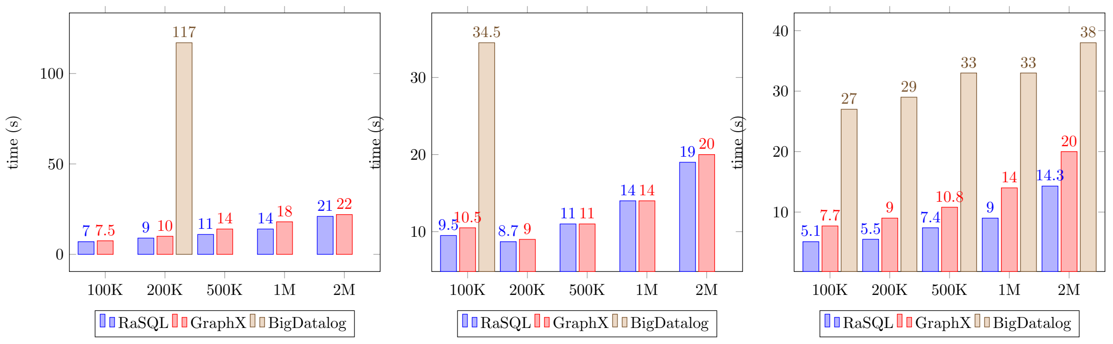
```
Base Table: edge(Src, Dst)
WITH recursive reach(Dst) AS
(SELECT 1) UNION
(SELECT edge.Dst
FROM reach
WHERE reach.Dst = edge.Src)
SELECT Dst FROM path
```

(b) REACH: Find all reachable vertices from given source

Figure 4: Recursive Queries used in the Experiments

In order to evaluate my system I compared its performance against GraphX and BigDatalog using three popular graph algorithms, with varying input graph sizes. In order to generate graphs with varying number of vertices and edges, I used the PaRMAT[7] graph generator and generated graphs with million of vertices and edges. The algorithms that I used are the Connected Components (CC) which I have already explained, the Single-Source Shortest Path (SSSP) which finds all the shortest paths from a given source vertex, and the REACH which implements a Depth-First-Search (DFS) in order to discover all the reachable vertices from a given source. Figure 4 shows the recursive queries that implement these two algorithms, but also shows the potential of the *RaSQLParser* which is able to parse complex queries that follow the pre-defined structure of a recursive query. Regarding the execution of GraphX, even though there are some ready implementations for the CC and SSSP, they consider the graph to be undirected and there is no way to instruct it otherwise. So, for all three

[7]`https://github.com/farkhor/PaRMAT`



(a) Experiment with CC **(b)** Experiment with SSSP **(c)** Experiment with REACH

Figure 5: Experiments I

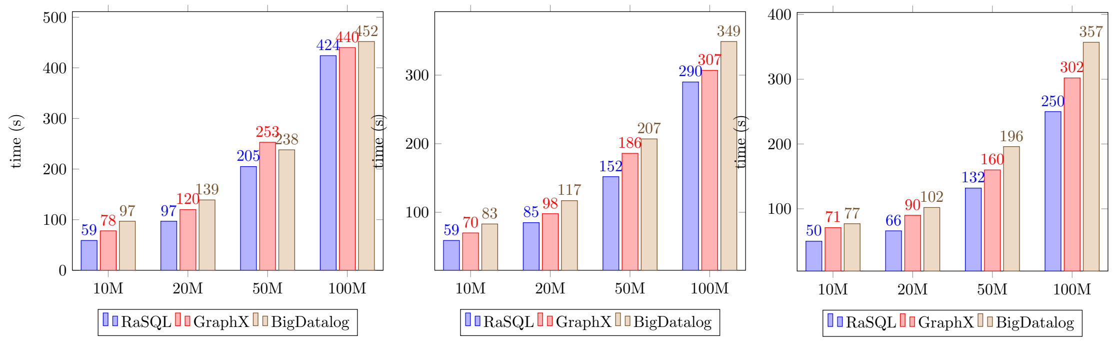
algorithms I developed their implementation using GraphX's pregel operation, which you can find in the *GraphXExperiments* class int he `examples` folder.

The time of each experiment was measured using the `time` command of terminal and the presented time is the average after two executions. The generated graphs contain the same number of vertices as edges (i.e. the 2M graph contains 2M vertices and 2M edges).

The first experiments were performed in my personal computer which contains a Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz (12 cores) and 8GB of memory. In all experiments, the input load was divided into 12 partitions (as the number of available CPU cores) and were processed concurrently. The results are displayed in Figure 5. The main issue with BigDatalog is that in all the experiments that it is absent, it crashed due to memory errors, despite I had configured it with 4GB of memory and `spark.datalog.recursion.memorycheckpoint=false` which is suggested by the developers for more fault-tolerant execution. Regarding the comparison between RaSQL and GraphX, surprisingly they perform quite similarly, with RaSQL to slightly to outperform GraphX in most cases. This was unexpected to me because I have not implemented much of the optimization and scheduling techniques presented in the paper and GraphX is a highly optimized library for parallel graph computations that contains partition techniques based on both vertices and edges. However, the actual performance of RaSQL as shown in the paper is much better, but this is reasonable as I have not implemented the paper at its full extend.

Regarding the second experiments, luckily I granted access to a server of our team in the University of Athens, and I was able to perform large scale experiments. This server contains a Intel(R) Xeon(R) CPU E5-4603 v2 @ 2.20GHz with 32[8] CPU cores and 124GB of memory. As a result I was able to perform experiments with graphs containing up to 100M of vertices and edges. The results are presented in Figure 6, and you can see that RaSQL and GraphX performed similarly as in the previous experiments. In contrast, BigDatalog performs better, and this probably due to the abundance of available memory. Probably in the experiment in Figure **??** there was a lot of Garbage Collector activity, which probably slowed down the whole procedure.

[8]`themachineuseshyper-threading,hence16physicalcores,32virtual`



(a) Experiment with CC **(b)** Experiment with SSSP **(c)** Experiment with REACH

Figure 6: Experiments II

7 Conclusion

In this work I presented my implementation of RaSQL as an extension of SparkSQL that enables aggregate in recursion, applicable especially for graph algorithms. In more details I describe execution pipeline for an input recursive query and how from a logical plan I create Spark plans. Furthermore I describe the algorithm that implements the recursive aggregation and provide detailed information about the necessary data structures.

If I could try to make something differently, I would try to make the whole procedure to work with SetRDDs and most importantly the partial aggregation. This way the aggregation would be performed partition-wised and as a result it would avoid unnecessary data shuffling. However this is quite difficult as it would require to intervene and change most of the intermediate steps in order to work with SetRDDs and to preserve the partitioning scheme. This is something I tried to do, but failed.

As future work, I believe it would be good to move this work into a more updated version of Apache Spark, and to enrich it graph-oriented partitioning schemes like the ones presented in the publication of GraphX [4]. Furthermore to increase the popularity of RaSQL by implementing it into other DBMSs like PostgresDB, MariaDB, etc.

References

[1] Michael Armbrust et al. “Spark SQL: Relational Data Processing in Spark”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives. ACM, 2015, pp. 1383–1394. DOI: `10.1145/2723372.2742797`. URL: `https://doi.org/10.1145/2723372.2742797`.

[2] Orri Erling and Ivan Mikhailov. “RDF Support in the Virtuoso DBMS”. In: *The Social Semantic Web 2007, Proceedings of the 1st Conference on Social Semantic Web (CSSW), September 26-28, 2007, Leipzig, Germany*. Ed. by Sören Auer et al. Vol. P-113. LNI. GI, 2007, pp. 59–68. URL: `https://dl.gi.de/20.500.12116/22407`.

[3] Nadime Francis et al. “Cypher: An Evolving Query Language for Property Graphs”. In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. Ed. by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 1433–1445. DOI: `10.1145/3183713.3190657`. URL: `https://doi.org/10.1145/3183713.3190657`.

[4] Joseph E. Gonzalez et al. “GraphX: Graph Processing in a Distributed Dataflow Framework”. In: *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI ’14, Broomfield, CO, USA, October 6-8, 2014*. Ed. by Jason Flinn and Hank Levy. USENIX Association, 2014, pp. 599–613. URL: `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez`.

[5] Jiaqi Gu et al. “RaSQL: Greater Power and Performance for Big Data Analytics with Recursive-aggregate-SQL on Spark”. In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. Ed. by Peter A. Boncz et al. ACM, 2019, pp. 467–484. DOI: `10.1145/3299869.3324959`. URL: `https://doi.org/10.1145/3299869.3324959`.

[6] Ralf Hartmut Güting. “GraphDB: Modeling and Querying Graphs in Databases”. In: *VLDB’94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*. Ed. by Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo. Morgan Kaufmann, 1994, pp. 297–308. URL: `http://www.vldb.org/conf/1994/P297.PDF`.

[7] Alexander Shkapsky et al. “Big Data Analytics with Datalog Queries on Spark”. In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. Ed. by Fatma Özcan, Georgia Koutrika, and Sam Madden. ACM, 2016, pp. 1135–1149. DOI: `10.1145/2882903.2915229`. URL: `https://doi.org/10.1145/2882903.2915229`.

[8] Matei Zaharia et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*. Ed. by Steven D. Gribble and Dina Katabi. USENIX Association, 2012, pp. 15–28. URL: `https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia`.

[9] Matei Zaharia et al. “Spark: Cluster Computing with Working Sets”. In: *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud’10, Boston, MA, USA, June 22, 2010*. Ed. by Erich M. Nahum and Dongyan Xu. USENIX Association, 2010. URL: `https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets`.