Poster Paper

Efficient Implementation of Joins over Cassandra DBs

Haridimos Kondylakis
FORTH-Inst. of Computer Science
N. Plastira 100, 700 13 Heraklion,
Crete, Greece
kondylak@ics.forth.gr

Antonis Fountouris
Computer Science Department
University of Crete,
700 13 Heraklion, Greece
afountour@gmail.com

Dimitris Plexousakis
FORTH-Inst. of Computer Science
N. Plastira 100, 700 13 Heraklion,
Crete, Greece
dp@ics.forth.gr

ABSTRACT

NoSQL databases provide new opportunities by enabling elastic scaling, fault tolerance, high availability and schema flexibility. Despite these benefits, their limitations in the flexibility of query mechanisms impose a real barrier for any application that has not predetermined access use-cases. One of the main reasons for this bottleneck is that NoSQL databases do not support joins. In this poster we present a solution that efficiently supports joins over such databases. More specifically, we present a query optimization and execution module placed on top of Cassandra clusters that is able to efficiently combine information stored in different column-families. Our preliminary evaluation demonstrates the feasibility of our solution and the advantages gained when compared to a recent commercial solution by DataStax. To the best of our knowledge our approach is the first and the only available open source solution allowing joins over NoSQL Cassandra databases.

1. INTRODUCTION

During the latest years, the explosive growth of data and the emerging requirements for big data management solutions led to the development of NoSQL databases. Among the reasons for the rapid adoption of NoSQL databases is that they scale across a large number of servers by horizontal partitioning of data items, they are fault tolerant and achieve high write throughput, low read latencies and schema flexibility. To achieve all these benefits, the main idea is that you have to denormalize your data model and avoid costly operations in order to speed up the database engine. As such, the NoSQL databases were initially designed to support only single-table queries and explicitly excluded the support for join operations allowing applications to implement such tasks. However, modern applications increasingly require the efficient combination of information from multiple tables and column-families.

To this direction the first approaches are starting to emerge for operators similar to join, based on Map-Reduce such as rank-join queries [1] and set-similarity joins [2]. Rank-join queries try to find the most relevant documents for two or more keywords whereas set-similarity joins are those that try to find similar pairs of records instead of exact ones. However, both these approaches execute joins at the application level using Map-Reduce implementations and the joins implemented do not focus on an exact matching of the joined tuples. This emerging need has also been recently recognized by DataStax, the biggest vendor of Cassandra NoSQL commercial products which recently introduced a commercial join-capable ODBC driver. The company claims that Cassandra can now perform joins just as well as relational database management systems. However, no results were presented nor the specific join implementation algorithms and optimization techniques..

To fill these gaps, in this poster we present a naïve, yet efficient query optimization and execution module enabling joins over Cassandra NoSQL databases surpassing DataStax's commercial solution and highlighting the differences between NoSQL and relational solutions.

This work was partially support by the iManageCancer (H2020-643529) and the MyHealthAvatar (FP7-600929) EU projects.

2. PRELIMINARIES

Cassandra is a NoSQL database developed by the Apache Software Foundation. It uses a hybrid model between key-value and column-oriented database. The structure of the database is defined by super-columns and column-families. In this paper the term column-family and table will be used interchangeably although they are not exactly the same.

All stored data can be easily manipulated using the Cassandra Query Language (CQL) which is based on the widely used SQL. CQL can be thought of as an SQL fragment with the following restrictions over the classical SQL:

- *R1*. Joins are now allowed.
- *R2*. You cannot project the value of a column without selecting first the key of the column. Every select query requires that you restrict all partition keys. Select queries restricting a clustering key have to restrict all the previous clustering keys in order. Queries that don't restrict all partition keys and any possibly required clustering keys, can run only if they can query secondary indices. To be allowed to run a query including more than two secondary indices, Cassandra requires that "*allow filtering"* is used in the query to show that you really want to do it. All Cassandra queries that require this run extremely slow and Cassandra's recommendation is to avoid running them. Tables can be stored sorted by clustering keys. This is the only case in which you are allowed to run range queries and *order by* clauses.
- *R3*. Unlike the projection in a CQL SELECT, there is no guarantee that the results will contain all of the columns specified because Cassandra is schema-optional. An error does not occur if you request non-existent columns.
- *R4*. Nested queries are not allowed, there is no "OR" operator and queries that select all rows of a table are extremely slow.

CQL statements change data, look up data, store data or change the way data is stored. A select CQL expression selects one or more records from Cassandra column family and returns a result-set of rows. Similarly to SQL each row consists of a row key and a collection of columns corresponding to the query.

3. QUERY OPTIMIZER & EXECUTION

Our query optimization and execution module can be placed on top of any Cassandra cluster and is composed of the following components, shown in Fig. 1:

a) *Rewriter*: The rewriter accepts the CQL query containing joins and creates the queries for accessing each individual column-family/tables. For example assuming that *Q0* is issued by the user, this module produces as output *Q1* and *Q2* as shown in Fig. 1.

b) *Planner*: This component plans the execution of the individual queries as constructed by the rewriter. First it identifies the available indices on the queried column-families and tries to comply with *R2*. For example, if the queries don't restrict all partition keys they can only run if there are available secondary indices on these keys. To satisfy this restriction the planner automatically generates secondary indices on the required fields. In our running example, a secondary index will be automatically generated to the *producedBy.movieID* column.

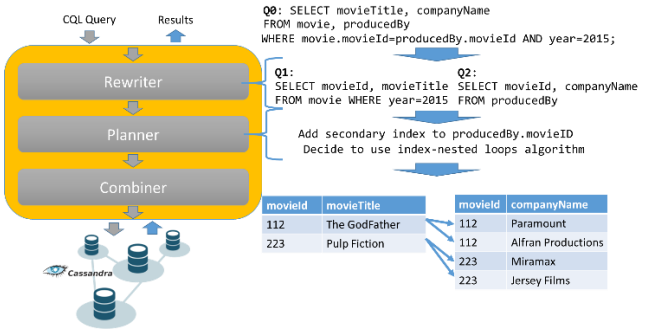

Figure 1. Components of the optimization & execution module

Besides trying to comply with all Cassandra restrictions the planner identifies which join algorithms should be used for executing the various joins by comparing the cost of left-deep trees. Currently two join algorithms have been implemented: a) a variation of Index-Nested Loops taking advantage of the existing indexes and additionally allowing joins over collection sets – indexed collections of elements (maps, sets and lists) supported after the Cassandra version 2.1; b) the sort-merge join allowing the join to be implemented in one pass over the data when the joined relations are indexed. When joining two column-families, if only one of them has an index on the joined field, the optimizer reads all rows from the non-indexed one and then uses the index for searching the indexed column-family. On the other hand when both column-families are sorted on the join column, the Sort-Merge join algorithm would be faster and is preferred by the optimizer.

c) Combiner: This component executes the queries, calculates the join using the selected algorithm and returns the results to the users.

4. EVALUATION & CONCLUSION

All algorithms reported in this paper were implemented as a Java API named CassandraJoins. The API is going to be released soon under an open-source license. To perform a preliminary evaluation of our implementation we used a single Cassandra DataStax Community Server 2.1.5 x64 node running on a system with an I5 Intel Processor, 8GB of RAM on a Samsung SSD 850 EVO. We compared our approach with the Simba-DataStax ODBC 0.7 driver and with a MySQL Server CE 5.6.24. The execution time reported in each case is the average of 50 runs of each query execution.

The first series of experiments we performed tries to join two tables with a join on the indexed field. When we have indices on the joined field the CassandraJoins optimizer is using the Index-Nested Loops join algorithm whereas, when the input relations are sorted, the optimizer uses sort-merge join. We cannot identify the specific algorithm used by Simba-DataStax - the source code is not publicly available. The results are shown in Fig. 2 for different input and output sizes. We can observe that CassandraJoins is by far more efficient than the Simba-DataStax implementation in all cases. For example, when joining column-families with $2*10^5$ rows each and the result is of the same size our approach needs 166 secs whereas Simba-DataStax ODBC driver needs 1087 secs. Obviously, when the selectivity of the query is increased the execution time is decreased. This is reasonable since Cassandra is known to be extremely slow when a query needs to retrieve all rows of a table, whereas it is extremely fast when only a small subset of the rows is selected. In addition, in all cases the implementation of Index-Nested Loops in a relational database (MySQL) is more efficient as shown in the third column of the graphs, whereas when the selectivity of the queries is high, our results are similar. However, we have to note that Cassandra scales linearly in a multi-node environment and we expect that our implementation will have even better results than MySQL when more nodes are used. Finally, to demonstrate the advantages of our implementation compared to a MySQL Database, we performed another experiment trying to join two column-families using collection indices. Since MySQL does not support collection indices the dataset has to be modelled using an additional indexed table. On the other hand Simba-DataStax does not support joins on collections. The results depicted in the last graph show that using CassandraJoins we need 0,01 sec whereas using MySQL we need 0,64 sec.

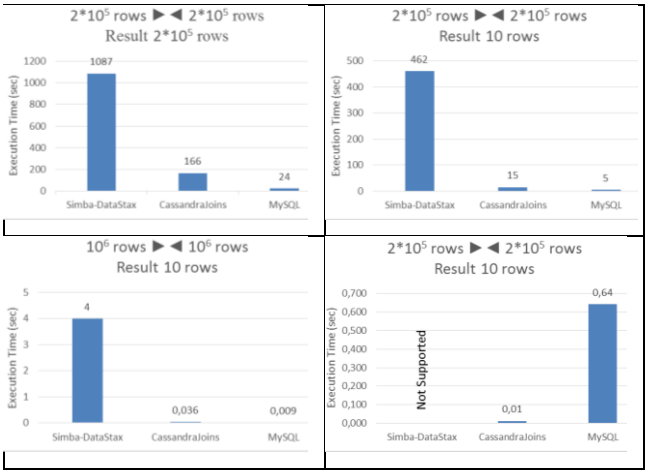

Figure 2. Results of preliminary evaluation on a single node

To the best of our knowledge our implementation is the only available non-commercial solution implementing joins over Cassandra databases. Our experiments demonstrate the advantages of our solution and confirm that our algorithms run efficiently and effectively. In all cases, we achieved better execution times than the commercial Simba-DataStax Driver currently available and our results are comparable to the execution times achieved in the relational database world. We have to note that our experiments were performed in an environment that favors relational databases (single node cluster). Surprisingly, our implementation is more efficient than relational databases when collection indices are used. The next step is to evaluate our implementation in a multi-node cluster with more data, to integrate our algorithms directly in the CQL language and to implement additional join methods.

5. REFERENCES

[1] Ntarmos, N., Patlakas, I. Triantafillou, P. 2014, Rank Join Queries in NoSQL Databases, PVLDB, 7(7), 493-504.

[2] Kim, C., Shim, K. 2015. Supporting set-valued joins in NoSQL using MapReduce, IS Journal, 49, 52-64.