Gumbo: Guarded Fragment Queries over Big Data

[Demo paper]

Jonny Daenen
Hasselt University
Diepenbeek, Belgium
jonny.daenen@uhasselt.be

Frank Neven
Hasselt University
Diepenbeek, Belgium
frank.neven@uhasselt.be

Tony Tan
Hasselt University
Diepenbeek, Belgium
tony.tan@uhasselt.be

ABSTRACT

We present Gumbo, a system for the efficient evaluation of guarded fragment queries on top of Hadoop and Spark. A key asset of Gumbo is the reduced number of jobs in comparison with recent systems such as Pig, Hive or Shark. For unnested guarded fragment queries, Gumbo even provides a constant bound on the number of jobs independent of the size of the query. In the demo, we will address the following features of Gumbo: ease-of-use, query plan construction and visualisation, and query execution.

Categories and Subject Descriptors

[**Information systems**]: MapReduce-based systems, relational parallel and distributed DBMSs, key-value stores, relational database model

General Terms

DBMS

Keywords

MapReduce, Hadoop, Spark, Guarded-fragment Queries

1. INTRODUCTION

Recent years have seen a massive growth in parallel and distributed computations based on the use of the key-value paradigm. This proliferation was fostered by the emergence of popular systems such as Hadoop [14] and Spark [1]. Recent systems such as Hive [13], Pig [9], Shark [15], etc. provide an SQL-like query language on top of Hadoop and Spark. In this demo we showcase a novel system, called Gumbo,[1] that also operates on top of Hadoop and Spark, and is specifically tailored for the evaluation of so called *guarded fragment* (GF) queries. We show that, in general, Gumbo takes less jobs to evaluate GF queries than Pig, Hive or Shark.

[1]In case you are wondering about the name, Gumbo's brother is an elephant featuring in several animated movies but not known to be interested in guarded fragment queries.

2. GUARDED-FRAGMENT QUERIES (GF)

We briefly review the definition of guarded fragment (GF) queries. They are defined inductively as follows:

- Every atomic formula $S(\bar{x})$ is a GF query.
- If $R(\bar{x},\bar{y})$ is an atomic formula and $\psi_1(\bar{y},\bar{z}),\ldots,\psi_l(\bar{y},\bar{z})$ are GF queries, then the following $\varphi(\bar{x})$ is also a GF query:

$$\varphi(\bar{x}) \;:=\; \exists\bar{y}\;\exists\bar{z}_1\;\cdots\;\exists\bar{z}_l \; R(\bar{x},\bar{y}) \wedge \left(\begin{array}{l}\text{Boolean combination of}\\ \psi_1(\bar{y},\bar{z}_1),\ldots,\psi_l(\bar{y},\bar{z}_l)\end{array}\right)$$

The original definition of GF queries in [2] allows for 'ungarded negation' by including that if $\varphi(\bar{x})$ is a GF query, then so is $\neg\varphi(\bar{x})$. For pragmatic reason, the Gumbo system does not consider such ungarded negation. Take, for example, the negation of atomic relation $\neg R(\bar{x})$. Under the closed world assumption, its evaluation will involve collecting the active domain and performing a Cartesian product on them $|\bar{x}|-1$ times, which, in general, is a very expensive operation in distributed databases. We stress that Gumbo allows for guarded negation as is allowed in the definition above and is exemplified in the example in Section 5.

Originally GF queries were introduced by Andréka, van Benthem and Németi [2] to investigate various properties of modal logic. Since then, they have become popular and found numerous applications in various fields. One example is the description logic $\mathcal{ALC}$, the basis of the knowledge representation in artificial intelligence as well as ontologies and web semantics. We refer the reader to [3, 11] and the references therein for more details. In fact, $\mathcal{ALC}$ itself is a subclass of GF queries [10]. Recent studies such as [6, 7] investigate the complexity of query answering in description logic. GF queries and its natural extension, *guarded negation* queries have also found applications in various database settings (for example, [4, 5, 12]).

To end this section, we sketch a scenario in which GF queries can be used. Consider a library that records which members borrows which books over a period of time. Specifically, there is a table R containing records with the following fields: `d`, `mem_id`, `b1`, `b2`, `b3`, `b4`, `b5`. Here `d` stands for date, `mem_id` for the member id, and each `b` represents a borrowed book.[2] Every night the new data that arose during the day

[2]In our imaginary library, each member can only borrow up

is added. To provide better service to its members, the librarian decided to find out more about the dynamics of the book transactions and starts exploring the data. For example, she may want to find out how (un)popular books about technology compared to other books. So she compiles a list S of the ISBNs of the books about technology, and uses the following query:

$$R(\mathtt{d}, \mathtt{mem\_id}, \mathtt{b1}, \ldots, \mathtt{b5}) \\ \wedge \neg S(\mathtt{b1}) \wedge \neg S(\mathtt{b2}) \wedge \neg S(\mathtt{b3}) \wedge \neg S(\mathtt{b4}) \wedge \neg S(\mathtt{b5})$$

This query computes for each member all days that only non-technology books are borrowed..

3. GF QUERY EVALUATION IN GUMBO

We contrast GF query evaluation in GUMBO with that in Pig and Hive by means of an example.

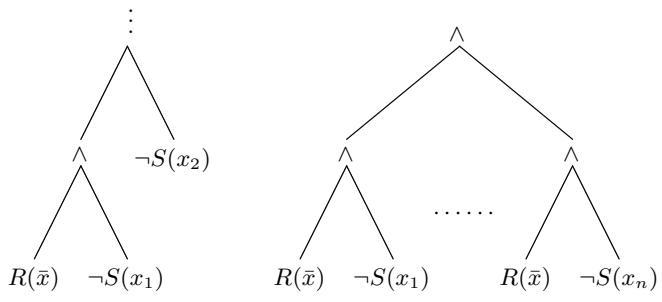
GF in Pig and Hive.

To implement a relational operation between two datasets, Pig requires us to perform a `cogroup`. Consider for instance the following query:

$$\varphi(\bar{x}) \;:=\; R(\bar{x}) \wedge \Big(\neg S(x_1) \wedge \cdots \wedge \neg S(x_n)\Big),$$

where $\bar{x} = (x_1, \ldots, x_n)$. Here, R is an n-ary relation and S is a unary database relation.

Implementing the query φ in Pig/Hive/Shark will yield either one of the following execution plans:



The plan on the left requires n **rounds of shuffling** the datasets. Here, the first round evaluates $R(\bar{x}) \wedge \neg S(x_1)$, and the result is passed to the second round which in turn evaluates $(R(\bar{x}) \wedge \neg S(x_1)) \wedge \neg S(x_2)$, and so on. In contrast, the plan on the right requires **reading the input n number of times**: once for the evaluation of each $R(\bar{x}) \wedge \neg S(x_i)$.

In either plan, we are required to either read the same input multiple times or shuffling the datasets multiple times. This creates a bottleneck, since shuffling and reading the input can be very expensive, especially when the datasets are huge.

GF in Gumbo.

GUMBO operates on top of Hadoop as well as Spark and is specifically tailored to efficiently evaluate queries of the form:

$$\varphi(\bar{x}) \;:=\; \exists \bar{y}\ \exists \bar{z}_1\ \cdots\ \exists \bar{z}_l \\ R(\bar{x}, \bar{y}) \wedge \left(\begin{array}{l} \text{Boolean combination of} \\ S_1(\bar{y}, \bar{z}_1), \ldots, S_l(\bar{y}, \bar{z}_l) \end{array} \right) \quad (1)$$

where $S_1(\bar{y}, \bar{z}_1), \ldots, S_l(\bar{y}, \bar{z}_l)$ are atomic formulas. GUMBO can evaluate such queries in **two MapReduce jobs independent of** l and the form of the Boolean combination of $S_1(\bar{y}, \bar{z}_1), \ldots, S_l(\bar{y}, \bar{z}_l)$. In particular, GUMBO reads the input datasets only once. This should be contrasted with Pig and Hive where the number of jobs grows in the size of the number of Boolean combinations.

We call queries of the form (1) basic queries, i.e. when all $S_i(\bar{y}, \bar{z}_i)$ are atomic formulas. GUMBO can also evaluate multiple queries that depend on one another. For example, we can define $\varphi_1(x) = \exists y\ E(x, y) \wedge F(y)$, where $E(x, y)$ and $F(y)$ are atomic formulas, and $\varphi_2(z) = \exists y E(z, x) \wedge \varphi_1(x)$. In this case, φ_1 is a basic query, whereas φ_2 is a nested query, since it depends on the outcome of the query φ_1. In this case, GUMBO first takes two rounds to evaluate $\varphi_1(x)$, and then another two rounds to evaluate $\varphi_2(x)$. In general, to evaluate a set of queries in which the depth of dependency is m, GUMBO requires $2m$ dependent map-reduce jobs.

4. COMPONENTS OF GUMBO

GUMBO is written in Java and consists of the six components shown in Figure 1. We will briefly describe each of these components in the following paragraphs.

Parser. GF queries are provided together with the location of input and output relations. The queries are broken up into basic GF queries, i.e. queries of the form (1) and dependencies among them are determined. Structural errors such as cyclic dependencies are also checked here. The result of this component is a DAG, where the nodes are sets of basic GF queries to be evaluated, and the edges indicate the dependencies among the queries.

Partitioner. Given a DAG of basic GF queries as input, the partitioner aims to group queries in an optimal fashion in an effort to reduce the total number of parallel rounds. To this end, each query is assigned a *round number*, and all queries that have the same round number are grouped together. The result of this phase is a list of consecutive rounds each containing a set of basic GF queries.

The partitioner can greatly reduce the number of jobs as well as "balance" the computation load among the rounds. GUMBO approximates the computational load of a query by calculating or estimating the size of its input relations. We can show that, in general, obtaining the most optimal schedule is NP-hard, even if we assume that the computational load to evaluate each query is uniform. In our initial version of GUMBO, we therefore use a greedy algorithm to approximate the optimal schedule. This is a reasonable approach assuming that the queries are "few," or that the dependency depth is quite shallow. In the later versions of GUMBO, we plan to use an SMT solver (e.g. Microsoft's Z3 system [8]) to obtain the optimal schedule.

Job Constructor. Given a list of rounds, each round is compiled into two high-level map-reduce job as described below. Locations for intermediate files are also determined here.

to 5 books. If one member borrows less than 5 books, only the first few fields are assigned and the rest are assigned with the null value.

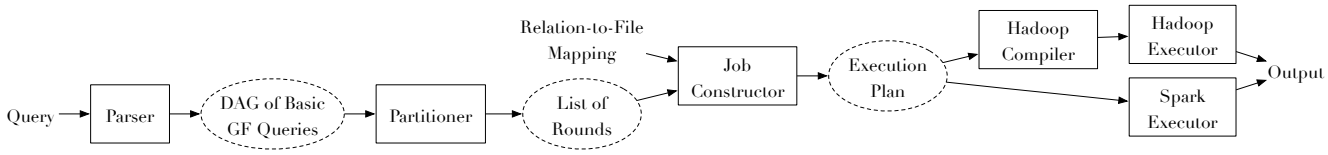


Figure 1: The GUMBO workflow.

The result of this phase is a GUMBO-plan.

For a query $\psi(\bar{x})$ of the form (1) GUMBO's job constructor builds the following mappers and reducers:

Mapper 1: On each tuple $R(\bar{a},\bar{b})$ Mapper 1 emits following the key-value pairs:

$$\big\langle S_1(\bar{c}_1) : R(\bar{a},\bar{b})\big\rangle, \ldots \big\langle S_l(\bar{c}_l) : R(\bar{a},\bar{b})\big\rangle$$

where each $\bar{c}_i$ is the projection of $(\bar{a},\bar{b})$ according to the $\bar{y}$ coordinates. The intuitive meaning of this part is that a tuple $R(\bar{a},\bar{b})$ *inquires* whether the tuple $S_i(\bar{c}_i,\bar{d})$ exists for some $\bar{d}$.

On each tuple $S_i(\bar{c},\bar{d})$ Mapper 1 emits the key-value pair $\big\langle S_i(\bar{c}) : \#\big\rangle$, where $\bar{c}$ is the projection of $(\bar{c},\bar{d})$ to the $\bar{y}$ coordinate The intuitive meaning of this part is that the key $S_i(\bar{c})$ *states* that the tuple $S_i(\bar{c},\bar{d})$ exists for some $\bar{d}$.

Reducer 1: For a key $S_i(\bar{c})$, the reducer operates on its set of values V as follows.

If # appears as value, for each value of the form $R(\bar{a},\bar{b})$, it emits a key-value pair $\big\langle R(\bar{a},\bar{b}) : i\big\rangle$, which means that $S_i(\bar{c},\bar{d})$ exists for some $\bar{d}$.

Mapper 2: This is an identity mapper that just reads and emits the key-value pairs created by Reducer 1.

Reducer 2: The keys for this reducer are all of the form $R(\bar{a},\bar{b})$, with associated values subsets of $\{1,\ldots,l\}$. The values determine the Boolean assignment ξ, where $\xi(S_j(\bar{z}_j))$ is true, if and only if j appears among the values associated with $R(\bar{a})$. So, on key $R(\bar{a},\bar{b})$, the reducer evaluates the formula according to the assignment ξ. If it evaluates to true, it writes the tuple $\bar{a}$ into the output file.

In our implementation of GUMBO, we further optimize the algorithm above, such as by compressing the keys and values, thus, decreasing the number of data bytes to be shuffled.

Hadoop Compiler & Executor. This component takes a GUMBO-plan and compiles into a set of Hadoop map-reduce jobs using the mappers and reducers constructed above. The resulting plan can then be directly executed using Hadoop.

Spark Executor. This component takes a GUMBO-plan and executes the rounds one by one. The input data is stored in Spark's RDD data structure and the execution plan is translated into RDD's transformations and actions such as `flatMap()` and `groupByKey()` to execute the jobs described in the GUMBO-plan.

Assuming that the relations $R, S_1, \ldots, S_l$ are all dumped in a single RDD A, a straightforward translation of the algorithm above to one that uses Spark's RDD is as follows.

```
B = A.flatMapToPair(<Mapper 1>);
C = B.groupByKey();
D = C.flatMap(<Reducer 1>);
E = D.flatMapToPair(<Mapper 2>);
F = E.groupByKey();
G = F.flatMap(<Reducer 2>); // G is the output
```

To make the Spark's algorithm above more efficient, we incorporate a few optimization strategies. For example, Mapper 2 in the algorithm above basically does nothing, so it can be omitted.

5. DEMO OVERVIEW

The goal of the demo is to show how GUMBO can be used to evaluate GF queries on top of Hadoop/Spark and to give insight in how it compares to the existing systems Pig, Hive and Hadoop. This comparison can be done on several levels: the query design, the query plan (which gives insight in the workings of the system) and the query execution where performance really matters. In the demo, the users can do the following.

Input the queries and the dataset. Queries are written in standard logic notation, where `&`, `|`, `!` denote the `and`, `or` and `negation` operations, respectively. We use the standard symbol `=` to define the query. For example, the user can input the following queries, where $E(x,y)$, $F(y)$ and $G(x,z)$ are atomic formulas:

```
Out1(x)   = E(x,y) & !F(y) & G(x,z);
Out2(x)   = E(x,y) & Out1(y);
Out3(x)   = E(x,y) & Out1(y) & !Out2(x);
Out4(x,y) = E(x,y) & !Out1(x);

E(x,y)    - E.txt;
F(y)      - F.txt;
G(x,z)    - G.txt;

Out1(x)   - Out1.txt;
Out2(x)   - Out2.txt;
Out3(x)   - Out3.txt;
Out4(x,y) - Out4.txt;
```

The query `Out1(x)` collects all the `x`'s where for some `y,z`, the tuple `(x,y)` is in `E` and `(x,z)` is in `G`, but `y` is not in `F`. Similarly, the query `Out2(x)` collects all the `x`'s where for some `y`, the tuple `(x,y)` is in `E` and `y` is in `Out1`.

Users also indicate which relations should be read from disk and where these reside in the file system. In our example above, $E(x,y)$ is an atomic formula, so `E(x,y) - E.txt` indicates that the relation E is to be read from the file `E.txt`

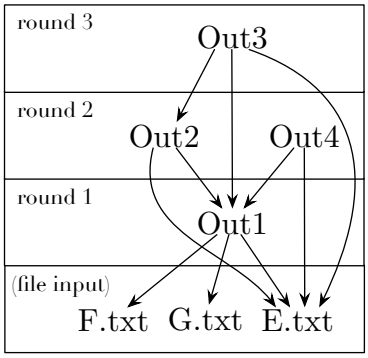


Figure 2: A DAG of jobs.

To indicate where to write the output, we use the same format. For example, `Out1(x) - Out1.txt` indicates that the output of `Out1(x)` will be written in the file `Out1.txt`.

For the users to experiment, we provide a set of queries and for each query a collection of datasets on which the query behaves differently. For example, we provide some datasets in which the query outputs a lot of tuples, as well as some datasets in which the query outputs very few tuples.

Visualise the query plan. Similar to Pig, GUMBO also provides a "visualisation" of the dependencies among the jobs to evaluate the input queries. In our demo we are going to compare the map-reduce jobs constructed by GUMBO with those constructed by Pig.

In our example of `Out1`, `Out2`, `Out3` and `Out4` above, in GUMBO we obtain the DAG shown in Figure 2. GUMBO has two choices: evaluating the query `Out4` together with `Out2` or with `Out3`. If the computation load of `Out2` is much smaller than `Out3`, then the partitioner in GUMBO will merge `Out2` with `Out4`. In this case, the partitioner assigned the same round number to `Out2` with `Out4`, which means that they are to be evaluated simultaneously in the same map-reduce job.

Such visualisation provides the users the following benefits: (i) an insight in the plan construction; (ii) viewing the round numbers assigned by the partitioner; (iii) the effect of enabling/disabling certain optimizations (e.g. partitioners); (iv) comparison in the number of jobs in the query plans of GUMBO and those written in different systems such as Pig.

Execute the queries. In the final part of our demo, we will allow the users to execute some queries on sample data. We supply some sample-data of limited size, as "real" big data would require too much execution time.

The progress of a query and the log messages produced by the system can be viewed during execution. After the execution the user is able to inspect the output files to ensure that the queries were executed correctly and also the intermediate files to clarify the inner workings of the systems.

To further illustrate the inner working of GUMBO's Hadoop executor, we will show the content of the intermediate data passed from one round to the next, as well as some metrics such as the number of mappers and reducers used by GUMBO as well as by Pig and Hive. For the case of GUMBO's Spark executor, we will show the content of the RDD in the intermediate steps leading to the evaluation of the queries.

The key points that we want to highlight in this final part of the demo are: the time GUMBO takes to evaluate a query, and the performance gain obtained by combining multiple queries.

Acknowledgement

The third author is supported by FWO Pegasus Marie Curie fellowship.

6. REFERENCES

[1] Spark. http://spark.apache.org.

[2] H. Andréka, J. van Benthem, and I. Németi. Modal languages and bounded fragments of predicate logic. *Journal of Philosophical Logic*, 1998.

[3] F. Baader, D. Calvanese, D. McGuiness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.

[4] V. Bárány, G. Gottlob, and M. Otto. Querying the guarded fragment. In *LICS*, 2010.

[5] V. Bárány, B. ten Cate, and M. Otto. Queries with guarded negation. *PVLDB*, 5(11):1328–1339, 2012.

[6] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Data complexity of query answering in description logics. In *KR*, 2006.

[7] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. *Journal of Automated Reasoning*, 39(3):385–429, 2007.

[8] L. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[9] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a highlevel dataflow system on top of mapreduce: The pig experience. *PVLDB*, 2(2):1414–1425, 2009.

[10] E. Grädel. Description logics and guarded fragments of first order logic. In *DL*, 1998.

[11] I. Horrocks. Ontologies and the semantic web. *Commun. ACM*, 51(12):58–67, 2008.

[12] R. Rosati. On the decidability and finite controllability of query processing in databases with incomplete information. In *PODS*, 2006.

[13] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, 2010.

[14] T. White. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (3. ed., revised and updated)*. O'Reilly, 2012.

[15] R. Xin, J. Rosen, M. Zaharia, M. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *SIGMOD*, 2013.