Parallel Evaluation of Multi-Semi-Joins

Jonny Daenen
Hasselt University
jonny.daenen@uhasselt.be

Frank Neven
Hasselt University
frank.neven@uhasselt.be

Tony Tan
National Taiwan University
tonytan@csie.ntu.edu.tw

Stijn Vansummeren
Université Libre de Bruxelles
stijn.vansummeren@ulb.ac.be

ABSTRACT

While services such as Amazon AWS make computing power abundantly available, adding more computing nodes can incur high costs in, for instance, pay-as-you-go plans while not always significantly improving the net running time (aka wall-clock time) of queries. In this work, we provide algorithms for parallel evaluation of SGF queries in MapReduce that optimize total time, while retaining low net time. Not only can SGF queries specify all semi-join reducers, but also more expressive queries involving disjunction and negation. Since SGF queries can be seen as Boolean combinations of (potentially nested) semi-joins, we introduce a novel multi-semi-join (MSJ) MapReduce operator that enables the evaluation of a set of semi-joins in one job. We use this operator to obtain parallel query plans for SGF queries that outvalue sequential plans w.r.t. net time and provide additional optimizations aimed at minimizing total time without severely affecting net time. Even though the latter optimizations are NP-hard, we present effective greedy algorithms. Our experiments, conducted using our own implementation Gumbo on top of Hadoop, confirm the usefulness of parallel query plans, and the effectiveness and scalability of our optimizations, all with a significant improvement over Pig and Hive.

1. INTRODUCTION

The problem of evaluating joins efficiently in massively parallel systems is an active area of research (e.g., [2–5, 7, 8, 14, 19, 25, 28]). Here, efficiency can be measured in terms of different criteria, including net time, total time, amount of communication, resource requirements and the number of synchronization steps. As parallel systems aim to bring down the net time, i.e., the difference between query end and start time, it is often considered the most important criterium. The amount of computing power is no longer an issue through the readily availability of services such as Amazon AWS. However, in pay-as-you-go plans, the cost is determined by the total time, that is, the aggregate sum of time spent by *all* computing nodes. In this paper, we focus on parallel evaluation of queries that minimize total time while retaining low net time. We consider parallel query plans that exhibit low net times and exploit commonalities between queries to bring down the total time.

Semi-joins have played a fundamental role in minimizing communication costs in traditional database systems through their role in semi-join reducers [9, 10], facilitating the reduction of communication in multi-way join computations. In more recent work, Afrati et al. [2] provide an algorithm for computing n-ary joins in MapReduce-style systems in which semi-join reducers play a central role. Motivated by the general importance of semi-joins, we study the system aspects of implementing semi-joins in a MapReduce context. In particular, we introduce a multi-semi-join operator MSJ that enables the evaluation of a set of semi-joins in one Mapreduce job while reducing resource usage like total time and requirements on cluster size without sacrificing net time. We then use this operator to efficiently evaluate Strictly Guarded Fragment (SGF) queries [6, 20]. Not only can this query language specify all semi-join reducers, but also more expressive queries involving disjunction and negation.

We illustrate our approach by means of a simple example. Consider the following SGF query Q:

$$\texttt{SELECT}\ (x,y)\ \texttt{FROM}\ R(x,y)$$
$$\texttt{WHERE}\ \big(S(x,y)\ \texttt{OR}\ S(y,x)\big)\ \texttt{AND}\ T(x,z)$$

Intuitively, this query asks for all pairs (x, y) in R for which there exists some z such that (1) (x, y) or (y, x) occurs in S and (2) (x, z) occurs in T. To evaluate Q it suffices to compute the following semi-joins

$$\begin{array}{lll} X_1 & := & R(x,y) \ltimes S(x,y); \\ X_2 & := & R(x,y) \ltimes S(y,x); \\ X_3 & := & R(x,y) \ltimes T(x,z); \end{array}$$

store the results in the binary relations X_1, X_2, or X_3, and subsequently compute $\varphi := (X_1 \cup X_2) \cap X_3$. Our multi-semi-join operator $\mathsf{MSJ}(\mathcal{S})$ (defined in Section 4.2) takes a number of semi-join-equations as input and exploits commonalities between them to optimize evaluation. In our framework, a possible query plan for query Q is of the form:

$$\begin{tikzcd} & \boxed{\mathsf{EVAL}(R,\varphi)} \arrow[dl, dash] \arrow[dr, dash] & \\ \boxed{\mathsf{MSJ}(X_1,X_2)} & & \boxed{\mathsf{MSJ}(X_3)} \end{tikzcd}$$

Proceedings of the VLDB Endowment, Vol. 9, No. 10
Copyright 2016 VLDB Endowment 2150-8097/16/06.

In this plan, the calculation of X_1 and X_2 is combined in a single MapReduce job; X_3 is calculated in a separate job; and $\mathsf{EVAL}(R, \varphi)$ is a third job responsible for computing the subset of R defined by φ. We provide a cost model to determine the best query plan for SGF queries. We note that, unlike the simple query Q illustrated here, SGF queries can be nested in general. In addition, we also show how to generalize the method to the simultaneous evaluation of multiple SGF queries.

The contributions of this paper can be summarized as follows:

1. We introduce the multi-semi-join operator $\ltimes\!\cdot(\mathcal{S})$ to evaluate a set $\mathcal{S}$ of semi-joins and present a corresponding MapReduce implementation $\mathsf{MSJ}(\mathcal{S})$.
2. We present query plans for basic, that is, unnested, SGF queries and propose an improved version of the cost model presented by [27, 36] for estimating their cost. As computing the optimal plan for a given basic SGF query is NP-hard, we provide a fast greedy heuristic GREEDY-BSGF.
3. We show that the evaluation of (possibly nested) SGF queries can be reduced to the evaluation of a set of basic SGF queries in an order consistent with the dependencies induced by the former. In this way, computing an optimal plan for a given SGF query (which is NP-hard as well) can be approximated by first determining an optimal subquery ordering, followed by an optimal evaluation of these subqueries. For the former, we present a greedy algorithm called GREEDY-SGF.
4. We experimentally assess the effectiveness of GREEDY-BSGF and GREEDY-SGF and obtain that, backed by an updated cost model, these algorithms successfully manage to bring down total times of parallel evaluation, making it comparable to that of sequential query plans, while still retaining low net times. This is especially true in the presence of commonalities among the atoms of queries. Finally, our system outperforms Pig and Hive in all aspects when it comes to parallel evaluation of SGF queries and displays interesting scaling characteristics.

Outline. This paper is organized as follows. We discuss related work in Section 2. We introduce the strictly guarded fragment (SGF) queries and discuss MapReduce and the accompanying cost model in Section 3. In Section 4, we consider the evaluation of multi-semi-joins and SGF queries. In Section 5, we discuss the experimental validation. We conclude in Section 6.

2. RELATED WORK

Recall that first-order logic (FO) queries are equivalent in expressive power to the relational algebra (RA) and form the core fragment of SQL queries (cf., e.g., [1]). Guarded-fragment (GF) queries have been studied extensively by the logicians in 1990s and 2000s, and they emerged from the intensive efforts to obtain a syntactical classification of FO queries with decidable satisfiability problems. For more details, we refer the reader to the highly influential paper by Andreka, van Benthem, and Nemeti [6], as well as survey papers by Grädel and Vardi [23,35]. In traditional database terms, GF queries are equivalent in expressive power to semi-join algebra [26]. Closely related are freely acyclic GF queries, which are GF queries restricted to using only the $\wedge$ operator and guarded existential quantifiers [31]. Flum et al. [20] introduced the term strictly guarded fragment queries for queries of the form $\exists \bar{y}(\alpha \wedge \varphi)$. That is, guarded fragment queries without Boolean combinations at the outer level. We consider a slight generalization of these queries as explained in Remark 1.

In general, obtaining the optimal plan in SQL-like query evaluation, even in centralized computation, is a hard problem [13,24]. Classic works by Yannakakis and Bernstein advocate the use of semi-join operations to optimize the evaluation of conjunctive queries [9, 10, 39]. A lot of work has been invested to optimize query evaluation in Pig [22,29], Hive [34] and SparkSQL [38] as well as in MapReduce setting in general [12]. None of them target SGF queries directly.

Tao, Lin and Xiao [33] studied *minimal* MapReduce algorithms, i.e. algorithms that scale linearly to the number of servers in all significant aspects of parallel computation such as reduce compute time, bits of information received and sent, as well as storage space required by each server. They show that, among many other problems, a single semi-join query between two relations can be evaluated by a one round minimal algorithm. This is a simpler problem, as a single basic SGF query may involve multiple semi-join queries. Afrati et al. [2] introduced a generalization of Yannakakis' algorithm (using semi-joins) to a MapReduce setting. Note that Yannakakis' algorithm starts with a sequence of semi-join operations, which is a (nested) SGF query in a very restricted form.

3. PRELIMINARIES

We start by introducing the necessary concepts and terminologies. In Section 3.1, we define the strictly guarded fragment queries, while we discuss MapReduce in Section 3.2 and the cost model in Section 3.3.

3.1 Strictly Guarded Fragment Queries

In this section, we define the strictly guarded fragment queries (SGF) [20], but use a non-standard, SQL-like notation for ease of readability.

We assume given a fixed infinite set $\mathbf{D} = \{a, b, \dots\}$ of data values and a fixed collection of relation symbols $\mathbf{S} = \{R, S, \dots\}$, disjoint with $\mathbf{D}$. Every relation symbol $R \in \mathbf{S}$ is associated with a natural number called the *arity of R*. An expression of the form $R(\bar{a})$ with R a relation symbol of arity n and $\bar{a} \in \mathbf{D}^n$ is called a *fact*. A database DB is then a finite set of facts. Hence, we write $R(\bar{a}) \in \mathsf{DB}$ to denote that a tuple $\bar{a}$ belongs to the R relation in DB.

We also assume given a fixed infinite set $\mathbf{V} = \{x, y, \dots\}$ of variables, disjoint with $\mathbf{D}$ and $\mathbf{S}$. A *term* is either a data value or a variable. An *atom* is an expression of the form $R(t_1, \dots, t_n)$ with R a relation symbol of arity n and each of the t_i a term, $i \in [1, n]$. (Note that every fact is also an atom.) A *basic strictly guarded fragment (BSGF) query* (or just a basic query for short) is an expression of the form

$$Z := \texttt{SELECT } \bar{x} \texttt{ FROM } R(\bar{t}) \; [\texttt{ WHERE } C \;]; \qquad (1)$$

where $\bar{x}$ is a sequence of variables that all occur in the atom $R(\bar{t})$, and the WHERE C clause is optional. If it occurs, C must be a Boolean combination of atoms. Furthermore, to ensure that queries belong to the guarded fragment, we require that for each pair of distinct atoms $S(\bar{u})$ and $T(\bar{v})$ in C it must hold that all variables in $\bar{u} \cap \bar{v}$ also occur in $\bar{t}$. (See also Remark 1 below.) The atom $R(\bar{t})$ is called the *guard* of the query, while the atoms occurring in C are called the *conditional atoms*. We interpret Z as the output relation of the query.

On a database DB, the BSGF query (1) defines a new relation Z containing all tuples $\bar{a}$ for which there is a substitution σ for the variables occurring in $\bar{t}$ such that $\sigma(\bar{x}) = \bar{a}$, $R(\sigma(\bar{t})) \in \mathsf{DB}$, and C evaluates to true in DB under substitution σ. Here, the evaluation of C in DB under σ is defined by recursion on the structure of C. If C is C_1 `OR` C_2, C_1 `AND` C_2, or `NOT` C_1, the semantics is the usual boolean interpretation. If C is an atom $T(\bar{v})$ then C evaluates to true if $\sigma(\bar{t}) \in R(\bar{t}) \ltimes T(\bar{v})$, i.e., if there exists a T-atom in DB that equals $R(\sigma(\bar{t}))$ on those positions where $R(\bar{t})$ and $T(\bar{v})$ share variables.

Example 1. The intersection $Z_1 := R \cap S$ and the difference $Z_2 := R - S$ between two relations R and S are expressed as follows:

$$Z_1 \;:=\; \texttt{SELECT } \bar{x} \texttt{ FROM } R(\bar{x}) \texttt{ WHERE } S(\bar{x});$$
$$Z_2 \;:=\; \texttt{SELECT } \bar{x} \texttt{ FROM } R(\bar{x}) \texttt{ WHERE NOT } S(\bar{x});$$

The semijoin $Z_3 = R(\bar{x}, \bar{y}) \ltimes S(\bar{y}, \bar{z})$ and the antijoin $Z_4 = R(\bar{x}, \bar{y}) \triangleright S(\bar{y}, \bar{z})$ are expressed as follows:

$$Z_3 \;:=\; \texttt{SELECT } \bar{x}, \bar{y} \texttt{ FROM } R(\bar{x}, \bar{y}) \texttt{ WHERE } S(\bar{y}, \bar{z});$$
$$Z_4 \;:=\; \texttt{SELECT } \bar{x}, \bar{y} \texttt{ FROM } R(\bar{x}, \bar{y}) \texttt{ WHERE NOT } S(\bar{y}, \bar{z});$$

The following BSGF query selects all the pairs (x, y) for which $(x, y, 4)$ occurs in R and either $(1, x)$ or $(y, 10)$ is in S, but not both:

$$\begin{aligned} Z_5 \;:=\; & \texttt{SELECT } (x, y) \texttt{ FROM } R(x, y, 4) \\ & \texttt{WHERE } (S(1, x) \texttt{ AND NOT } S(y, 10)) \\ & \texttt{OR } (\texttt{NOT } S(1, x) \texttt{ AND } S(y, 10)); \end{aligned}$$

Finally, the traditional star semi-join between $R(x_1, \ldots, x_n)$ and relations $S_i(x_i, y_i)$, for $i \in [1, n]$, is expressed as follows:

$$\begin{aligned} Z_6 \;:=\; & \texttt{SELECT } (x_1, \ldots, x_n) \texttt{ FROM } R(x_1, \ldots, x_n) \\ & \texttt{WHERE } S(x_1, y_1) \texttt{ AND} \ldots \texttt{ AND } S(x_n, y_n); \end{aligned}$$ □

A *strictly guarded fragment (SGF) query* is a collection of BSGFs of the form $Z_1 := \xi_1; \ldots; Z_n := \xi_n$; where each ξ_i is a BSGF that can mention any of the predicates Z_j with $j < i$. On a database DB, the SGF query then defines a new relation Z_n where every occurrence of Z_i is defined by evaluating ξ_i.

Example 2. Let **Amaz**, **BN**, and **BD** be relations containing tuples (title, author, rating) corresponding to the books found at Amazon, Barnes and Noble, and Book Depository, respectively. Let **Upcoming** contain tuples (newtitle, author) of upcoming books. The following query selects all the upcoming books (newtitle, author) of authors that have not yet received a "bad" rating for the same title at all three book retailers; Z_2 is the output relation:

$$\begin{aligned} Z_1 \;:=\; & \texttt{SELECT } \text{aut} \texttt{ FROM } \mathbf{Amaz}(\text{ttl}, \text{aut}, \text{"bad"}) \\ & \texttt{WHERE } \mathbf{BN}(\text{ttl}, \text{aut}, \text{"bad"}) \texttt{ AND } \mathbf{BD}(\text{ttl}, \text{aut}, \text{"bad"}); \\ Z_2 \;:=\; & \texttt{SELECT } (\text{new}, \text{aut}) \texttt{ FROM } \mathbf{Upcoming}(\text{new}, \text{aut}) \\ & \texttt{WHERE NOT } Z_1(\text{aut}); \end{aligned}$$

Note that this query cannot be written as a basic SGF query, since the atoms in the query computing Z_1 must share the ttl variable, which is not present in the guard of the query computing Z_2. □

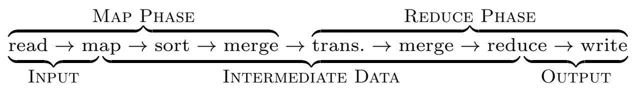


Figure 1: A depiction of the inner workings of Hadoop MR.

Remark 1. The syntax we use here differs from the traditional syntax of the Guarded Fragment [20], and is actually closer in spirit to join trees for acyclic conjunctive queries [9, 11], although we do allow disjunction and negation in the where clause. In the traditional syntax, a projection in the guarded fragment is only allowed in the form $\exists \bar{w} R(\bar{x}) \wedge \varphi(\bar{z})$ where all variables in $\bar{z}$ must occur in $\bar{x}$. One can obtain a query in the traditional syntax of the guarded fragment from our syntax by adding extra projections for the atoms in C. For example,

$$\texttt{SELECT } x \texttt{ FROM } R(x, y) \texttt{ WHERE } S(x, z_1) \texttt{ AND NOT } S(y, z_2)$$

becomes $\exists y (R(x, y) \wedge (\exists z_1) S(x, z_1) \wedge \neg (\exists z_2) S(y, z_2))$. We note that this transformation increases the nesting depth of the query. □

3.2 MapReduce

We briefly recall the Map/Reduce model of computation (MR for short), and its execution in the open-source Hadoop framework [18, 37]. An *MR job* is a pair (μ, ρ) of functions, where μ is called the map and ρ the reduce function. The execution of an MR job on an input dataset I proceeds in two stages. In the first stage, called the *map stage*, each fact $f \in I$ is processed by μ, generating a collection $\mu(f)$ of key-value pairs of the form $\langle k : v \rangle$. The total collection $\bigcup_{f \in I} \mu(f)$ of key-value pairs generated during the map phase is then grouped on the key, resulting in a number of groups, say $\langle k_1 : V_1 \rangle, \ldots, \langle k_n : V_n \rangle$ where each V_i is a set of values. Each group $\langle k_i : V_i \rangle$ is then processed by the reduce function ρ resulting again in a collection of key-value pairs per group. The total collection $\bigcup_i \rho(\langle k_i : V_i \rangle)$ is the output of the MR job.

An *MR program* is a directed acyclic graph of MR jobs, where an edge from job $(\mu, \rho) \to (\mu', \rho')$ indicates that (μ', ρ') operates on the output of (μ, ρ). We refer to the length of the longest path in an MR program as the *number of rounds* of the program.

3.3 Cost Model for MapReduce

As our aim is to reduce the total cost of parallel query plans, we need a cost model that estimates this metric for a given MR job. We briefly touch upon a cost model for analyzing the I/O complexity of an MR job based on the one introduced in [27, 36] but with a distinctive difference. The adaptation we introduce, and that is elaborated upon below, takes into account that the map function may have a different input/output ratio for different parts of the input data.

While conceptually an MR job consists of only the map and reduce stage, its inner workings are more intricate. Figure 1 summarizes the steps in the execution of an MR job. See [37, Figure 7-4, Chapter 7] for more details. The map phase involves (*i*) applying the map function on the input; (*ii*) sorting and merging the *local* key-value pairs produced by the map function, and (*iii*) writing the result to local disk.

Let $\mathcal{I}_1 \cup \cdots \cup \mathcal{I}_k$ denote the partition of the input tuples such that the mapper behaves *uniformly*[1] on every data item in $\mathcal{I}_i$. Let N_i be the size (in MB) of $\mathcal{I}_i$, and let M_i be the size (in MB) of the intermediate data output by the mapper on $\mathcal{I}_i$. The cost of the map phase on $\mathcal{I}_i$ is:

$$cost_{map}(N_i, M_i) \quad = \quad h_r N_i + \text{merge}_{map}(M_i) + l_w M_i,$$

where $\text{merge}_{map}(M_i)$, denoting the cost of sort and merge in the map stage, is expressed by

$$\text{merge}_{map}(M_i) \quad = \quad (l_r + l_w) M_i \log_D \left\lceil \frac{(M_i + \widehat{M_i})/m_i}{buf_{map}} \right\rceil.$$

See Table 1 for the meaning of the variables h_r, l_w, l_r, l_w, D, $\widehat{M_i}$, m_i, and buf_{map}.[2] The total cost incurred in the map phase equals the sum

$$\sum_{i=1}^{k} cost_{map}(N_i, M_i). \tag{2}$$

Note that the cost model in [27, 36] defines the total cost incurred in the map phase as

$$cost_{map}\left(\sum_{i=1}^{k} N_i, \sum_{i=1}^{k} M_i\right). \tag{3}$$

The latter is not always accurate. Indeed, consider for instance an MR job whose input consists of two relations R and S where the map function outputs many key-value pairs for each tuple in R and at most one key-value pair for each tuple in S, e.g., because of filtering. This difference in map output may lead to a non-proportional contribution of both input relations to the total cost. Hence, as shown by Equation (2), we opt to consider different inputs separately. This cannot be captured by map cost calculation of Equation (3), as it considers the *global* average map output size in the calculation of the merge cost. In Section 5, we illustrate this problem by means of an experiment that confirms the effectiveness of the proposed adjustment.

To analyze the cost in the reduce phase, let $M = \sum_{i=1}^{k} M_i$. The reduce stage involves (i) transferring the intermediate data (i.e., the output of the map function) to the correct reducer, (ii) merging the key-value pairs locally for each reducer, (iii) applying the reduce function, and (iv) writing the output to hdfs. Its cost will be

$$cost_{red}(M, K) \quad = \quad tM + \text{merge}_{red}(M) + h_w K,$$

where K is the size of the output of the reduce function (in MB). The cost of merging equals

$$\text{merge}_{red}(M) \quad = (l_r + l_w) M \log_D \left\lceil \frac{M/r}{buf_{red}} \right\rceil.$$

The total cost of an MR job equals the sum

$$cost_h + \sum_{i=1}^{k} cost_{map}(N_i, M_i) + cost_{red}(M, K),$$

where $cost_h$ is the overhead cost of starting an MR job.

[1] Uniform behaviour means that for every I_i, each input tuple in I_i is subjected to the same map function and generates the same number of key-value pairs. In general, a partition is a subset of an input relation.

[2] In Hadoop, each tuple output by the map function requires 16 bytes of metadata.

l_r	local disk read cost (per MB)
l_w	local disk write cost (per MB)
h_r	hdfs read cost (per MB)
h_w	hdfs write cost (per MB)
t	transfer cost (per MB)
$\widehat{M_i}$	map output meta-data for $\mathcal{I}_i$ (in MB)
m_i	number of mappers for $\mathcal{I}_i$
r	number of reducers
D	external sort merge factor
buf_{map}	map task buffer limit (in MB)
buf_{red}	reduce task buffer limit (in MB)

Table 1: Description of constants used in the cost model.

4. EVALUATING MULTI-SEMI-JOIN AND SGF QUERIES

In this section, we describe how SGF queries can be evaluated. We start by introducing some necessary building blocks in Sections 4.1 to 4.3, and describe the evaluation of BSGF queries and multiple BSGF queries in Section 4.4 and 4.5, respectively. These are then generalized to the full fragment of SGF queries in Section 4.6 and 4.7.

First, we introduce some additional notation. We say that a tuple $\bar{a} = (a_1, \ldots, a_n) \in \mathbf{D}^n$ of n data values *conforms* to a vector $\bar{t} = (t_1, \ldots, t_n)$ of terms, if

1. $\forall i, j \in [1, n]$, $t_i = t_j$ implies $a_i = a_j$; and,
2. $\forall i \in [1, n]$ if $t_i \in \mathbf{D}$, then $t_i = a_i$.

For instance, $(1, 2, 1, 3)$ conforms to $(x, 2, x, y)$. Likewise, a fact $T(\bar{a})$ conforms to an atom $U(\bar{t})$ if $T = U$ and $\bar{a}$ conforms to $\bar{t}$. We write $T(\bar{a}) \models U(\bar{t})$ to denote that $T(\bar{a})$ conforms to $U(\bar{t})$. If $f = R(\bar{a})$ is a fact conforming to an atom $\alpha = R(\bar{t})$ and $\bar{x}$ is a sequence of variables that occur in $\bar{t}$, then the projection $\pi_{\alpha;\bar{x}}(f)$ of f onto $\bar{x}$ is the tuple $\bar{b}$ obtained by projecting $\bar{a}$ on the coordinates in $\bar{x}$. For instance, let $f = R(1, 2, 1, 3)$ and $\alpha = R(x, y, x, z)$. Then, $R(1, 2, 1, 3) \models R(x, y, x, z)$ and hence $\pi_{\alpha;x,z}(f) = (1, 3)$.

4.1 Evaluating One Semi-Join

As a warm-up, let us explain how single semi-joins can be evaluated in MR. A single semi-join is a query of the form

$$Z := \texttt{SELECT } \bar{w} \texttt{ FROM } \alpha \texttt{ WHERE } \kappa; \tag{4}$$

where both α and κ are atoms. For notational convenience, we will denote this query simply by $\pi_{\bar{w}}(\alpha \ltimes \kappa)$.

To evaluate (4), one can use the following one round repartition join [12]. The mapper distinguishes between guard facts (i.e., facts in DB conforming to α) and conditional facts (i.e., facts in DB conforming to κ). Specifically, let $\bar{z}$ be the join key, i.e., those variables occurring in both α and κ. For each guard fact f such that $f \models \alpha$, the mapper emits the key-value pair $\langle \pi_{\alpha;\bar{z}}(f) : [\textsc{Req}\,\kappa; \textsc{Out}\,\pi_{\alpha;\bar{w}}(f)] \rangle$. Intuitively, this pair is a "message" sent by guard fact f to request whether a conditional fact $g \models \kappa$ with $\pi_{\kappa;\bar{z}}(g) = \pi_{\alpha;\bar{z}}(f)$ exists in the database, stating that if such a conditional fact exists, the tuple $\pi_{\alpha;\bar{w}}(f)$ should be output. Conversely, for each conditional fact $g \models \kappa$, the mapper emits a message of the form $\langle \pi_{\kappa;\bar{z}}(g) : [\textsc{Assert}\,\kappa] \rangle$, asserting the existence of a κ-conforming fact in the database with join key $\pi_{\kappa;\bar{z}}(g)$. On input $\langle \bar{b} : V \rangle$, the reducer outputs all tuples $\bar{a}$ to relation Z for which $[\textsc{Req}\,\kappa; \textsc{Out}\,\bar{a}] \in V$, provided that V contains at least one assert message.

Example 3. Consider the query $Z \coloneqq \pi_x(R(x,z) \ltimes S(z,y))$ and let I contain the facts $\{R(1,2), R(4,5), S(2,3)\}$. Then the mapper emits key-value pairs $\langle 2 : [\text{Req}\, S(z,y); \text{Out}\, 1]\rangle$, $\langle 5 : [\text{Req}\, S(z,y); \text{Out}\, 4]\rangle$ and, $\langle 2 : [\text{Assert}\, S(z,y)]\rangle$, which after reshuffling result in groups $\langle 5 : \{[\text{Req}\, S(z,y); \text{Out}\, 4]\}\rangle$ and, $\langle 2 : \{[\text{Req}\, S(z,y); \text{Out}\, 1], [\text{Assert}\, S(z,y)]\}\rangle$. Only the reducer processing the second group produces an output, namely the fact $Z(1)$. □

Cost Analysis. To compare the cost of separate and combined evaluation of multiple semi-joins in the next section, we first illustrate how to analyze the cost of evaluating a single semi-join using the cost model described above. Hereto, let $|\alpha|$ and $|\kappa|$ denote the total size of all facts that conform to α and κ, respectively. Five values are required for estimating the total cost: N_1, N_2, M_1, M_2 and K. We can now choose $M_1 = |\alpha|$ and $M_2 = |\kappa|$. For simplicity, we assume that key-value pairs output by the mapper have the same size as their corresponding input tuples, i.e., $N_1 = M_1$ and $N_2 = M_2$.[3] Finally, the output size K can be approximated by its upper bound N_1. Correct values for meta-data size and number of mappers can be derived from the number of input records and the system settings.

4.2 Evaluating a Collection of Semi-Joins

Since a BSGF query is essentially a Boolean combination of semi-joins, it can be computed by first evaluating all semi-joins followed by the evaluation of the Boolean combination. In the present section, we introduce a single-job MR program MSJ that evaluates a set of semi-joins in parallel. In the next section we introduce the single-job MR program EVAL to evaluate the Boolean combination.

We introduce a unary multi-semi-join operator $\ltimes\!\cdot(\mathcal{S})$ that takes as input a set of equations $\mathcal{S} = \{X_1 \coloneqq \pi_{\bar{x}_1}(\alpha_1 \ltimes \kappa_1), \ldots, X_n \coloneqq \pi_{\bar{x}_n}(\alpha_n \ltimes \kappa_n)\}$. It is required that the X_i are all pairwise distinct and that they do not occur in any of the right-hand sides. The semantics is straightforward: the operator computes every semi-join $\pi_{\bar{x}_i}(\alpha_i \ltimes \kappa_i)$ in $\mathcal{S}$ and stores the result in the corresponding output relation X_i.

We now expand the MR job described in Section 4.1 into a job that computes $\ltimes\!\cdot(\mathcal{S})$ by evaluating all semi-joins in parallel. Let $\bar{z}_i$ be the join key of semi-join $\pi_{\bar{x}_i}(\alpha_i \ltimes \kappa_i)$. Algorithm 1 shows the single MR job $\mathsf{MSJ}(\mathcal{S})$ that evaluates all n semi-joins at once. More specifically, MSJ simulates the repartition join of Section 4.1, but outputs request messages for *all* the guard facts at once (i.e., those facts conforming to one of the α_i for $i \in [1,n]$). Similarly, assert messages are generated simultaneously for all of the conditional facts (i.e., those facts conforming to one of the κ_i for $i \in [1,n]$). The reducer then reconciles the messages concerning the same κ_i. That is, on input $\langle \bar{b} : V\rangle$, the reducer outputs the tuple $\bar{a}$ to relation X_i for which $[\text{Req}\,(\kappa, i); \text{Out}\, \bar{a}] \in V$, provided that V contains at least one message of the form $[\text{Assert}\, \kappa]$. The output therefore consists of the relations $X_1, \ldots, X_n$, with each X_i containing the result of evaluating $\pi_{\bar{x}_i}(\alpha_i \ltimes \kappa_i)$.

Combining the evaluation of a collection of semi-joins into a single MSJ job avoids the overhead of starting multiple jobs, reads every input relation only once, and can reduce the amount of communication by packing similar messages together (cf. Section 5.1). At the same time, grouping all semi-joins together can potentially increase the average load of map and/or reduce tasks, which directly leads to an increased net time. These trade-offs are made more apparent in the following analysis and are taken into account in the algorithm Greedy-BSGF introduced in Section 4.4.

[3]Gumbo uses sampling to estimate M_i (cf. Section 5.1).

Algorithm 1 $\mathsf{MSJ}(X_1 \coloneqq \pi_{\bar{x}_1}(\alpha_1 \ltimes \kappa_1), \ldots, X_n \coloneqq \pi_{\bar{x}_n}(\alpha_n \ltimes \kappa_n))$

```
 1: function Map(Fact $f$)
 2:     buff = [ ]
 3:     for every $i$ such that $f \models \alpha_i$ do
 4:         buff += $\langle \pi_{\alpha_i;\bar{z}_i}(f) : [\text{Req}\,(\kappa_i, i); \text{Out}\,\pi_{\alpha_i;\bar{x}_i}(f)]\rangle$
 5:     for every $i$ such that $f \models \kappa_i$ do
 6:         buff += $\langle \pi_{\kappa_i;\bar{z}_i}(f) : [\text{Assert}\,\kappa_i]\rangle$
 7:     emit buffer

 8: function Reduce($\langle k : V\rangle$)
 9:     for all $[\text{Req}\,\kappa_i; \text{Out}\,\bar{a}]$ in $V$ do
10:         if $V$ contains $[\text{Assert}\,\kappa_i]$ then
11:             add $\bar{a}$ to $X_i$
```

Cost Analysis. Let all κ_i's be different atoms and $\alpha_1 = \cdots = \alpha_n = \alpha$. A similar analysis can be performed for other comparable scenarios. As before, we assume that the size of the key-value pair is the same as the size of the conforming fact, and all tuples conform to their corresponding atom. The cost of $\mathsf{MSJ}(\mathcal{S})$, denoted by *cost*$(\mathcal{S})$, equals

$$cost_h + cost_{map}(|\alpha|, n|\alpha|) + \sum_{i=1}^{n} cost_{map}(|\kappa_i|, |\kappa_i|) + cost_{red}\Big(n|\alpha| + \sum_{i=1}^{n}|\kappa_i|, \sum_{i=1}^{n}|X_i|\Big), \quad (5)$$

where $|X_i|$ is the size of the output relation X_i. If we evaluate each X_i in a separate MR job, the total cost is:

$$\sum_{i=1}^{n}\left(\begin{array}{l} cost_h + cost_{map}(|\alpha|, |\alpha|) + cost_{map}(|\kappa_i|, |\kappa_i|) \\ + cost_{red}(|\alpha| + |\kappa_i|, |X_i|) \end{array}\right) \quad (6)$$

So, single-job evaluation of all X_i's is more efficient than separate evaluation iff Equation (5) is less than Equation (6).

4.3 Evaluating Boolean Combinations

Let $X_0, X_1, \ldots X_n$ be relations with the same arity and let φ be a Boolean formula over $X_1, \ldots X_n$. It is straightforward to evaluate $X_0 \wedge \varphi$ in a single MR job: on each fact $X_i(\bar{a})$, the mapper emits $\langle \bar{a} : i\rangle$. The reducer hence receives pairs $\langle \bar{a} : V\rangle$ with V containing all the indices i for which $\bar{a} \in X_i$, and outputs $\bar{a}$ only if the Boolean formula, obtained from $X_0 \wedge \varphi$ by replacing every X_i with *true* if $i \in V$ and *false* otherwise, evaluates to true. For instance, if $\varphi = X_1 \wedge X_2 \wedge \neg X_3$, it will emit $\bar{a}$ only if V contains 0, 1 and 2 but not 3.

We denote this MR job as $\mathsf{EVAL}(X_0, \varphi)$. We emphasize that multiple Boolean formulas $Y_1 \wedge \varphi_1, \ldots, Y_n \wedge \varphi_n$ with distinct sets of variables can be evaluated in one MR job which we denote as $\mathsf{EVAL}(Y_1, \varphi_1, \ldots, Y_n, \varphi_n)$.

Cost Analysis. Let $|X_i|$ be the size of relation X_i. Then, when $|\varphi|$ is the size of the output, *cost*$(\mathsf{EVAL}(X_0, \varphi))$ equals

$$cost_h + \sum_{i=0}^{n} cost_{map}(|X_i|, |X_i|) + cost_{red}\big(\sum_{i=0}^{n}|X_i|, |\varphi|\big). \quad (7)$$

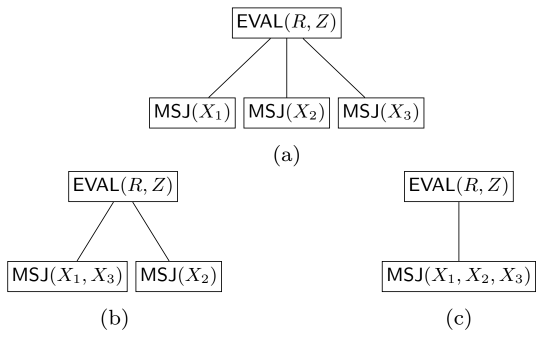


Figure 2: Different possible query plans for the query given in Example 4. Here, $X_1 := R(x,y) \ltimes S(x,z)$, $X_2 := R(x,y) \ltimes T(y)$, $X_3 := R(x,y) \ltimes U(x)$ and $Z := X_1 \wedge (X_2 \vee \neg X_3)$; trivial projections are omitted.

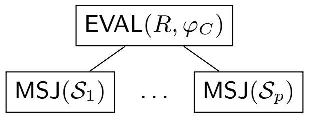
4.4 Evaluating BSGF Queries

We now have the building blocks to discuss the evaluation of basic queries. Consider the following basic query Q:

$$Z := \texttt{SELECT } \bar{w} \texttt{ FROM } R(\bar{t}) \texttt{ WHERE } C.$$

Here, C is a Boolean combination of conditional atoms κ_i, for $i \in [1, n]$, that can only share variables occurring in $\bar{t}$. Note that it is implicit that $\kappa_1, \ldots, \kappa_n$ are all different atoms. Furthermore, let $\mathcal{S}$ be the set of equations $\{X_1 := \pi_{\bar{w}}(R(\bar{t}) \ltimes \kappa_1), \ldots, X_n := \pi_{\bar{w}}(R(\bar{t}) \ltimes \kappa_n)\}$ and let φ_C be the Boolean formula obtained from C by replacing every conditional atom κ_i by X_i.

Then, for every partition $\{\mathcal{S}_1, \ldots, \mathcal{S}_p\}$ of $\mathcal{S}$, the following MR program computes Q:



We refer to any such program as a *basic MR program for Q*. Notice that all MSJ jobs can be executed in parallel. So, the above program consists in fact of two rounds, but note that there are $p+1$ MR jobs in total: one for each MSJ$(\mathcal{S}_i)$, and one for EVAL(R, φ_C).

Example 4. Figure 2 shows three alternative basic MR programs for the following query:

$$\begin{aligned} Z \;:=\; & \texttt{SELECT } x, y \texttt{ FROM } R(x,y) \\ & \texttt{WHERE } S(x,z) \texttt{ AND } (T(y) \texttt{ OR NOT } U(x)) \end{aligned} \tag{8}$$

In alternative (a), all semijoins X_1, X_2, X_3 are evaluated as separate jobs. In alternative (b), X_1 and X_3 are computed in one job, while X_2 is computed separately. In alternative (c), all semijoins X_1, X_2, X_3 are computed in a single job. □

Cost Analysis. When $\mathcal{S}$ is partitioned into $\mathcal{S}_1 \cup \cdots \cup \mathcal{S}_p$, the cost of the MR program is:

$$\textit{cost}(\mathsf{EVAL}(R, \varphi_C)) + \sum_{i=1}^{p} \textit{cost}(\mathcal{S}_i), \tag{9}$$

where the cost of $\textit{cost}(\mathcal{S}_i)$ is as in Equation (5).

Computing the Optimal Partition. By BSGF-Opt we denote the problem that takes a BSGF query Q as above and computes a partition $\mathcal{S}_1 \cup \cdots \cup \mathcal{S}_p$ of $\mathcal{S}$ such that its total cost as computed in Equation (9) is *minimal*. The Scan-Shared Optimal Grouping problem, which is known to be NP-hard, is reducible to this problem [27]:

Theorem 1. *The decision variant of* BSGF-Opt *is NP-complete.*

While for small queries the optimal solution can be found using a brute-force search, for larger queries we adopt the fast greedy heuristic introduced by Wang et al [36]. For two disjoint subsets $\mathcal{S}_i, \mathcal{S}_j \subseteq \mathcal{S}$, define:

$$\text{gain}(\mathcal{S}_i, \mathcal{S}_j) = \textit{cost}(\mathcal{S}_i) + \textit{cost}(\mathcal{S}_j) - \textit{cost}(\mathcal{S}_i \cup \mathcal{S}_j).$$

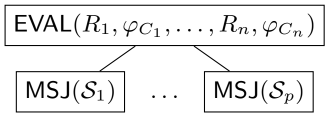
That is, gain$(\mathcal{S}_i, \mathcal{S}_j)$ denotes the cost gained by evaluating $\mathcal{S}_i \cup \mathcal{S}_j$ in one MR job rather than evaluating each of them separately. For a partition $\mathcal{S}_1 \cup \cdots \cup \mathcal{S}_p$, our heuristic algorithm greedily finds a pair $i, j \in [p] \times [p]$ such that $i \neq j$ and gain$(\mathcal{S}_i, \mathcal{S}_j) > 0$ is the greatest. If there is such a pair i, j, we merge $\mathcal{S}_i$ and $\mathcal{S}_j$ into one set. We iterate such heuristic starting with the trivial partition $\mathcal{S}_1 \cup \cdots \cup \mathcal{S}_n$, where each $\mathcal{S}_i = \{X_i := \pi_{\bar{w}}(R(\bar{t}) \ltimes \kappa_i)\}$. The algorithm stops when there is no pair i, j for which gain$(\mathcal{S}_i, \mathcal{S}_j) > 0$. We refer to this algorithm as Greedy-BSGF. For a BSGF query Q, we denote by OPT(Q) the optimal (least cost) basic MR program for Q, and by GOPT(Q) we denote the program computed by Greedy-BSGF.

4.5 Evaluating Multiple BSGF Queries

The approach presented in the previous section can be readily adapted to evaluate multiple BSGF queries. Indeed, consider a set of n BSGF queries, each of the form

$$Z_i := \texttt{SELECT } \bar{w}_i \texttt{ FROM } R_i(\bar{t}_i) \texttt{ WHERE } C_i$$

where none of the C_i can refer to any of the Z_j. A corresponding MR program is then of the form



The child nodes constitute a partition of all the necessary semi-joins. Again, φ_{C_i} is the Boolean formula obtained from C_i. We assume that the set of variables used in the Boolean formulas are disjoint. For a set of BSGF queries F, we refer to any MR program of the above form as a *basic MR program for F*, whose cost can be computed in a similar manner as above. The optimal basic program for F and the program computed by the greedy algorithm of Section 4.4 are denoted by OPT(F) and GOPT(F), respectively, and their costs are denoted by *cost*(OPT(F)) and *cost*(GOPT(F)).

4.6 Evaluating SGF Queries

Next, we turn to the evaluation of SGF queries. Recall that an SGF query Q is a sequence of basic queries of the form $Z_1 := \xi_1; \ldots; Z_n := \xi_n$; where each ξ_i can refer to the relations Z_j with $j < i$. We denote the BSGF $Z_i := \xi_i$ by Q_i. The most naive way to compute Q is to evaluate the BSGF queries in Q sequentially, where each ξ_i is evaluated using the approach detailed in the previous section. This leads to a $2n$-round MR program. We would like to have a better strategy that aims at decreasing the total time by combining the evaluation of different independent subqueries.

To this end, let $\mathcal{G}_Q$ be the dependency graph induced by Q. That is, $\mathcal{G}_Q$ consists of a set F of n nodes (one for each BSGF query) and there is an edge from Q_i to Q_j if relation Z_i is mentioned in ξ_j. A *multiway topological sort* of the dependency graph $\mathcal{G}_Q$ is a sequence $(F_1, \ldots, F_k)$ such that

1. $\{F_1, \ldots, F_k\}$ is a partition of F;
2. if there is an edge from node u to node v in $\mathcal{G}_Q$, then $u \in F_i$ and $v \in F_j$ such that $i < j$.

Notice that any multiway topological sort $(F_1, \ldots, F_k)$ of $\mathcal{G}_Q$ provides a valid ordering to evaluate Q, i.e., all the queries in F_i are evaluated before F_j whenever $i < j$.

Example 5. Let us illustrate the latter by means of an example. Consider the following SGF query Q:

$$
\begin{array}{lll}
Q_1: & Z_1 & := \texttt{SELECT } x, y \texttt{ FROM } R_1(x,y) \texttt{ WHERE } S(x) \\
Q_2: & Z_2 & := \texttt{SELECT } x, y \texttt{ FROM } Z_1(x,y) \texttt{ WHERE } T(x) \\
Q_3: & Z_3 & := \texttt{SELECT } x, y \texttt{ FROM } Z_2(x,y) \texttt{ WHERE } U(x) \\
Q_4: & Z_4 & := \texttt{SELECT } x, y \texttt{ FROM } R_2(x,y) \texttt{ WHERE } T(x) \\
Q_5: & Z_5 & := \texttt{SELECT } x, y \texttt{ FROM } Z_3(x,y) \texttt{ WHERE } Z_4(x,x)
\end{array}
$$

The dependency graph $\mathcal{G}_Q$ is as follows:

$$\begin{tikzcd}
 & Q_3 \arrow[dl] & Q_2 \arrow[l] & Q_1 \arrow[l] \\
Q_5 & & & \\
 & Q_4 \arrow[ul] & &
\end{tikzcd}$$

There are four possible multiway topological sorts of $\mathcal{G}_Q$:

1. $(\{Q_1, Q_4\}, \{Q_2\}, \{Q_3\}, \{Q_5\})$.
2. $(\{Q_1\}, \{Q_2, Q_4\}, \{Q_3\}, \{Q_5\})$.
3. $(\{Q_1\}, \{Q_2\}, \{Q_3, Q_4\}, \{Q_5\})$.
4. $(\{Q_1\}, \{Q_2\}, \{Q_3\}, \{Q_4\}, \{Q_5\})$. □

Let $\mathcal{F} = (F_1, \ldots, F_k)$ be a topological sort of $\mathcal{G}_Q$. Since the optimal program $\mathsf{OPT}(F_i)$, defined in Subsection 4.5, is intractable (due to Theorem 1), we will use the greedy approach to evaluate F_i, i.e., $\mathsf{GOPT}(F_i)$ as defined in Section 4.5. The cost of evaluating Q according to $\mathcal{F}$ is

$$cost(\mathcal{F}) \;=\; \sum_{i=1}^{k} \; cost(\mathsf{GOPT}(F_i)) \tag{10}$$

We define the optimization problem SGF-Opt that takes as input an SGF query Q and constructs a multiway topological sort $\mathcal{F}$ of $\mathcal{G}_Q$ with minimal $cost(\mathcal{F})$. By reduction from Subset Sum [21] we obtain the following result (cf. [16]):

Theorem 2. *The decision variant of* SGF-Opt *is NP-complete.*

In the following, we present a novel heuristic for computing a multiway topological sort of an SGF that tries to maximize the overlap between queries. To this end, we define the *overlap* between a BSGF query Q and a set of BSGF queries F, denoted by $\text{overlap}(Q, F)$, to be the number of relations occurring in Q that also occur in F. For instance, in Example 5, the overlap between Q_2 and $\{Q_1, Q_3, Q_4, Q_5\}$ is 1 as they share only relation T.

Consider the following algorithm Greedy-SGF that computes a multiway topological sort $\mathcal{F}$ of an SGF query Q. Initially, all the vertices in the dependency graph $\mathcal{G}_Q$ are colored blue and $\mathcal{X} = ()$. The algorithm performs the following iteration with the invariant that $\mathcal{X}$ is a multiway topological sort of the red vertices in $\mathcal{G}$:

1. Suppose $\mathcal{X} = (F_1, \ldots, F_m)$ and blue vertices remain.
2. Let D be the set of those blue vertices in $\mathcal{G}_Q$ for which none of the incoming edges are from other blue vertices. (Due to the acyclicity of $\mathcal{G}_Q$, the set D is nonempty if $\mathcal{G}_Q$ still has blue vertices.)
3. Find a pair (u, F_i) such that $u \in D$, $(F_1, \ldots, F_i \cup \{u\}, \ldots, F_m)$ is a topological sort of the vertices $\{u\} \cup \bigcup_i F_i$, and $\text{overlap}(u, F_i)$ is non-zero.
4. If such a pair (u, F_i) exists, choose one with maximal $\text{overlap}(u, F_i)$, and set $\mathcal{X} = (F_1, \ldots, F_i \cup \{u\}, \ldots, F_m)$. Otherwise, set $\mathcal{X} = (F_1, \ldots, F_m, \{u\})$.
5. Color the vertex u red.

The iteration stops when every vertex in $\mathcal{G}_Q$ is red, and hence, $\mathcal{X}$ is a multiway topological sort of $\mathcal{G}_Q$. Clearly, the number of iterations is n, where n is the number of vertices in $\mathcal{G}_Q$. Each iteration takes $O(n^2)$. Therefore, the heuristic algorithm outlined above runs in $O(n^3)$ time.

Note that a naive dynamic evaluation strategy may consist of re-running Greedy-SGF after each BSGF evaluation in order to obtain an updated MR query plan.

4.7 Evaluating Multiple SGF Queries

Evaluating a collection of SGF queries can be done in the same way as evaluating one SGF query. Indeed, we can simply consider the union of all BSGF subqueries. Note that this strategy can exploit overlap between different subqueries, potentially bringing down the total and/or net time.

5. EXPERIMENTAL VALIDATION

In this section, we experimentally validate the effectiveness of our algorithms. First, we discuss our experimental setup in Section 5.1. In Section 5.2, we discuss the evaluation of BSGF queries. In particular, we compare with Pig and Hive and address the effectiveness of the cost model. The experiments concerning nested SGF queries are presented in Section 5.3. Finally, Section 5.4 discusses the overal performance of our own system called Gumbo.

5.1 Experimental Setup

The algorithms Greedy-BSGF and Greedy-SGF are implemented in a system called Gumbo [15, 17]. Gumbo runs on top of Hadoop, and adopts several important optimizations:

(1) Message packing, as also used in [36], reduces network communication by packing all the request and assert messages associated with the same key into one list.
(2) Emitting a reference to each guard tuple (i.e., a tuple id) rather than the tuple itself when evaluating (B)SGF queries significantly reduces the number of bytes that are shuffled. To compensate for this reduction, the guard relation needs to be re-read in the EVAL job but the latter is insignificant w.r.t. the gained improvement.
(3) Setting the number of reducers in function of the intermediate data size. An estimate of the intermediate size is obtained through simulation of the map function on a sample of the input relations. The latter estimates are also used as approximate values for N_{inp}, N_{int}, and N_{out}. For the experiments below, 256MB of data was allocated to each reducer.
(4) When the conditional atoms of a BSGF query all have the same join-key, the query can be evaluated in one job by combining MSJ and EVAL. A similar reduction to one job can be obtained when the the Boolean condition is

QID	Query	Type of query
A1	$R(x,y,z,w) \ltimes S(x) \wedge T(y) \wedge U(z) \wedge V(w)$	guard sharing
A2	$R(x,y,z,w) \ltimes S(x) \wedge S(y) \wedge S(z) \wedge S(w)$	guard & conditional name sharing
A3	$R(x,y,z,w) \ltimes S(x) \wedge T(x) \wedge U(x) \wedge V(x)$	guard & conditional key sharing
A4	$R(x,y,z,w) \ltimes S(x) \wedge T(y) \wedge U(z) \wedge V(w)$ $G(x,y,z,w) \ltimes W(x) \wedge X(y) \wedge Y(z) \wedge Z(w)$	no sharing
A5	$R(x,y,z,w) \ltimes S(x) \wedge T(y) \wedge U(z) \wedge V(w)$ $G(x,y,z,w) \ltimes S(x) \wedge T(y) \wedge U(z) \wedge V(w)$	conditional name sharing
B1	$R(x,y,z,w) \ltimes S(x) \wedge T(x) \wedge U(x) \wedge V(x) \wedge S(y) \wedge T(y) \wedge U(y) \wedge V(y) \wedge S(z) \wedge T(z) \wedge U(z) \wedge V(z) \wedge S(w) \wedge T(w) \wedge U(w) \wedge V(w)$	large conjunctive query
B2	$R(x,y,z,w) \ltimes (S(x) \wedge \neg T(x) \wedge \neg U(x) \wedge \neg V(x)) \vee (\neg S(x) \wedge T(x) \wedge \neg U(x) \wedge \neg V(x)) \vee (S(x) \wedge \neg T(x) \wedge U(x) \wedge \neg V(x)) \vee (\neg S(x) \wedge \neg T(x) \wedge \neg U(x) \wedge V(x))$	uniqueness query

Table 2: Queries used in the BSGF-experiment

restricted to only disjunction and negation. The same optimization also works for multiple BSGF queries. We refer to these programs as 1-ROUND below.

All experiments are conducted on the HPC infrastructure of the Flemish Supercomputer Center (VSC). Each experiment was run on a cluster consisting of 10 compute nodes. Each node features two 10-core "Ivy Bridge" Xeon E5-2680v2 CPUs (2.8 GHz, 25 MB level 3 cache) with 64 GB of RAM and a single 250GB harddisk. The nodes are linked to a IB-QDR Infiniband network. We used Hadoop 2.6.2, Pig 0.15.0 and Hive 1.2.1; the specific Hadoop settings and cost model constants can be found in [16]. All experiments are run three times; average results are reported.

Queries typically contain a multitude of relations and the input sizes of our experiments go up to 100GB depending on the query and the evaluation strategy. The data that is used for the guard relations consists of 100M tuples that add up to 4GB per relation. For the conditional relations we use the same number of tuples that add up to 1GB per relation; 50% of the conditional tuples match those of the guard relation.

We use the following performance metrics:

1. *total time*: the aggregate sum of time spent by all mappers and reducers;
2. *net time*: elapsed time between query submission to obtaining the final result;
3. *input cost*: the number of bytes read from hdfs over the entire MR plan;
4. *communication cost*: the number of bytes that are transferred from mappers to reducers.

5.2 BSGF Queries

Table 2 lists the type of BSGF queries used in this section.[4] Figures 3 & 4 show the results that are discussed next.

[4] The results obtained here generalize to non-conjunctive BSGF queries. Conjunctive BSGF queries were chosen here to simplify the comparison with sequential query plans.

Sequential vs. Parallel. We first compare sequential and parallel evaluation of queries A1–A5 to highlight the major differences between sequential and parallel query plans and to illustrate the effect of grouping. In particular, we consider three evaluation strategies in Gumbo: (*i*) evaluating all semi-joins sequentially by applying a semi-join to the output of the previous stage (SEQ), where the number of rounds depends on the number of semi-joins; (*ii*) using the 2-round strategy with algorithm GREEDY-BSGF (GREEDY); and, (*iii*) a more naive version of GREEDY where no grouping occurs, i.e., every semi-join is evaluated separately in parallel (PAR). As semi-join algorithms in MR have not received significant attention, we choose to compare with the two extreme approaches: no parallelization (SEQ) and parallelization without grouping (PAR). Relative improvements of PAR and GREEDY w.r.t. SEQ are shown in Figure 3b.

We find that both PAR and GREEDY result in lower net times. In particular, we see average improvements of 39% and 31% over SEQ, respectively. On the other hand, the total times for PAR are much higher than for SEQ: 132% higher on average. This is explained by the increase in both input and communication bytes, whereas the data size can be reduced after each step in the sequential evaluation. For GREEDY, total times vary depending on the structure of the query. Total times are significantly reduced for queries where conditional atoms share join keys and/or relation names. This effect is most obvious for queries A1, A2 and A5 where we oberve reductions in net time of 30%, 29% and 30%, respectively, w.r.t. PAR.

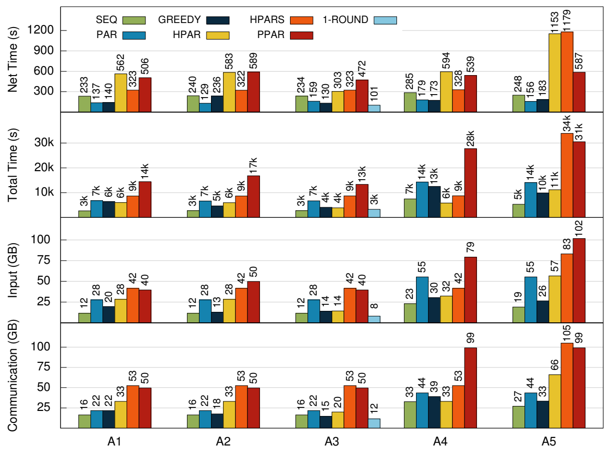
For query A3, all conditional atoms have the same join key, making 1-round (1-ROUND, see Section 5.1) evaluation possible. This further reduces the total and net time to only 49% and 63% of those of PAR, respectively.

Hive & Pig. We now examine parallel query evaluation in Pig and Hive and show that Gumbo outperforms both systems for BSGF queries. For this test, we implement the 2-round query plans of Section 4.4 directly in Pig and Hive. For Hive, we consider two evaluation strategies: one using Hive's left-outer-join operations (HPAR) and one using Hive's semi-join operations (HPARS). For Pig, we consider one strategy that is implemented using the COGROUP operation (PPAR). We also studied sequential evaluation of BSGF queries in both systems but choose to omit the results here as both performed drastically worse than their Gumbo equivalent (SEQ) in terms of net and total time.

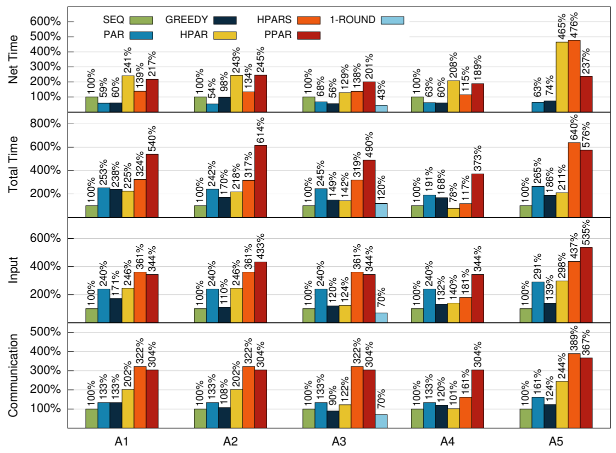
First, we find that HPAR lacks parallelization. This is caused by Hive's restriction that certain join operations are executed sequentially, even when parallel execution is enabled. This leads to net times that are 238% higher on average, compared to PAR. Note that query A3 shows a better net time than the other queries. This is caused by Hive allowing grouping on certain join queries, effectively bringing the number of jobs (and rounds) down to 2.

Next, we find that HPARS performs better than HPAR in terms of net time but is still 126% higher on average than PAR. The lower net times w.r.t. HPAR are explained by Hive allowing parallel execution of semi-join operations, without allowing any form of grouping. This effectively makes HPAR the Hive equivalent of PAR. The high net times are caused by Hive's higher average map and reduce input sizes.

Finally, Pig shows an average net time increase of 254%.



(a) Absolute values.



(b) Values relative to SEQ.

Figure 3: Results for evaluating the BSGF queries using different strategies.

This is mainly caused by the lack of reduction in intermediate data and in input bytes, together with input-based reducer allocation (1GB of map input data per reducer). For these queries, this leads to a low number of reducers, causing the average reduce time, and hence overall net time, to go up.

As the reported net times for Hive and Pig are much higher than for sequential evaluation in Gumbo (SEQ), we conclude that Pig and Hive, with default settings, are unfit for parallel evaluation of BSGF queries. For this reason we restrict our attention to Gumbo in the following sections.

Large Queries. Next, we compare the evaluation of two larger BSGF queries B1 and B2 from Table 2. The results are shown in Figure 4. Query B1 is a conjunctive BSGF query featuring a high number of atoms. Its structure ensures a deep sequential plan that results in a high net time for SEQ. We find that PAR only takes 22% of the net time, which shows that parallel query plans can yield significant improvements. Conversely, PAR takes up 261% more total time than SEQ, as the latter is more efficient in pruning the data at each step. Here, GREEDY is able to successfully parallelize query execution without sacrificing total time. Indeed, GREEDY exhibits a net time comparable to that of PAR and a total time comparable to that of SEQ.

Query B2 consists of a large boolean combination and is called the *uniqueness query*. This query returns the tuples that can be connected to precisely one of the conditional relations through a given attribute. The number of distinct conditional atoms is limited, and the disjunction at the highest level makes it possible to evaluate the four conjunctive subexpressions in parallel using SEQ. Still, we find that the net time of of PAR improves that of SEQ by 66%. As PAR only needs to calculate the result of four semi-join queries in its first round, we also find a reduction of 57% in total time. GREEDY further reduces both numbers.

Finally, for B2, a 1-round evaluation (1-ROUND, see Section 5.1) can be considered, as only one key is used for the conditional atoms. This evaluation strategy brings down both net and total time of SEQ by more than 80%.

Cost Model. As explained in Section 3.3, the major difference between our cost model and that of Wang et al. [36] (referred to as $cost_{gumbo}$ and $cost_{wang}$, respectively, from here onward) concerns identifying the individual map cost contributions of the input relations. For queries where the map input/output ratio differs greatly among the input relations, we notice a vast improvement for the GREEDY strategy. We illustrate this using the following query:

$$\begin{aligned} R(x,y,z,w) \ltimes\ & S_1(\bar{x}_1,c) \wedge \ldots \wedge S_1(\bar{x}_{12},c) \wedge \\ & S_2(\bar{x}_1,c) \wedge \ldots \wedge S_2(\bar{x}_{12},c) \wedge \\ & S_3(\bar{x}_1,c) \wedge \ldots \wedge S_3(\bar{x}_{12},c) \wedge \\ & S_4(\bar{x}_1,c) \wedge \ldots \wedge S_4(\bar{x}_{12},c), \end{aligned}$$

where $\bar{x}_1,\ldots,\bar{x}_{12}$ are all distinct keys and c is a constant that filters out all tuples from $S_1,\ldots,S_4$. The results for evaluating this query using GREEDY with $cost_{gumbo}$ and $cost_{wang}$ are unmistakable: $cost_{gumbo}$ provides a 43% reduction in total time and a 71% reduction in net time. The explanation is that $cost_{wang}$ does not discriminate between different input relations, it averages out the intermediate data and therefore fails to account for the high number of map-side merges and the accompanying increase in both total and net time.

For queries A1–A5 and B1–B2, where input relations have a contribution to map output that is proportional to their input size, we find that both cost models behave similarly. When comparing two random jobs, the cost models correctly identify the highest cost job in 72.28% ($cost_{gumbo}$) and 69.37% of the cases ($cost_{wang}$). Hence, we find that $cost_{gumbo}$ provides a more robust cost estimation as it can isolate input relations that have a non-proportional contribution to the map output, while it automatically resorts to $cost_{wang}$ in the case of an equal contribution.

Conclusion. We conclude that parallel evaluation effectively lowers net times, at the cost of higher total times. GREEDY, backed by an updated cost model, successfully manages to bring down total times of parallel evaluation, especially in the presence of commonalities among the atoms of BSGF queries. For larger queries, total times similar to

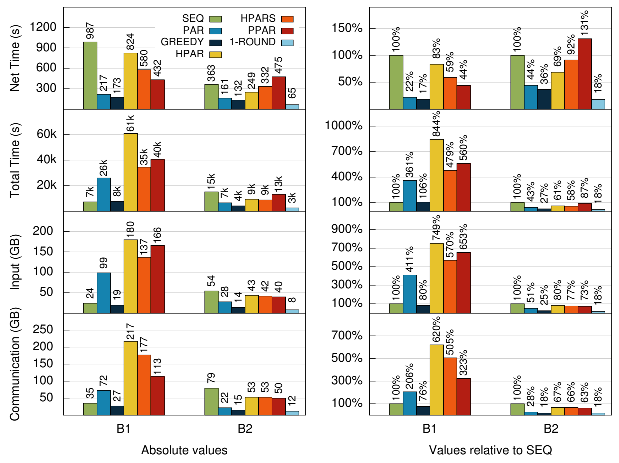


Figure 4: Results for large BSGF queries.

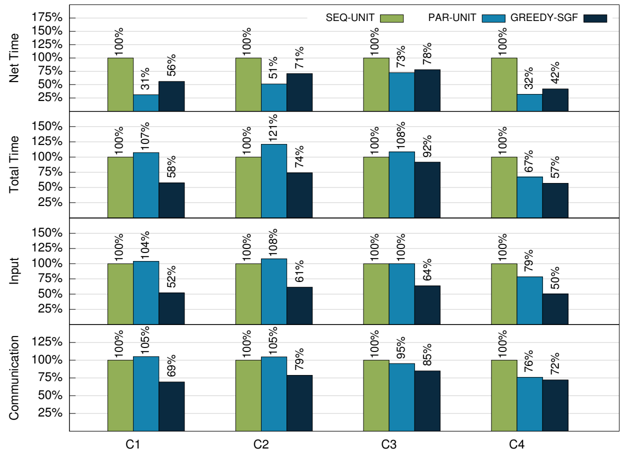


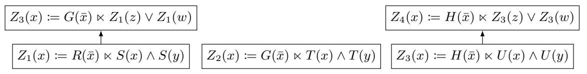
Figure 5: SGF results, values relative to SEQUNIT.

SEQ are obtained. Finally, Gumbo outperforms Pig and Hive in all aspects when it comes to parallel evaluation of BSGF queries.

5.3 SGF Queries

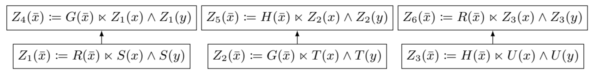
In this section, we show that the algorithm Greedy-SGF succeeds in lowering total time while avoiding significant increase in net time. Figure 6 gives an overview of the type of queries that are used. Results are depicted in Figure 5. Note that these queries all exhibit different properties. Queries C1 and C2 both contain a set of SGF queries where a number of atoms overlap. Query C3 is a complex query that contains a multitude of different atoms. Finally, Query C4 consists of two levels and many overlapping atoms.

We consider the following evaluation strategies in Gumbo: (i) sequentially, i.e., one at a time, evaluating all BSGF queries in a bottom-up fashion (SEQUNIT); (ii) evaluating all BSGF queries in a bottom-up fashion level by level where queries on the same level are executed in parallel (PARUNIT); and, (iii) using the greedily computed topological sort combined with Greedy-BSGF (Greedy-SGF);



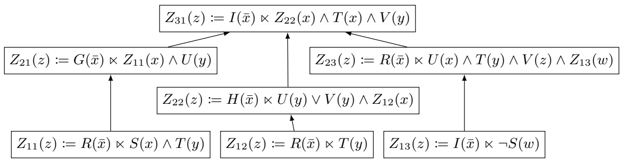
$Z_3(x) \coloneqq G(\bar{x}) \ltimes Z_1(z) \vee Z_1(w)$
$Z_4(x) \coloneqq H(\bar{x}) \ltimes Z_3(z) \vee Z_3(w)$
$Z_1(x) \coloneqq R(\bar{x}) \ltimes S(x) \wedge S(y)$
$Z_2(x) \coloneqq G(\bar{x}) \ltimes T(x) \wedge T(y)$
$Z_3(x) \coloneqq H(\bar{x}) \ltimes U(x) \wedge U(y)$

(a) Query Set C1



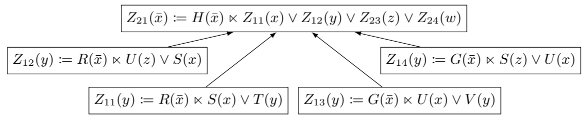
$Z_4(\bar{x}) \coloneqq G(\bar{x}) \ltimes Z_1(x) \wedge Z_1(y)$
$Z_5(\bar{x}) \coloneqq H(\bar{x}) \ltimes Z_2(x) \wedge Z_2(y)$
$Z_6(\bar{x}) \coloneqq R(\bar{x}) \ltimes Z_3(x) \wedge Z_3(y)$
$Z_1(\bar{x}) \coloneqq R(\bar{x}) \ltimes S(x) \wedge S(y)$
$Z_2(\bar{x}) \coloneqq G(\bar{x}) \ltimes T(x) \wedge T(y)$
$Z_3(\bar{x}) \coloneqq H(\bar{x}) \ltimes U(x) \wedge U(y)$

(b) Query Set C2



$Z_{31}(z) \coloneqq I(\bar{x}) \ltimes Z_{22}(x) \wedge T(x) \wedge V(y)$
$Z_{21}(z) \coloneqq G(\bar{x}) \ltimes Z_{11}(x) \wedge U(y)$
$Z_{23}(z) \coloneqq R(\bar{x}) \ltimes U(x) \wedge T(y) \wedge V(z) \wedge Z_{13}(w)$
$Z_{22}(z) \coloneqq H(\bar{x}) \ltimes U(y) \vee V(y) \wedge Z_{12}(x)$
$Z_{11}(z) \coloneqq R(\bar{x}) \ltimes S(x) \wedge T(y)$
$Z_{12}(z) \coloneqq R(\bar{x}) \ltimes T(y)$
$Z_{13}(z) \coloneqq I(\bar{x}) \ltimes \neg S(w)$

(c) Query C3



$Z_{21}(\bar{x}) \coloneqq H(\bar{x}) \ltimes Z_{11}(x) \vee Z_{12}(y) \vee Z_{23}(z) \vee Z_{24}(w)$
$Z_{12}(y) \coloneqq R(\bar{x}) \ltimes U(z) \vee S(x)$
$Z_{14}(y) \coloneqq G(\bar{x}) \ltimes S(z) \vee U(x)$
$Z_{11}(y) \coloneqq R(\bar{x}) \ltimes S(x) \vee T(y)$
$Z_{13}(y) \coloneqq G(\bar{x}) \ltimes U(x) \vee V(y)$

(d) Query C4

Figure 6: The queries used in the SGF experiment. Each node represents one BSGF subquery ($\bar{x} = x, y, z, w$).

Note that in SEQUNIT and PARUNIT all semi-joins are evaluated in separate jobs. For all tests conducted here, we found that Greedy-SGF yields multiway topological sorts that are identical to the optimal topological sort (computed trough brute-force methods); hence, we omit the results for the optimal plans.

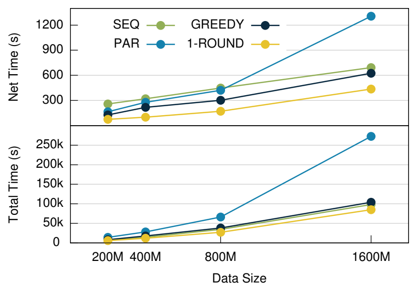
Similar to our observations for BSGF queries, we find that full sequential evaluation (SEQUNIT) results in the largest net times. Indeed, PARUNIT exhibits 55% lower net times on average. We also observe that PARUNIT exhibits significantly larger total times than SEQUNIT for queries C1 and C2, while this is not the case for C3 and C4. The reason is that for C3 and C4, queries on the same level still share common characteristics, leading to a lower number of distinct semi-joins.

For Greedy-SGF, we find that it exhibits net times that are, on average, 42% lower than SEQUNIT, while still being 29% higher than PARUNIT. The main reason for this is the fact that Greedy-SGF aims to minimize total time, and may introduce extra levels in the MR query plan to obtain this goal. Indeed, we find that total times are down 27% w.r.t. SEQUNIT, and 29% w.r.t. PARUNIT.

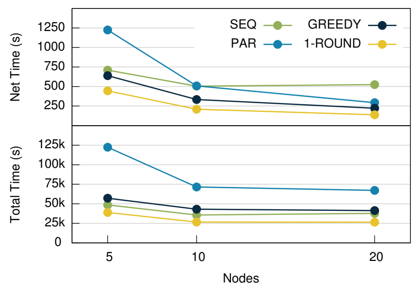
Finally, we note that the absolute savings in net time range from 115s to 737s for these queries, far outweighing the overhead cost of calculating the query plan itself, which typically takes around 10s (sampling included). Hence, we conclude that Greedy-SGF provides an evaluation strategy for SGF queries that manages to bring down the total time (and hence, the resource cost) of parallel query plans, while still exhibiting low net times when compared to sequential approaches.

5.4 System Characteristics

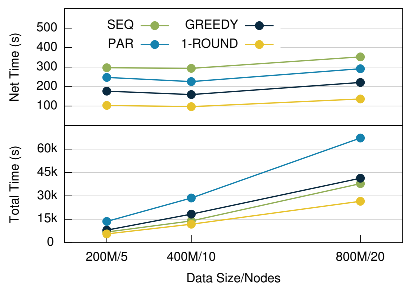
In this final experiment, we study the effect of growing data size, cluster size, query size, and selectivity. We choose



(a) Varying data size (10 nodes).



(b) Varying cluster size (800M tuples).



(c) Varying data and cluster size.

Figure 7: Results for system characteristics tests for Gumbo.

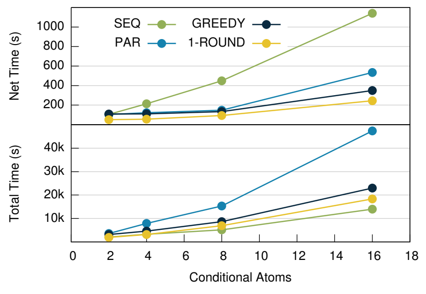


Figure 8: Varying the number of atoms.

	Net time			Total time		
	A1	A2	A3	A1	A2	A3
SEQ	10%	9%	8%	79%	95%	88%
PAR	33%	46%	69%	41%	47%	58%
GREEDY	23%	30%	13%	45%	57%	15%

Table 3: Increase in net and total time when changing selectivity from 0.1 to 0.9 for queries A1–A3.

queries similar to A3 to include the 1-ROUND strategy. Similar growth properties hold for the other query types.

Data & Cluster Size. Figures 7a-7c show the result of evaluating A3 using SEQ, PAR, GREEDY and 1-ROUND under the presence of variable data and cluster size. We summarize the most important observations:

1. in all scenarios, 1-ROUND performs best in terms of net and total time;
2. due to its lack of grouping, PAR needs a high number of mappers, which at some point exceeds the capacity of the cluster. This causes a large increase in net and total time, an effect that can be seen in Figure 7a.
3. with regard to net time, adding more nodes is very effective for the parallel strategies PAR, GREEDY and 1-ROUND; in contrast, adding more nodes does not improve SEQ substantially after some point;
4. when scaling data and cluster size at the same time, all strategies are able to maintain their net times in the presence of an increasing total time.

Query Size. We consider a set of queries similar to A3, where the number of conditional atoms ranges from 2 to 16. Results are depicted in Figure 8. With regard to net time, we find that SEQ shows an increase in net time that is strongly related to the query size, while PAR, GREEDY and 1-ROUND are less affected. For total time, we observe the converse for PAR, as this strategy cannot benefit from the packing optimization in the same way as GREEDY and 1-ROUND can.

Selectivity. For a conditional relation, we define its *selectivity rate* as the percentage of guard tuples it matches. We tested queries A1–A3 for selectivity rates 0.1 (high selectivity), 0.3, 0.5, 0.7 and 0.9 (low selectivity). The increase in net time and total time between selectivity rates 0.1 and 0.9 is summarized in Table 3. In general, we find that the selectivity has the most influence on the net times of PAR and GREEDY, and on the total times of SEQ. Finally, we observe that the filtering characteristics of SEQ disappear in the presence of low selectivity data, causing total times to become comparable to GREEDY for queries where packing is possible, such as A3. This can be explained by GREEDY being less sensitive to selectivity for queries where conditional atoms share a common join key, making an effective compression of intermediate data possible through packing.

6. DISCUSSION

We have shown that naive parallel evaluation of semi-join and (B)SGF queries can greatly reduce the net time of query execution, but, as expected, generally comes at a cost of an increased total time. We presented several methods that aim to reduce the total cost (total time) of parallel MR query plans, while at the same time avoiding a high increase in net time. We proposed a two-tiered strategy for selecting the optimal parallel MR query plan for an SGF query in terms of total cost. As the general problem was proven to be NP-hard, we devised a two-tiered greedy approach that leverages on the existing technique of grouping MR jobs together based on a cost-model. The greedy approach was shown to be effective for evaluating (B)SGF queries in practice through several experiments using our own implementation called Gumbo. For certain classes of queries, our approach makes it even possible to evaluate (B)SGF queries in parallel with a total time similar to that of sequential evaluation. We also showed that the profuse number of optimizations that are offered in Gumbo allow it to outperform Pig and Hive in several aspects.

We remark that the techniques introduced in this paper generalize to any map/reduce framework (as, e.g., [38]) given an appropriate adaptation of the cost model.

Even though the algorithms in this paper do not directly take skew into account, the presented framework can readily be adapted to do so when information on so-called heavy hitters is available or can be computed at the expense of an additional round (see, e.g., [22, 30, 32, 34]).

Acknowledgment. The third author was supported in part by grant no. NTU-ERP-105R89082D and the Ministry of Science and Technology Taiwan under grant no. 104-2218-E-002-038. The computational resources and services used in this work were provided by the VSC (Flemish Supercomputer Center), funded by the Research Foundation - Flanders (FWO) and the Flemish Government – department EWI. We thank Jan Van den Bussche and Jelle Hellings for inspiring discussions and Geert Jan Bex for assistance with cluster setup.

7. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] F. Afrati, M. Joglekar, C. Ré, S. Salihoglu, and J. Ullman. GYM: A multiround join algorithm in mapreduce. *CoRR*, abs/1410.4156, 2014.

[3] F. Afrati, A. D. Sarma, S. Salihoglu, and J. Ullman. Upper and lower bounds on the cost of a map-reduce computation. In *VLDB*, pages 277–288, 2013.

[4] F. Afrati and J. Ullman. Optimizing multiway joins in a map-reduce environment. *IEEE Trans. Knowl. Data Eng.*, 23(9):1282–1298, 2011.

[5] F. Afrati, J. Ullman, and A. Vasilakopoulos. Handling skew in multiway joins in parallel processing. *CoRR*, abs/1504.03247, 2015.

[6] H. Andreka, J. van Benthem, and I. Nemeti. Modal languages and bounded fragments of predicate logic. *Journal of Philosophical Logic*, 27(3):217–274, 1998.

[7] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. In *PODS*, pages 273–284, 2013.

[8] P. Beame, P. Koutris, and D. Suciu. Skew in parallel query processing. In *PODS*, pages 212–223, 2014.

[9] P. Bernstein and D. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, 1981.

[10] P. Bernstein and N. Goodman. Power of natural semijoins. *SIAM Journal on Computing*, 10(4):751–771, 1981.

[11] P. A. Bernstein and N. Goodman. The power of inequality semijoins. *Inf. Syst.*, 6(4):255–265, 1981.

[12] S. Blanas et al. A comparison of join algorithms for log processing in MapReduce. *SIGMOD*, pages 975–986, 2010.

[13] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43, 1998.

[14] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD*, pages 63–78, 2015.

[15] J. Daenen, F. Neven, and T. Tan. Gumbo: Guarded fragment queries over big data. In *EDBT*, pages 521–524, 2015.

[16] J. Daenen, F. Neven, T. Tan, and S. Vansummeren. Parallel evaluation of multi-semi-joins. *CoRR*, abs/1605.05219, 2016.

[17] J. Daenen and T. Tan. Gumbo v0.4, May 2016. `http://dx.doi.org/10.5281/zenodo.51517`.

[18] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[19] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch. Scalable and adaptive online joins. In *VLDB*, pages 441–452, 2014.

[20] J. Flum, M. Frick, and M. Grohe. Query evaluation via tree-decompositions. *J. ACM*, 49(6):716–752, 2002.

[21] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[22] A. Gates, J. Dai, and T. Nair. Apache pig's optimizer. *IEEE Data Engineering Bulletin*, 36(1):34–45, 2013.

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

[24] Y. Ioannidis. Query optimization. *ACM Computing Survey*, 28(1):121–123, 1996.

[25] P. Koutris and D. Suciu. Parallel evaluation of conjunctive queries. In *PODS*, 2011.

[26] D. Leinders, M. Marx, J. Tyszkiewicz, and J. Van den Bussche. The semijoin algebra and the guarded fragment. *Journal of Logic, Language and Information*, 14:331–343, 2005.

[27] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: Sharing across multiple queries in mapreduce. In *VLDB*, pages 494–505, 2010.

[28] A. Okcan and M. Riedewald. Processing theta-joins using MapReduce. In *SIGMOD*, pages 949–960, 2011.

[29] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX Annual Technical Conference*, pages 267–273, 2008.

[30] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[31] F. Picalausa, G. Fletcher, J. Hidders, and S. Vansummeren. Principles of guarded structural indexing. In *ICDT*, pages 245–256, 2014.

[32] S. R. Ramakrishnan, G. Swart, and A. Urmanov. Balancing reducer skew in mapreduce workloads using progressive sampling. In *SoCC*, page 16. ACM, 2012.

[33] Y. Tao, W. Lin, and X. Xiao. Minimal mapreduce algorithms. In *SIGMOD*, pages 529–540, 2013.

[34] 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*, pages 996–1005, 2010.

[35] M. Vardi. Why is modal logic so robustly decidable? In *DIMACS Workshop on Descriptive Complexity and Finite Models*, pages 149–184, 1996.

[36] G. Wang and C.-Y. Chan. Multi-query optimization in mapreduce framework. In *VLDB*, pages 145–156, 2013.

[37] T. White. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale*. O'Reilly, 2015.

[38] R. Xin, J. Rosen, M. Zaharia, M. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD*, pages 13–24, 2013.

[39] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981.